Journal of Computer Virology and Hacking Techniques (2021) 17:47–59
https://doi.org/10.1007/s11416-020-00364-w

ORIGINAL PAPER

Detection of anomalies in compiled computer program files inspired by immune mechanisms using a template method

Krzysztof Wawryn[1] **· Patryk Widuliński**[1]

Received: 12 December 2019 / Accepted: 5 August 2020 / Published online: 12 August 2020
© The Author(s) 2020

Abstract

An intrusion detection system inspired by the human immune system is described: a custom artificial immune system that monitors a local area containing critical files in the operating system. The proposed mechanism scans the files and checks for possible malware-induced alterations in them, based on a negative selection algorithm. The system consists of two modules: a receptor generation unit, which generates receptors using an original method based on templates, and an anomaly detection unit. Anomalies detected in the files using previously generated receptors are reported to the user. The system has been implemented and experiments have been conducted to compare the effectiveness of the algorithms with that of a different receptor generation method, called the random receptor generation method. In a controlled testing environment, anomalies in the form of altered program code bytes were injected into the monitored programs. Real-world tests of this system have been performed regarding its performance and scalability. Experimental results are presented, evaluated in a comparative analysis, and some conclusions are drawn.

Keywords Artificial immune system · Receptor · Anomaly · Negative selection algorithm · Template · Intrusion detection system · Malware · Virus

1 Introduction

Recently, the protection of computer software from virus attacks has become a very important task for software security designers. All over the world, intrusion detection systems (IDSs) are being intensively developed. IDSs are used to protect against network attacks on both the local and the global level. The main challenges for software designers are the detection of unknown attacks and the increasing complexity of computer software. Designers have been forced to search for novel, more effective software protection. They have noticed analogies between the tasks done by IDSs and those done by the natural immune system. Adaptation of immune system mechanisms gives a chance to meet the listed challenges. Plenty of new approaches based on artificial immune systems have appeared in the literature in the past two decades. Most of them adapt a negative selection method in their IDS to detect intrusions in computer network traffic. The negative selection method is based on the recognition of self and nonself structures, and leads to detecting anomalies in the computer software. The fundamental aspects of the mechanisms in IDS are presented in [1–5]. More sophisticated systems are described in [6–19].

In the present article, an IDS to search for infections in compiled computer programs is proposed. The detection mechanism is based on a negative selection method [3]. It is composed of receptor generation and anomaly detection algorithms. Programs monitored by the IDS are read in fragments of binary strings of a specific length. Our method of generating the receptors is an original method, using binary strings called templates, as described in [14,20]. A preliminary study of the use of template generation for this IDS was described by us in [21]. The present paper is a continuation of that one, expanding upon the template generation. New content includes a description of random generation, a comparative analysis between the template method and the random method, and also testing in a real-world scenario, with regard to performance and scalability. The generation

✉ Patryk Widuliński
patryk.widulinski@tu.koszalin.pl

Krzysztof Wawryn
wawryn@tu.koszalin.pl

[1] Department of Electronics and Computer Science, Koszalin University of Technology, ul. Śniadeckich 2, 75-453 Koszalin, Poland

of receptors relies on the division of the template structures into self templates and nonself templates. Nonself structures become receptors. Anomalies are detected by the receptors. The monitored program is infected if a desired number of bits is matched between a receptor and a read fragment of the program. The novelty of our approach comes from the employment of two separate sets of receptors to detect possible anomalies. The study discusses the possibility of the employment of the IDS in systems with small storage space and a requirement to protect the files in a specific directory.

This article is organized as follows. The proposed IDS is described in Sect. 2. The template method is presented in Sect. 3, a random-based method used for the sake of comparison is described in Sect. 4, and the experimental results are presented in Sect. 5. A comparative analysis is presented in Sect. 6, performance and scalability are discussed in Sect. 7, and concluding remarks are presented in Sect. 8.

2 Intrusion detection system

The present paper proposes an intrusion detection system (IDS) able to detect irregularities, called anomalies, in compiled computer programs.

An anomaly is a sequence of bytes in a program which has been altered by malware. An example of an anomaly in a program would be five NOP (no operation) CPU instructions (x86 machine code bytes $(9090909090)_{16}$) injected by malware instead of the original bytes, e.g., $(E900100000)_{16}$ (x86 machine code for a JMP instruction) to intentionally avoid the behavior of the original, unmodified program.

The IDS monitors computer file resources susceptible to infections in an operating system. For example, it could be the C:\Windows folder in a Microsoft Windows system. Monitored file resources have to be protected against infections by the IDS. In order to ensure the correct functioning of the IDS, the original noninfected programs have to be accessible during the first launch. All programs have to be in the Win32 Portable Executable (PE) format. Typically, a regular PE file consists of *sections*, whose purposes have specific characteristics (program code sections, initialized data, and uninitialized data), read/write access rights, and central processing unit (CPU) execution privileges.

The proposed IDS contains a main control block supervising the execution of the procedures for generating receptors and detecting anomalies. The system scans the folder of monitored PE files in search of code sections to generate binary strings of length l, called receptors, for each file in the folder. Receptors are able to recognize nonself structures in the program code. They are used to allow the detection of anomalies in the files in the monitored folder. Receptors, inspired by immune mechanisms (the negative selection algorithm, which was in turn inspired by the human immune system),

Read code fragment	00101100	**1101100100110101**	11000110
Generated receptor	01001011	**1101100100110101**	11101111

Read code fragment	2C	**D9 35**	C6
Generated receptor	4B	**D9 35**	EF

Fig. 1 Example of matching of generated receptors and fragments of read program for $l = 32$, $m = 16$, and $k = 8$. Top: binary representation, bottom: hexadecimal representation

possess an activation threshold, denoted by m. It defines the minimal number of subsequent receptor bits which must match the read program fragment to activate the receptor; otherwise the receptor is not activated. Receptors detect anomalies in the program. This means that receptors must not recognize self structures. In the generation procedure, every candidate receptor structure matching any self structure has to be rejected, otherwise they are accepted as receptors. In our IDS system, the PE file is opened by using the system interface, and delivers information about the code sections. The IDS generates possible receptor structures and compares them to all read PE program code fragments of length l in the monitored file. The receptor generation has to be done once for each of the monitored program files, producing one receptor set for each program file. A generated receptor set can then be stored as a file or remain in the system memory, and can be used by the anomaly detection algorithm.

In the receptor generation procedure, the activation threshold m is also called a window. The current position of the window in the receptor is called the *window shift* and denoted by k. If the currently generated structure matches at least m bits in even a single monitored fragment of program code (i.e., a matching of m bits between the structure and the program with any window shift k), then it is rejected, and cannot be used as a receptor to detect anomalies, because it recognizes self code as well. An example of a matching between the read code of the program and a generated structure is shown in Fig. 1. When the generated structure does not match any self structures, it becomes a receptor and is appended to the set of receptors. The number of receptors in the set of receptors is denoted by R_n. The set of receptors is now used to detect anomalies in the possibly infected file. An anomaly is detected when a receptor matches any fragment of a possibly infected fragment of the program (i.e., matching of m bits between receptor and program with any window shift k). In the anomaly detection algorithm, every receptor is compared with every fragments of a possibly infected program.

Our IDS adopts an immunity algorithm inspired by negative selection, relying on a novel method that uses templates.

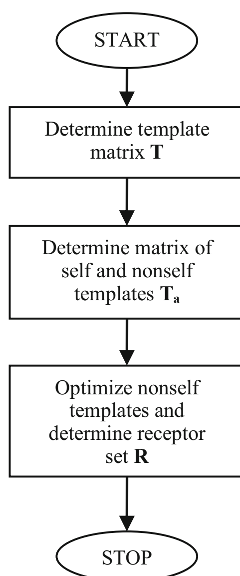

Fig. 2 A flowchart of the receptor generation algorithm

Our novel method allows the detection of anomalies placed not only within particular memory cells, but also between consecutive program memory cells.

3 The template-based method

3.1 The generation of the receptors

A method of generating receptors that was based on templates was introduced in [20,22]. Here, appropriate modifications to that method are proposed to generate receptors that will efficiently detect intrusions in the operating system (Fig. 2).

A template is defined as a binary string consisting of a fixed value and don't care bits. Fixed value bits have a logical value of 0 or 1. Don't care bits are labelled with an asterisk ("*").

In the first step of the generation algorithm, a table, denoted by **T**, is built. **T** is composed of templates of length l bits. For a given activation threshold, denoted by m, the templates consist of m fixed value bits. The rest of the bits (l–m) in the template are don't care bits.

The number of templates is denoted by L_s and is expressed as follows:

$$L_s = (l - m + 1) \cdot 2^m \quad (1)$$

For $l = 16$ and $m = 8$ and for $l = 32$ and $m = 8$, L_s equals $9 \cdot 256$ and $25 \cdot 256$, respectively. These numbers are too large to illustrate the idea of the proposed algorithm, therefore, an example of **T** for $l = 6$ and $m = 4$ is presented in Table 1.

Table 1 Template table **T** for $l = 6$ and $m = 4$

i	m given bits	**T**[i,1]	**T**[i,2]	**T**[i,3]
1	0000	0000**	*0000*	**0000
2	0001	0001**	*0001*	**0001
3	0010	0010**	*0010*	**0010
4	0011	0011**	*0011*	**0011
14	1101	1101**	*1101*	**1101
15	1110	1110**	*1110*	**1110
16	1111	1111**	*1111*	**1111

Table 2 Table $\mathbf{T}_a$ for $l = 6$ and $m = 4$

i	m given bits	$\mathbf{T}_a[i, 1]$	$\mathbf{T}_a[i, 2]$	$\mathbf{T}_a[i, 3]$
1	0000	1	1	1
2	0001	1	1	1
3	0010	1	1	1
4	0011	1	1	1
5	0100	1	1	1
6	0101	1	0	1
7	0110	1	0	1
8	0111	0	0	1
9	1000	1	1	1
10	1001	1	1	1
11	1010	0	1	0
12	1011	0	1	0
13	1100	1	1	0
14	1101	1	1	0
15	1110	1	0	0
16	1111	1	1	1

Consider the following example of a set **S** of own binary strings (noninfected program fragments):

$$\mathbf{S} = \{101110, 101101, 101100, 101011, 101010, 011100\} \quad (2)$$

The table $\mathbf{T}_a$ is constructed from **T** taking into account the set **S**. The table $\mathbf{T}_a$ consists of self and nonself templates. If at least $m = 4$ of the subsequent bits of the template from **T** match at least one self template from **S** at the same position, the template is considered as an own (self) template and denoted by 0 in $\mathbf{T}_a$, otherwise the template is considered as nonself and denoted by 1 in $\mathbf{T}_a$.

For example, $\mathbf{T}[14, 3] =$ **1101 matches $\mathbf{S}(2) = 101101$, and so, $\mathbf{T}_a[14, 3] = 0$, while $\mathbf{T}[2, 3] =$ **0001 does not match any self template in **S**, therefore $\mathbf{T}_a[2, 3] = 1$ (Table 2).

In this case, every template represents four possible receptors. An example of receptor generation from nonself

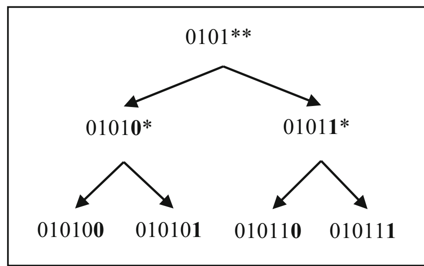

Fig. 3 Generation of receptors from **T**[6, 1] = 0101**

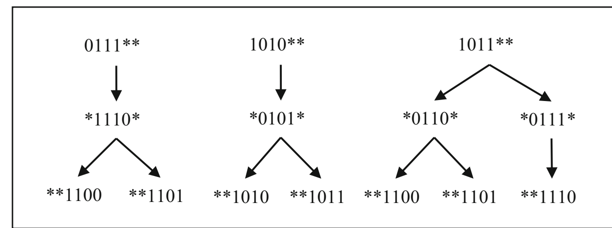

Fig. 4 Graphical representation of self templates coming from the set **S** defined by (2)

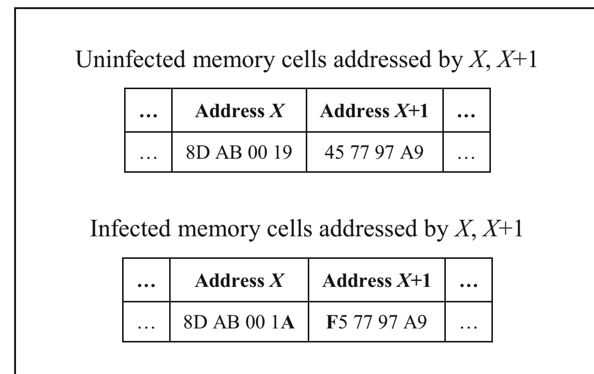

Fig. 5 An example of a one-byte anomaly appearing between 32-bit memory cells

T[6, 1] = 0101** is illustrated in Fig. 3. It is based on a binary tree. The two first leaves have four fixed bits of the template, and their first don't care bits ("*") are replaced by 0 or 1. The leaves that are obtained are 01010* and 01011*. Finally, when the last "*" bits are replaced by 0 or 1, the following four leaves are obtained: 010100, 010101, 010110 and 010111. Those leaves become receptors, because they don't match any self templates.

There are four leaves, 010100, 010101, 110100, and 110101, built from the nonself template **T**[11, 2] = *1010*. Two leaves (receptors), 010100 and 010101, are the same as leaves of the nonself template **T**[6, 1] and are redundant. The total number of possible receptors is four times the number of nonself templates in $\mathbf{T}_a$. There are plenty of redundant receptors obtained from $\mathbf{T}_a$. A method to eliminate redundant receptors is desired. To satisfy this requirement, any binary tree of a nonself template $\mathbf{T}_a[i, 1]$ is compared with the binary trees of all the nonself templates $\mathbf{T}_a[i, 2]$ and finally, to the binary trees of all the nonself templates $\mathbf{T}_a[i, 3]$, eliminating redundant branches and leaves. After eliminating all redundant branches and leaves, the set of leaves becomes the set of receptors **R**.

For the set of self binary strings **S** defined by (2), **R** ={000000, 100001, 000010, 100011, 000100, 100101, 000110, 100111, 001000, 001001, 010000, 110001, 010010, 110011, 010100, 110101, 010110, 110111, 011000, 111001, 111111}. All 21 receptors from **R** will be used to detect anomalies.

Let **D** be a set containing all possible binary strings of length l. Typically, for a specified value of m and a specified set **S**, there will be binary strings which are unable to be detected by any generated receptor. The existence of such binary strings in the system is caused by the so called holes. The reason for existence of such holes is the presence of strings in the set $\mathbf{D} \setminus \mathbf{S}$ that have been constructed using self templates from the set **T**.

It is possible to calculate the number of holes in the system for a given m and **S** by constructing a graph representing the binary tree of self templates. A graph for **S** defined by (2) and $m = 4$ is shown in Fig. 4. Unlike during the process of building receptors, in the process of calculating the number of holes all possible routes from the roots to the leaves are taken into consideration. In Fig. 4, it can easily be seen that there are seven routes from roots to leaves. Therefore, because there are only six self binary strings in **S** defined by (2), there is one hole in the system. The number of holes can be reduced by increasing m, which also increases the computational time and memory footprint of the receptor set **R**. For example, increasing m from four to five bits for **S** defined by (2) causes the only hole in the system to close, reducing the possible routes from roots to leaves from 7 to 6. However, this operation also increases the receptor count from 21 to 50.

In our method, receptors of length $l = 32$ bits and activation threshold $m = 16$ bits are generated. The size of the read cells equals 4 bytes. The method based on these parameters is called S-32, corresponding to the number of receptor bits from **R**. This set of receptors is able to detect an anomaly inside a single 4-byte memory cell, and is unable to detect anomalies among neighboring memory cells (for example beginning from the least significant byte of the memory cell addressed by X, and finishing at the most significant byte of the memory cell addressed by $X + 1$ as illustrated in Fig. 5). Our system takes a novel approach by generating an extra set of intercellular receptors, denoted by $\mathbf{R}_I$. $\mathbf{R}_I$ consists of receptors of length $l = 16$ bits and activation threshold $m = 8$ bits generated on the basis of pairs of subsequent neighboring 4-byte memory cells. To generate receptors for

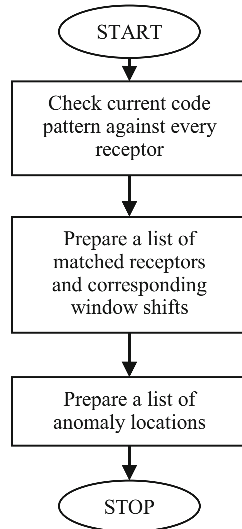

Fig. 6 A flowchart of the anomaly detection algorithm

$\mathbf{R}_I$, only the least significant byte of the memory cell and the most significant byte of the subsequent memory cell are taken into account. The template method using regular 32-bit receptors together with our new intercellular receptors is denoted by ICR.

It may be observed that the leaves of the binary trees of the templates discarded as receptors matching self structures at the end of program may actually be able to detect anomalies at the beginning of the program. Therefore, the section method (SM) is also proposed. SM divides the tested program into four subprograms and tests them separately. As a result, there is no collision between receptors from each of the four sections. The SM approach uses 32-bit templates (S-32). The combination of the SM approach and ICR is denoted by SM + I.

3.2 Detection of anomalies

The proposed system detects anomalies in the secured region by the use of the previously generated 32-bit receptors stored in $\mathbf{R}$ (method S-32) and optionally using the 16-bit receptors stored in $\mathbf{R}_I$ (method ICR). In contrast to receptor generation, anomalies are detected in the up-to-date, possibly infected, version of the program. The first 32-bit memory cell of the tested program is read and compared with every 32-bit receptor with activation threshold $m = 16$ from $\mathbf{R}$. Each time the read memory cell and receptor match m subsequent bits with the same window shift k, an anomaly is detected and a message is displayed for the user. If the memory cell is compared with all receptors and not one receptor is matched, there is no anomaly inside the memory cell. After comparison with all receptors, the next 4-byte memory cell is compared with all receptors, et cetera, until all memory cells have been compared with all receptors (Fig. 6).

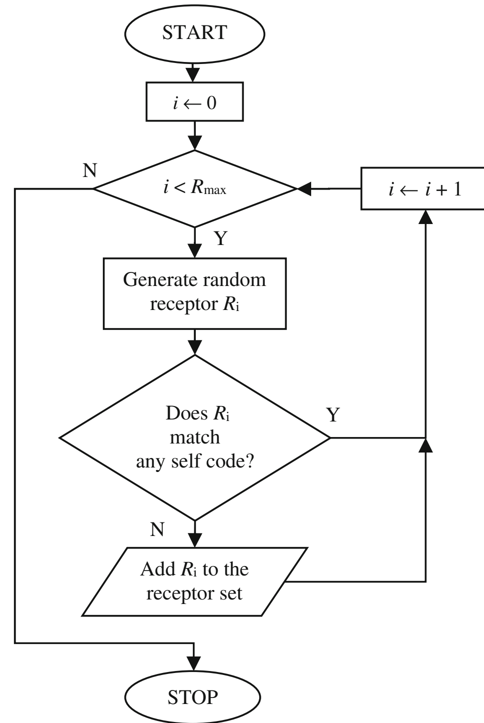

Fig. 7 A flowchart of the random receptor generation algorithm

4 The random-based method

To test the benefits gained from using a modified template method, a frequently used method of random generation of receptors has also been implemented in the IDS.

4.1 Random generation of receptors

A flowchart of the random receptor generation algorithm is shown in Fig. 7. The number of generated receptors is given by the user and denoted by R_{max}. The maximal number of possible random receptors generated by the system for single file is defined by the relation

$$max(R_{max}) = 2^l \tag{3}$$

where l denotes the length of the receptor in bits. 8-bit pseudorandom numbers from the range [0; 255] are generated and used to create receptors of size l. The number of 8-bit receptor fragments is denoted by

$$f = ceil(div(l, 8)) \tag{4}$$

where the $div(a, b)$ operation denotes division of a by b and the function $ceil(a)$ denotes the ceiling of the number a. The generated 8-bit fragments are concatenated to obtain a random receptor.

a)	01001011	11011001	00000101	1110**1111**
b)	4B	D9	05	E**F**

Fig. 8 An example of random receptor generation for $l = 28$, $f = 4$. Redundant digits are denoted in bold: **a** binary representation and **b** hexadecimal representation

a)	11100001	10011010
b)	E1	9A

Fig. 9 An example of random receptor generation for $l = 16$, $f = 2$, no redundant digits: **a** binary representation and **b** hexadecimal representation

If l is not divisible by 8, there are redundant bits in the receptor. Redundant bits are stored in memory, but not used in the generation and detection algorithms.

Two examples of random receptor generation are shown in Figs. 8 and 9. The example shown in Fig. 8 illustrates generated pseudorandom 4-byte numbers: $(4B)_{16}$, $(D9)_{16}$, $(05)_{16}$ and $(EF)_{16}$. They have been concatenated, resulting in a 32-bit number $(4BD905EF)_{16}$. However, $l = 28$, therefore 4 bits of this number will not be used in the proposed algorithms. In the example shown in Fig. 9, there are no redundant bits, because $l = 16$ is divisible by 8. Two pseudorandom 8-bit numbers $(E1)_{16}$ and $(9A)_{16}$ result in the 16-bit number $(E19A)_{16}$. Both resulting numbers from Figs. 8 and 9 may become receptors if they do not match any self structures.

4.2 Detection of anomalies

The random receptor generation algorithm has been used to detect anomalies in the same monitored program as in the case of template anomaly detection algorithm. In the beginning, the receptors in the set of receptors are counted and the number of all receptors is denoted by R_n. To detect anomalies, the algorithm reads fragments of program code of size l, and compares them with subsequent receptors in the same manner as in the case of the template intrusion detection algorithm.

If m subsequent bits are matched, the receptor is activated and an anomaly is detected. Information about the anomaly and its location is sent to the user.

5 Experimental results

The proposed system has been implemented in C# to test and verify the effectiveness of its anomaly detection. Two kinds of experimental tests were conducted.

The first test, called the benchmark test, involved a small sample Win32 PE executable file being manually injected with anomalies. The anomalies had random content, were placed at random file positions, and were of increasing size. The benchmark test was performed for two receptor generation methods: a template-based method (four variations) and random generation.

The second test involved real-world malware infecting the monitored files, which were modules of the Microsoft Windows operating system.

5.1 Benchmark tests

For the benchmark testing, an example file called “test.exe” with a size of 6584 bytes was written in C++ in Win32 PE format. The sample file displays a *Test* message on the screen before closing.

Since real-world infections usually modify the original binary data on various file positions, the approach for selecting the anomaly data used in this research allowed for file-position independent anomaly detection. The anomalies introduced into the program had sizes from the range [1 B; 8 B], incrementally with a one byte step. This range allowed testing the detection system in the worst case scenario, which would be a one-byte alteration by the infection in the program, as well as slightly better case scenarios with anomalies of sizes 2 B and up.

A *detection attempt* is the scanning of the entire program file in its current version using the receptors generated beforehand. Knowing that under the test conditions an anomaly has definitely been introduced in the program, a detection attempt can either yield a positive result (which would be counted as a successful detection in the statistics) or a negative result. For each size of the anomalies, 100 detection attempts were made.

The content of each anomaly bytes was generated randomly, which allowed testing the system against unknown infections. In the real world, the infected program bytes would not be as diverse, because they would constitute machine code bytes, which could be repetitive and limited.

For example, when it comes to an anomaly size of 1 B, a one-byte anomaly with random content was introduced in the program at a random position. A detection attempt would follow. This process was then repeated for a total of 100 times for that particular anomaly size, each time with new, random anomaly content and a random position to yield a detection rate for that specific size, independent of the content and location of the anomaly.

5.1.1 Template-based methods

The following template-based methods were used for the testing: the method of 32-bit templates (S-32), the method of 32-bit templates and 16-bit templates (ICR intercellular receptors), the section method (SM), and a method com-

Table 3 S-32 method, $R_n = 32$

Size of anomaly (B)	MAR	DT_{avg}	D (%)
1	4	12	30
2	3	22	50
3	8	8	20
4	2	13	30
5	14	21	50
6	6	26	60
7	6	26	60
8	8	35	80
Average	6.375	20.375	47.50

Table 4 ICR method, $R_n = 46$

Size of anomaly (B)	MAR	DT_{avg}	D (%)
1	4	12	30
2	3	22	50
3	11	8	40
4	11	13	70
5	14	21	50
6	6	26	60
7	6	26	60
8	11	8	100
Average	8.25	17	57.50

Table 5 SM method, $R_n = 780$

Size of anomaly (B)	MAR	DT_{avg}	D (%)
1	5	10	60
2	7	17	100
3	9	16	90
4	11	20	100
5	13	20	100
6	15	17	100
7	24	16	100
8	20	19	100
Average	13	16.875	93.75

bining the ICR and SM approaches, denoted by SM+I. The following parameters were used: $l = 32$ and $m = 16$ for method S-32, $l = 32$ and $m = 16$ for the receptors $\mathbf{R}$ and $l = 16$ and $m = 8$ for the intercellular receptors $\mathbf{R}_I$ in the ICR method, and $l = 32$ and $m = 8$ for the SM method with the number of sections equal to four.

The results of the experiments are presented in Tables 3, 4, 5, and 6, where MAR denotes the maximum number of activated receptors for 100 attempts, DT_{avg} denotes the average time needed for an anomaly detection in milliseconds, and D denotes the percentage of anomalies detected.

Table 6 SM+I method, $R_n = 870$

Size of anomaly (B)	MAR	DT_{avg}	D (%)
1	6	9	90
2	9	15	100
3	10	13	100
4	13	17	100
5	14	12	100
6	19	13	100
7	27	12	100
8	33	11	100
Average	16.375	12.75	98.75

Table 7 Random method, $R_{max} = 1000$, $R_n = 927$

Size of anomaly (B)	MAR	DT_{avg}	D (%)
1	1	1046	10
2	1	1030	20
3	1	1041	40
4	1	1020	20
5	1	1025	30
6	3	1037	20
7	2	1050	20
8	2	1045	20
Average	1.5	1036.75	22.50

Table 8 Random method, $R_{max} = 5000$, $R_n = 4693$

Size of anomaly (B)	MAR	DT_{avg}	D (%)
1	1	5396	30
2	2	5321	50
3	2	5313	40
4	3	5311	70
5	4	5298	90
6	4	5264	60
7	4	5348	80
8	5	5211	60
Average	3.125	5307.75	60

5.1.2 Random-based method

The test conditions were the same as before. The experimental tests were carried out using the following values of the parameters: $l = 32$ and $m = 16$. The results are shown in Tables 7, 8, and 9.

5.2 Real-world tests

The system was tested in two real-world scenarios. The first of the two tested scenarios was a single malware sample

Table 9 Random method, $R_{max} = 10{,}000$, $R_n = 9427$

Size of anomaly (B)	MAR	DT_{avg}	D (%)
1	2	10,072	40
2	3	10,017	40
3	2	10,020	90
4	3	10,072	60
5	4	10,063	90
6	3	9952	100
7	3	10,216	100
8	5	9995	100
Average	3.125	10,050.875	77.50

launched in an environment monitored by the proposed system. The second scenario was a stress test involving a virtual machine and ten malware samples.

5.2.1 Single malware sample test

A monitored (secure) folder was set up with three important Microsoft Windows operating system modules inside—"secinit.exe": a module initializing the security in the operating system, "backgroundTaskHost.exe": a module for hosting background tasks in the operating system, and "regsvr32.exe": a program for the Microsoft Register Server.

After being launched, the proposed IDS generates a separate receptor set for each of the three monitored files in their initial, unmodified state. Two variations of template-based methods were used for the receptor generation: the S-32 method with parameters $l = 32$ and $m = 10$, and a variation with parameters $l = 16$ and $m = 10$. After the receptor sets were ready, a real-world malware sample ("padania.exe") was launched in the operating system. The malware sample chosen for this purpose is a file infector named "Virus:Win32/Padania!epo" (Microsoft Security Intelligence name) with a size of 8192 B. The malware is a memory resident virus writing itself to the EXE files it finds in the system, overwriting the .reloc section of the EXE files with malicious code.

The proposed IDS then detected the infections in the monitored files. The results of the detections are presented in Tables 10 and 11, where TGT is the template generation time in ms, R_n is number of receptors generated, RGT is the receptor generation time in ms, ADT is the anomaly detection time in ms, MC is the number of matched receptors, and RM is the receptor memory footprint in bytes. It can be observed that the system reached a 100% detection rate in a real-world scenario involving the Padania virus.

5.2.2 Virtual machine stress test

To illustrate the false positive and false negative rates of our proposed approach, the system was tested inside a Windows virtual machine (VM). The system was installed in the VM and was set up to monitor the file "backgroundTaskHost.exe" described in Sect. 5.2.1 using the parameters $l = 16$ and $m = 10$.

The stress test involved launching ten different malware programs, randomly chosen from a virus research database, in the environment of the virtual machine, sequentially, and checking whether the infection of the monitored file by each malware was detected by the system. A lack of detection of the infection is called a false negative and occurs when the detected anomaly count does not change from the previous infection even though the monitored file has been modified by the malware.

On the other hand, a detection that should not have occurred is called a false positive. It is important to remark that any modifications to the monitored files without regenerating the receptors would be marked as an intrusion, causing false positives. It is therefore imperative to regenerate the receptor set every time the files are modified in a legitimate

Table 10 Virus detection results, $l = 16$, $m = 10$

Monitored file name	Size (B)	TGT (ms)	R_n	RGT (ms)	ADT (ms)	MC	RM (B)
secinit.exe	10,044	3185	680	26,295	2482	192	1360
backgroundTaskHost.exe	17,720	5616	45	176	252	5	90
regsvr32.exe	25,404	7296	350	5241	2540	6	700

Table 11 Virus detection results, $l = 32$, $m = 10$

Monitored file name	Size (B)	TGT (ms)	R_n	RGT (ms)	ADT (ms)	MC	RM (B)
secinit.exe	10,044	5079	1015	246,374	5350	611	4060
backgroundTaskHost.exe	17,720	8924	289	17,660	1616	25	1156
regsvr32.exe	25,404	11,437	606	74,296	4552	19	2424

fashion, for example when they are updated with newer versions. This behavior was also tested in this experiment.

The results of the VM stress tests, based on the logs from the proposed IDS, are presented in Table 12, where $T+$ denotes the number of seconds since the start of the test, "Event" highlights a specific event that happened at the time, FP and FN denote the total number of false positives and false negatives counted until that event so far, respectively, and AC denotes the current detected anomaly count at the time of the event. Microsoft Security Intelligence malware names have been chosen for use as such in this paper.

The experiment timer was started with the launch of the system in the VM and stopped when all malware samples had been tested. The experiments ran for 284.2 s in total. It can be observed that generating the templates and receptors took 5.2 s, after which the monitoring for anomalies started. At $T + 23$ s the monitored file was updated with a newer version by the user. This generated a false positive anomaly detection 0.1 s later due to the nature of the system, which has to be informed about legitimate modifications to the file. The receptor regeneration command was then issued, and after the algorithms completed their work, the monitoring restarted 5.4 s later.

The malware tests started at $T + 84.7$ s, when the first malware sample, Win32/Sality.F, was executed in the virtual machine. The proposed system detected the infection 7.7 s later with 910 anomaly bytes being detected by the algorithm. The system correctly identified four infections, after which it produced a false negative at $T+190.7$ s. The Win32/Alma.A and Win32/Kvex.A samples were then detected correctly, after which the system produced a false negative on one more malware sample. The monitoring was stopped at $T+284.2$ s after the last sample, Win32/Ataxia.B had been tested positively.

It can be observed that the total number of false positives in this experiment was 1 and the total number of false negatives was 2. Therefore, because 11 tests in all were performed (including the monitored file update at $T+23$ s), the proposed system achieved a 73% success rate with regard to detecting random infections in a real-world scenario.

6 Comparative analysis

In the benchmark tests, the generation of receptors was done for the uninfected program for every method mentioned before: S-32, ICR, SM, SM + I, and random. The time needed for generating the receptors (except for the random receptor generation) was 140 s for S-32, 175 s for ICR, 290 s for SM, and 325 s for SM + I. The time needed for the random receptor generation was 1 s for $R_{max} = 1000$, 5 s for $R_{max} = 5000$, and 10 s for $R_{max} = 10000$. The detection rates as functions of the anomaly size for the template methods are plotted in Fig. 10 and for the random method in Fig. 11.

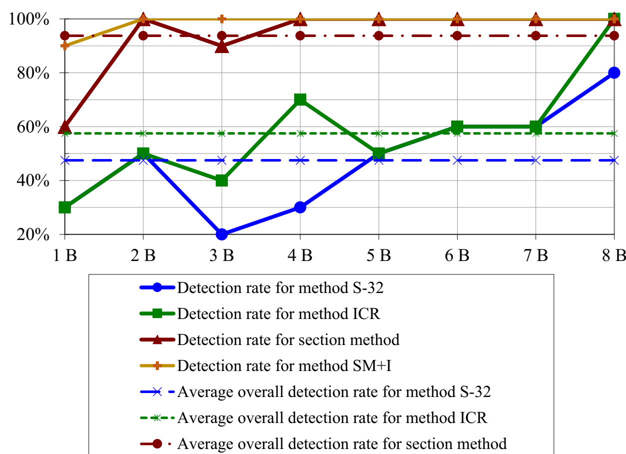

Fig. 10 Detection ratio as a function of anomaly size for template methods

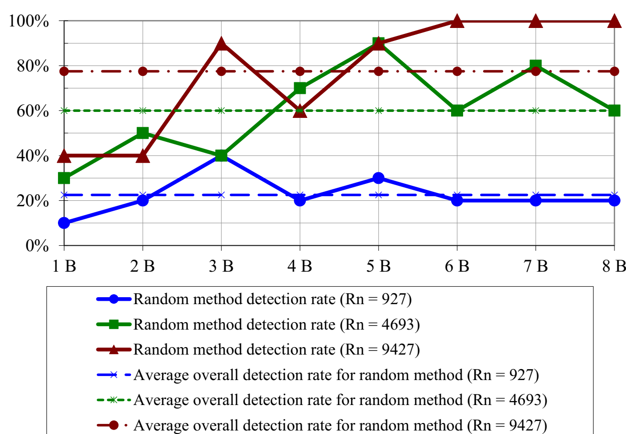

Fig. 11 Detection ratio as a function of anomaly size for the random method

For the tested file "test.exe" 0.23 receptors were generated per second by the S-32 method, and 1003.1 receptors per second by the random method. Although receptors are generated in only one second in the random method, the detection ratio is significantly lower: 22.5% for $R_{max} = 1000$ in comparison to 47.5% for the S-32 method. Increasing R_{max} to 5000 improves the detection ratio to 60%, at the expense of the memory footprint. The implementation of 16-bit receptors in ICR increased the average detection ratio to 57.5% and decreased the average detection time from 20.4 to 17 ms. The memory footprint was increased by an additional 14 16-bit receptors in $\mathbf{R}_I$. The ICR method uses only 464 bytes of operational memory cells, while the random method for $R_{max} = 5000$ uses 4693 memory cells. The average detection time is 17 ms for the ICR method, and 5307.75 ms for the random method. The time needed to generate the receptors equals 120 s for the ICR method, and 5 s for the random

Table 12 Virtual machine test results

T+ (s)	Event	FP	FN	AC
0	Generation started	0	0	0
5.2	Generation finished	0	0	0
5.2	Monitoring started	0	0	0
23	Modified a monitored file	0	0	0
23.1	Anomalies detected	1	0	6
43.3	Regeneration command issued	1	0	6
43.3	Monitoring paused	1	0	6
43.3	Regeneration started	1	0	6
48.7	Regeneration finished	1	0	0
48.7	Monitoring restarted	1	0	0
84.7	Executed Virus:Win32/Sality.F	1	0	0
92.4	New anomalies detected	1	0	910
117.8	Executed Virus:Win32/Amelg.A	1	0	910
118.2	New anomalies detected	1	0	931
143	Executed Virus:Win32/Delf.T	1	0	931
143.3	New anomalies detected	1	0	599
161.8	Executed Virus:Win32/Arcer.A	1	0	599
169.3	New anomalies detected	1	0	600
180.7	Executed Virus:Win32/Horope.A	1	0	600
190.7	No new anomalies detected	1	1	600
191.7	Executed Virus:Win32/Alma.A	1	1	600
192	New anomalies detected	1	1	602
208.1	Executed Worm:Win32/Kvex.A	1	1	602
208.5	New anomalies detected	1	1	1184
228.7	Executed Virus:Win32/Velost.1186	1	1	1184
253.1	No new anomalies detected	1	2	1184
254.1	Executed Virus:Win32/Jeefo.F	1	2	1184
254.7	New anomalies detected	1	2	1599
274.1	Executed Virus:Win32/Ataxia.B	1	2	1599
275.7	New anomalies detected	1	2	1687
284.2	Monitoring stopped	1	2	1687

method. The average detection ratio is comparable for both methods. It can be observed that the template methods generate fewer receptors per second, and they occupy less memory cells. This gives a shorter average detection time than in the random method. Upon increasing R_{max} to 10,000 in the random method, the average detection ratio increased to 77.5%. By the use of the SM method, the detection ratio increased to 93.75%, the number of receptors increased from 32 to 780, increasing the memory use and the time needed to generate the receptors to 150 s. It can be also observed that the detection ratio for S-32 increases when the size of the anomaly increases: for a 3-B anomaly it is 30% and for a 4 B one it is 60%. In the case of the random method for $R_{max} = 1000$, the detection ratio for anomalies above 3 B was approximately 30%. It can be observed that combining the section method (SM) with ICR (producing the SM + I method) achieved an average detection rate of 98.75%, which is the best result of all the aforementioned methods, but comes at the price of high receptor generation times and a larger memory footprint. The analysis shows that our novel ICR approach grants better detection rates in comparison with S-32 or random generation approaches for the IDS. Because 100 detection attempts were conducted for each anomaly size, the tests have a greater reliability with regard to achieving the same pattern of results in all common cases.

7 Performance and scalability

Based on the experiments presented in this paper, the proposed system has its advantages as well as its limitations. When it comes to real-world performance, generating the

templates for the monitored files took between 3 s at best and 11.4 s at worst, depending on the sizes of the receptors and the files. The template generation time increased by roughly 2 s per each 7 kB of program data for 16-bit receptors and by about 3.5 s for each 7 kB of program data for 32-bit receptors.

The template generation times increase with file size and receptor bit size, but the receptor generation times decrease with file size, along with the receptor count. This is because with a bigger file, there is a greater chance that it will have more unique memory cells, shrinking the pool of possible receptors, as they cannot detect self structures. The fewer the receptors, the smaller the possibility of detecting an infection. Hence, monitored files that have a larger entropy (e.g., compressed or encrypted infected files) will yield worse detection rates with the proposed system than they would do if they had a smaller entropy. This means that the system works better with files that have less diverse content.

As the monitored file size increases, fewer and fewer receptors are generated, until eventually the count becomes 0 and it is impossible to detect any intrusions for the given parameters l and m. This places a limitation on the monitored file size, for the given parameters. The solution to this problem is to increase l and m. Another limitation, however, is the fact that as observed in Tables 10 and 11, doubling the value of the parameter m from 16 to 32 caused approximately 10 times longer receptor generation times. By continuously increasing m, a point might be reached at which generating the receptors takes too long for the proposed system to remain viable in the chosen environment.

The viability and validity of the usage of the system relies on conditions such as the speed of the CPU, storage space, and size of the installed volatile memory. The receptors generated by the proposed IDS may be stored in both non-volatile and volatile memory, and their memory footprint can be vastly smaller than the monitored file itself, as evidenced by research data in Table 10 (e.g., only 90 bytes of receptor memory for a file of size 17720 B). This allows users employing the proposed IDS to exchange some CPU usage for smaller storage occupancy, as opposed to a system where a copy of the original executable might be kept at hand at all times, increasing the storage occupancy.

The effect of the IDS on overall system performance was measured during the virtual machine stress tests. The measured parameters were the CPU usage (%) and the memory usage (MB). The measurements are presented in Figs. 12 and 13. The CPU usage in a situation with no infections is at an average of 8% and the memory usage is about 19.7 MB. The drops to near zero CPU usage every second take place because after scanning all the monitored files, the IDS pauses for a second before it resumes scanning from the beginning. It can be observed that the CPU usage in Fig. 12 spiked to 70% twice: at about 4 s, and at about 44 s into the experiment.

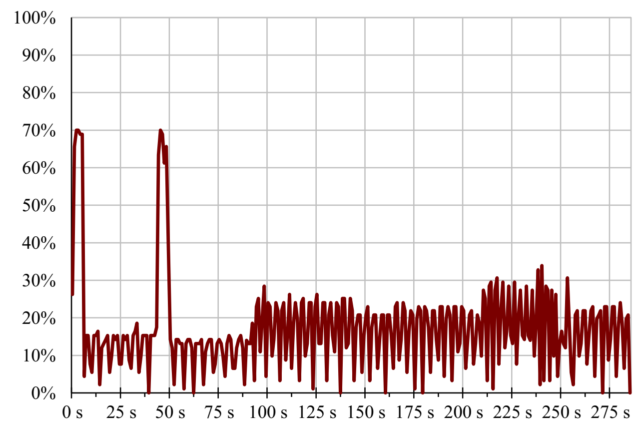

Fig. 12 CPU usage by the proposed IDS in the virtual machine stress test as a function of time

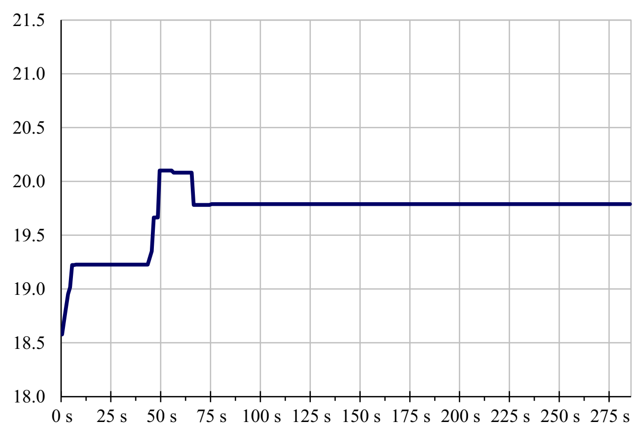

Fig. 13 RAM usage (MB) by the proposed IDS in the virtual machine stress test as a function of time

This is because although the template and receptor generation algorithms have to be called rarely, they are CPU intensive. It can also be observed that the IDS CPU usage rose to 25%–30% after about 100 s into the experiment, when the virtual machine was under very heavy malware load. One of the reasons for this behavior is the use of modest VM parameters for the experiment (512 MB RAM, 1 CPU core @ 3.2 GHz).

The IDS memory usage started at 18.5 MB, before the initial generation of the receptors. This was the memory cost of the C# runtime and the graphical user interface of the proposed system. It can be observed that the IDS memory usage rose by 0.7 MB each time the receptors were generated (at 5 s, and at 50 s). The increases in memory footprint were due to the requirement to store the newly generated set of receptors and because of the C# object memory overhead (i.e. the necessity to store extra information about the generated objects for runtime garbage collection and other purposes). A decrease in memory usage by 0.4 MB was recorded at 65 s. The decrease was due to the C# runtime garbage collection

routines eliminating receptors (along with their overhead) that were not used anymore after the regeneration. After eliminating the unused receptor objects the memory size remained at 19.7 MB until the end of the experiments. The average IDS CPU usage in the virtual machine experiment was 17.3%, the maximum CPU usage reached 70%, the average memory usage was 19.7 MB, and the maximum memory usage was 20.1 MB. The CPU and memory usage by the IDS could be reduced by implementing the algorithms in a lighter language, like C.

The proposed system can be applied not only for malware detection. For example, the proposed system could work in a small operating system environment where it is imperative to protect certain files from unauthorized modification, such as firmware or bootloader files in embedded systems, where the amount of installed storage space may be limited and conserving it is desired.

Another use case of the proposed IDS would be having a requirement to know where the anomaly has occurred in the monitored program. Learning where the anomaly occurred in the program would normally require storing a backup of the original program file for comparison. The system allows the detection of program anomalies in systems where it is desired to know the location of anomalies in the programs, and storing backups of unmodified programs is not desired, for example in limited disk space scenarios. The receptors use vastly smaller amounts of memory in comparison to storing backups of unmodified programs.

The approach used by the proposed IDS, an immune based approach, allows detecting unknown infections, in contrast to non-immune, signature-based approaches, which do not possess the ability to search for unknown infections.

8 Concluding remarks

The proposed system adopts an artificial immune mechanism called a negative selection algorithm to detect anomalies in program files. For the purpose of comparison, the negative selection algorithm was implemented in two versions: the novel template receptor generation proposed in the present paper, and random generation of receptors. Series of tests were carried out for both versions. The results show that our novel algorithm, based on templates, provides better detection ratios, shorter detection times, and lower memory use than the random generation algorithm. The most effective variant of this template method was the section method with intercellular receptors (SM + I). Real-world tests were conducted with real malware, and the results show the viability of the system.

Future work will be carried out to discover the dependence of the detection rate as a function of the number of bits of the receptors and the size of the activation threshold. The problem of holes in the system will also be explored further.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Somayaji, A., Forrest, S., Hofmeyr, S., Longstaff, T.: A sense of self for Unix processes. In: IEEE Symposium on Security and Privacy, pp. 120–128 (1996)
2. Somayaji, A., Hofmeyr, S., Forrest, S.: Principles of a computer immune system. In: New Security Workshop, pp. 75–82 (1997)
3. Forrest, S., Perelson, A., Allen, L., Cherukuri, R.: Self-nonself discrimination in a computer. In: IEEE Symposium on Security and Privacy, pp. 202–212. IEEE Computer Society (1994)
4. Kephart, J.: A biologically inspired immune system for computers. In: Fourth International Workshop on Synthesis and Simulation of Living Systems, Artificial Life IV, pp. 130–139 (1994)
5. Dasgupta, D.: Immunity-based intrusion detection systems: a general framework. In: 22nd National Information Systems Security Conference (NISSC), Arlington, Virginia, USA, pp. 147–160 (1999)
6. Andrews, P., Timmis, J.: Tunable detectors for artificial immune systems: from model to algorithm. In: Bioinformatics for Immunomics, pp. 103–127. Springer, New York (2010)
7. Sobh, T., Mostafa, W.: A cooperative immunological approach for detecting network anomaly. Appl. Soft Comput. **11**, 1275–1283 (2011)
8. Wang, D., Zhang, F., Xi, L.: Evolving boundary detector for anomaly detection. Expert Syst. Appl. **38**, 2412–2420 (2011)
9. Powers, S., He, J.: A hybrid artificial immune system and self organizing map for network intrusion detection. Inf. Sci. **78**, 3024–3042 (2008)
10. Li, G., Guo, T.: Receptor editing-inspired real negative selection algorithm. Comput. Sci. **39**, 246–251 (2012)
11. Laurentys, C., Ronacher, G., Palhares, R., Caminhas, W.: Design of an artificial immune system for fault detection: a negative selection approach. Expert Syst. Appl. **37**, 5507–5513 (2010)
12. Fanelli, R.: A hybrid model for immune inspired network intrusion detection. In: International Conference on Artificial Immune Systems, pp. 107–118. Springer (2008)
13. Coello, C., Greensmith, J., Krasnogor, N., Li, P., Nicosia, G., Pavone, M.: A negative selection approach to intrusion detection. In: Artificial Immune Systems. Lecture Notes in Computer Science, vol. 7597, pp. 178–190 (2012)
14. Farmer, J., Packard, N., Perelson, A.: The immune system, adaptation and machine-learning. Phys. D **22**, 187–204 (1986)
15. Saurabh, P., Verma, B.: A novel immunity inspired approach for anomaly detection. Int. J. Comput. Appl. **94**(15), 14–19 (2014)

16. Abdolahnezhad, M., Banirostam, T.: Improved negative selection algorithm for email spam detection application. Int. J. Adv. Res. Electron. Commun. Eng. **5**, 956–960 (2016)
17. Delona, C.J., Haripriya, P.V., Anju, J.S.: Negative selection algorithm: a survey. Int. J. Sci. Eng. Technol. Res. **6**(4), 711–715 (2017)
18. Vu Thanh, N., Toan Tan, N., Khang Trong, M., Tuan Dinh, L.: A combination of negative selection algorithm and artificial immune network for virus detection. In: International Conference on Future Data and Security Engineering. FDSE 2014: Future Data and Security Engineering, pp. 97–106 (2014)
19. Yang, T., Chen, W., Liu, Z., Lin, P.: A real value negative selection algorithm based on antibody evolution for anomaly detection. In: 2018 Tenth International Conference on Advanced Computational Intelligence (ICACI), Xiamen, China, pp. 692–699 (2018)
20. Wierzchoń, S.: Generating optimal repertoire of antibody strings in an artificial immune system. In: Intelligent Information Systems, pp. 119–133 (2000)
21. Wawryn, K., Widuliński, P.: A human immunity inspired algorithm to detect infections in a computer program. In: Proceedings of the 26th International Conference Mixed Design of Integrated Circuits and Systems, pp. 381–385 (2019)
22. Helman, P., Forrest, S.: An efficient algorithm for generating random antibody strings. Technical Report CS-94-07. The University of New Mexico (1994)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.