arXiv:2403.19489v2 [cs.NE] 10 Jun 2024

Evolving Assembly Code in an Adversarial Environment

Irina Maliukov irinamal@post.bgu.ac.il
Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel

Gera Weiss geraw@bgu.ac.il
Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel

Oded Margalit odedm@post.bgu.ac.il
Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel

Achiya Elyasaf achiya@bgu.ac.il
Department of Software and Information System Engineering, Ben-Gurion University of the Negev, Be'er Sheva, Israel

Abstract

In this work, we evolve Assembly code for the CodeGuru competition. The goal is to create a survivor—an Assembly program that runs the longest in shared memory, by resisting attacks from adversary survivors and finding their weaknesses. For evolving top-notch solvers, we specify a *Backus Normal Form* (BNF) for the Assembly language and synthesize the code from scratch using *Genetic Programming* (GP). We evaluate the survivors by running CodeGuru games against human-written winning survivors. Our evolved programs found weaknesses in the programs they were trained against and utilized them. To push evolution further, we implemented memetic operators that utilize machine learning to explore the solution space effectively. This work has important applications for cyber-security as we utilize evolution to detect weaknesses in survivors. The Assembly BNF is domain-independent; thus, by modifying the fitness function, it can detect code weaknesses and help fix them. Finally, the CodeGuru competition offers a novel platform for analyzing GP and code evolution in adversarial environments. To support further research in this direction, we provide a thorough qualitative analysis of the evolved survivors and the weaknesses found.

Keywords

Genetic Programming, Assembly, Code Generation, Cyber-Security, CodeGuru Xtreme

1 Introduction

CodeGuru Xtreme Leshem and Eyzenberg (2012) is a coding competition where short 8086 Assembly programs, called survivors, are loaded into a random address in a virtual computer memory arena. Their goal is to defeat all other survivors by staying the last program to run. An opponent is defeated when it runs an illegal command caused, e.g., by overwriting its memory. A screen-shot of the game is depicted in Figure 1. Each survivor gets a different color in the arena, representing the bytes it wrote to the shared memory. We elaborate on the game in Section 3.

©2024 by the Massachusetts Institute of Technology

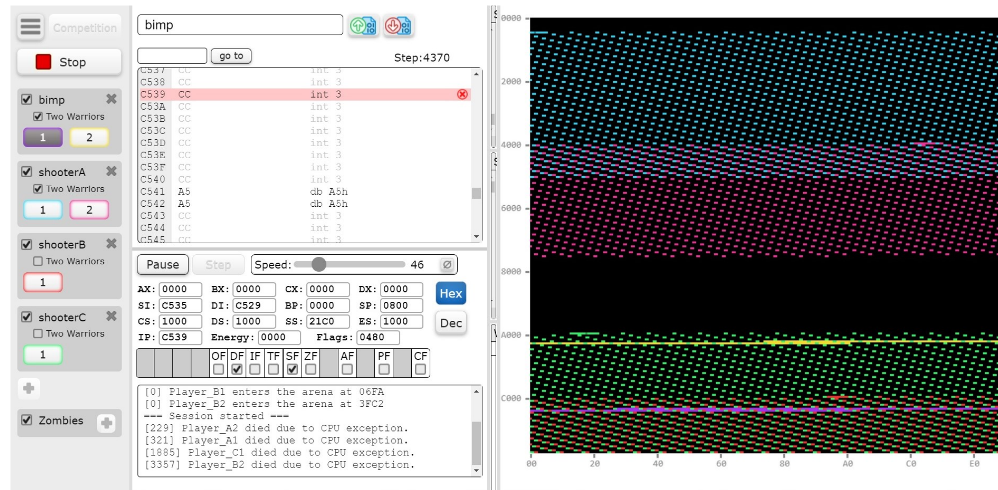


Figure 1: The CodeGuru Xtreme game. On the left are the survivors; on the center is the code of the selected survivor; and on the right is the arena, i.e., the memory status. Each survivor gets a different color in the arena, representing its written bytes.

In this work, we evolve winning survivors from scratch, i.e., from randomly generated Assembly code, and without access to the source code of known survivors. For this task, we utilize *Grammar-Guided Genetic Programming* (G3P)—an evolutionary computation technique that incorporates GP principles, employs context-free grammar and operates directly with tree-based representations. G3P allows us to evolve Assembly programs following grammar-type constraints. The goal of the individuals, embodied by the fitness function, is to overtake adversaries and win the game. The evolved code is represented using an *Abstract Syntax Tree* (AST) based on matching Assembly *Backus Normal Form* (BNF) we defined. A BNF is a meta-syntax notation for context-free grammar consisting of derivation rules. Since our BNF is general and domain-agnostic, the approach applies to generating Assembly programs for other domains and processors.

No previous work has been done on the CodeGuru Xtreme game, except for an undergraduate project Sysman and Leibu (2009) and some early work on the "Core War" game Andersen (2001); Corno et al. (2003), which served as the basis CodeGuru Xtreme (see Section 3). As elaborated in Section 2, some work has been done on the evolution of low-level languages (i.e., Assembly and Java bytecode), and some work has been done to improve existing Assembly code. Note that improving existing code is a simpler task than generating new code from scratch, as the former's state space is much smaller Petke et al. (2018); Banzhaf (2018).

CodeGuru Xtreme competition has been running since 2005, with all past survivors publicly available. Thus, winning is considerably tricky and requires, among other qualities, a good understanding of the 8086 Assembly language.

This work combines a fitness approximation approach based on machine learning (ML) to overcome the expensive fitness evaluation. We utilize this approximation for developing memetic operators—genetic operators that incorporate local search to enhance the exploration, leading to significant improvements in the overall evolutionary computation.

Our work has implications for cyber-security. We utilize evolution to detect and

exploit weaknesses in other survivors. Furthermore, understanding the Assembly language is a necessity for some viruses. By modifying the fitness function, our approach can be used for detecting weaknesses in code and help in fixing them, detecting suspicious adversarial code, or, on the contrary, can be intended to avoid security mechanisms by mutating the virus to avoid detection while keeping its functionality. In Section 6, we use the CodeGuru Xtreme game and GP for mimicking Assembly viruses and demonstrating how our method can be used for avoiding anti-virus detection.

The CodeGuru Xtreme competition provides a unique opportunity to analyze GP and code evolution in adversarial environments. In order to encourage further research in this area, we conducted a comprehensive qualitative analysis of the evolved survivors and identified their strengths and weaknesses. This analysis sheds light on the effectiveness of the evolved code and provides valuable insights for future improvements and advancements in the field.

The contributions of this work are:

- We develop a generic Assembly BNF.
- We demonstrate, for the first time, how evolution can evolve Assembly code in an adversarial environment, completely from scratch.
- We provide a qualitative analysis of the evolved survivors and the weaknesses found to facilitate further research in the field of cyber-security.
- We demonstrate how CodeGuru and our approach can be utilized to mimic and explore virus behavior.

2 Related Work

Low-level code evolution Several works have been done on low-level code evolution, some similar to Assembly, like Verilog (a hardware description language) and Java bytecode.

Karpuzcu (2005) used Grammatical Evolution (GE) for evolving a simple program of a one-bit full adder. In spite of the strong bias they employed, the success achieved was only about 5.7%. Orlov and Sipper (2011) proposed a method for genetic improvement and repair of existing Java programs or any software that can be compiled into Java bytecode. Although Java bytecode resembles Assembly, it has a simplified representation and does not have direct memory access. This contrasts with Assembly, which has a very strong correspondence between its instructions, the architecture's machine code instructions, and memory. Orlov and Sipper seeded the initial population with copies of a single hand-crafted individual and improved it over generations while we evolve Assembly code from scratch. Rosin (2019) synthesized simple programs with loops from input/output examples. They targeted a simplified low-level language similar to Assembly, where each instruction consists of an opcode and a single operand.

Limited Assembly code evolution Some works focused on constrained Assembly evolution—specific routines, predefined input and output tables, and manually written code parts. Serruto and Casas (2017) applied multi-objective linear GP for the automatic generation of specific Assembly driver routines. The evolved programs did not contain jump instructions because they could form infinite loops, in contrast to our wish to include them in the generated code. The results showed that the automatically

generated microcontroller code for specific tasks can compete with a human programmer with a smaller code size or faster execution. We aim to recreate this result within an adversarial environment. Similarly, Ferrel and Alfaro (2020) presented a methodology for writing Arduino programs using an automatic generator of Assembly language routines based on a cooperative co-evolutionary multi-objective linear GP. They decomposed the problem into sub-components, generating about 73% of the program, and the remaining 27%, which are the main program and initial configuration routines, are manually written. In our case, we cannot break the goal of winning an opponent to sub-tasks without reading its code first, which we avoid. In addition, there is no clear way to represent a winning result using an input-output table. We need to evolve from scratch, a winning program, including jumps and loops, completely automatically.

Overview of Program Synthesis Methods Dominik Sobania (2021) surveyed recent developments in program synthesis with evolutionary algorithms and found that the most influential approaches in the field are stack-based GP (usually PushGP), G3P, and Linear GP. PushGP produces code only in the Push language and cannot be used outside of a Push interpreter, while G3P and Linear GP can produce source code in any language, including Assembly.

Pantridge et al. (2017) compared the synthesis capabilities of PushGP and G3P over different programming languages. It was done on problems of two types: the first is usually approached using machine code or Assembly language (basic execution models problems), while the second is usually approached using high-level languages (general program synthesis benchmark suite). In the low-level field, the best-synthesized program was in TerpreT—a probabilistic programming language designed for inductive program synthesis. It was able to solve all the basic execution model problems, in contrast to PushGP, which succeeded in 6 out of 8. Sadly, there were no gathered results for G3P in this field. In the high-level field, G3P was able to solve almost as many problems as PushGP in the software synthesis benchmark suite.

All the related works present useful ideas for various research directions to achieve our goal of Assembly code evolution. Yet, none achieved total success in evolving independent Assembly programs from scratch. They all applied constraints and limitations on the programs, removed language features, or started from an initial given code. We aim to expand the above achievements.

GP usage in cyber-Security O'Reilly et al. (2020) stated that combining GP and competitive co-evolutionary algorithms enables evolving complex behaviors to be used as abstractions by adversaries. The paper presents the RIVALS framework, which serves as a testbed for computational modeling and simulation of the dynamics of networks under attack. Hu et al. (2020) describe the attack-defense relationship using LQRD (Logit Quantal Response Dynamics). They describe how, by analyzing the evolutionary equilibrium, we can obtain the optimal defense strategy and demonstrate it on WannaCry malware.

In addition to the works that use GP to obtain strategies, others utilize it to evolve and modify malware. Noreen et al. (2009) developed a feature representation of the Bagel malware family and, using GA and a few training samples, were able to evolve new unknown variants. Castro et al. (2019) used GP to evade malware detection by automatically finding code optimizations that, when injected into previously detected malware, result in misclassification of the malware scanner. They implemented their framework as a sandbox to track how the code samples behave. Murali and Velayutham (2022) used novelty search in order to generate malware variants of greater

diversity to evade detection. They used Assembly code samples and represented them in linear or graph representations. Generic Assembly code transformation functions are applied as operators: inserting fake instructions forced `jmp`, unreachable blocks, and conditional `jmp`s. They were able to evade over 98% of popular scanners using this technique.

In our work, we demonstrate the ability to evade detection using GP with general Assembly grammar and domain-independent operators. Unlike the described above works, we do not create special operators for code obfuscation. We utilize CodeGuru as an adversarial game framework to evolve Assembly code, which is able to evade an adversary and is specially designed to overtake it based on a characterizing signature. This resembles diverse forms of viruses that avoid malware scanners.

3 CodeGuru Xtreme

CodeGuru Xtreme by Leshem and Eyzenberg (2012) is a coding competition based on "Core War"—a 1984 programming game created by D. G. Jones and A. K. Dewdney Dewdney (1984). In CodeGuru Xtreme, short 8086 Assembly programs (at most 512 bytes long) of 16-bit commands, called survivors, are loaded into random space on a virtual computer memory arena of size 64KB. Each survivor is loaded to a random address with a stack of 2,048 bytes and a full set of registers. The distance between two survivors and the arena's edges is at least 1,024 bytes. The last survivor alive is the winner and gets one point. If several survivors stay alive, the point is divided equally between all. A survivor is disqualified if it runs an illegal command or attempts to access a memory address outside the arena or its stack. A survivor can be forced to run an illegal command due to a writing operation an opponent previously performed on its code or on an address it reads from. Therefore, performing various writings to memory enhances the chance of damaging opponents. Each battle of the game runs for 200,000 rounds or until only one survivor is left, whichever comes first. In every round, the next command of each survivor is executed in a round-robin fashion. The order of the survivor's execution is changed randomly for each game. The memory arena image and scoreboard that monitors the game's progress are depicted in Figure 1. In most cases, each participant has two programs, called parts, that can collaborate together. Each part has its own registers and is loaded into a different place in memory, although they both share a stack. The parts are executed separately, one after the other. Their scores are joined together at the end of the battle and create the survivor's score. This allows the design of a survivor with two parts collaborating together via a shared stack or with two parts completely independently running; both designs are used as a force multiplier for maximizing the survivor's abilities. The game includes the following special commands: `WAITx4` increases the survivor's speed, allowing it to run several opcodes in a single round, `INT 0x86` writes 256 bytes into memory, and `INT 0x87` re-writes a pattern of 4 bytes. The special commands allow performing several rounds of actions in one round.

The CodeGuru competition has taken place every year since 2005 among outstanding high-school students. Each year, the level rises, with more sophisticated survivors written. This work aims to evolve survivors that will win the top survivors of previous years by finding their weaknesses. Our goal is not to win the competition but rather to show that GP can be utilized for evolving code in an adversarial environment. Thus, we evolve a different survivor for each past survivor rather than evolving one survivor that takes them all.

4 Method

To evolve our survivors, we use Grammar-Guided Genetic Programming (G3P)—a technique that incorporates Genetic Programming principles, employs context-free grammar, often in a BNF form, and operates directly with tree-based representations. G3P allows evolving Assembly programs following grammar type constraints and a defined aim, overtaking adversary in this case. During the evolutionary process with G3P, the evolved code is represented using an AST based on matching Assembly BNF representation.

We now elaborate on the different parts of the evolutionary process.

4.1 Representation

Each individual consists of *two* programs, called *parts* (see Section 3), represented by an AST that follows a grammar defined by a BNF. Since Assembly is a symbolic programming language, it can be represented using it. The terminals are opcodes and operands, and the functions are structures in the language (see listings 4 and 5 in the appendix). We define our types and derivation rules based on Assembly language constraints. For example, we define an unary command as a command consisting of an opcode that takes only one operand. Notably, except for a few CodeGuru special operators, our BNF is general and can match any 8086 Assembly code. There are Assembly commands that are not supported by the game's engine and were left out of the grammar to preserve legal programs.

4.2 Fitness Function

We evaluate survivors' fitness by running a CodeGuru game of 200 battles with the selected human-written survivor the evolution performed against. The game's engine is an open-source Java program that outputs the final scores for each game. As previously explained, the score is one point given to the last survivor alive. If several survivors stay alive, the point is divided equally between all. We modified the engine to produce more information about each game, as elaborated by the fitness function that has four parts:

Engine score: the survivor's average engine score in all played games.

$$f_{\text{score}} = \frac{\sum_{i=1}^{\text{games}} \text{score}_i}{\text{games}}$$

Lifetime: the normalized average number of rounds the survivor stayed alive.

$$f_{\text{lifetime}} = 0.1 \log_{10} max\left(1, \frac{\sum_{i=1}^{\text{games}} \text{reached_round}_i}{\text{games}}\right)$$

Written bytes: the normalized average number of new bytes the survivor wrote. That is, the writing was performed on a memory fragment, which was not written before, or that the last one to write in was not the survivor itself.

$$f_{\text{written_bytes}} = 0.1 \log_{10} max\left(1, \frac{\sum_{i=1}^{\text{games}} \text{written_bytes}_i}{\text{games}}\right)$$

Writing rate: the average writing rate of the survivor.

$$f_{\text{writing_rate}} = 0.1 \frac{\sum_{i=1}^{\text{games}} \text{written_bytes}_i}{max\left(1, \sum_{i=1}^{\text{games}} \text{reached_round}_i\right)} \times \frac{1}{\text{games}}$$

The first two parts encourage evolution to win the competitions and survive for longer periods (respectively). The last two parts encourage the evolution of programs that write in different memory places, which enhances the chance of damaging opponents. We refer to the score parameter as the most significant since it reflects the

performance compared to the adversary. Nevertheless, the other parts are important for guiding the evolution towards the different sub-goals and discriminating the individuals. Division by 10 and $\log_{10}$ were used on the original values of lifetime and written_bytes in order to normalize them to an easy-to-process range yet maintain the tendency they represent. We also defined a bloat weight parameter, which equals 10^{-5}. It slightly lowers the fitness of large evolved trees in order to prevent them from bloating and yet allows large but powerful trees to evolve.

The fitness formula which performed the best was:

$$f = 2f_{\text{score}} + 0.2f_{\text{lifetime}} + 0.3f_{\text{written_bytes}} + 0.1f_{\text{writing_rate}} \\ -10^{-5}max(\#part1_nodes, \#part2_nodes)$$

It produces fitness values in the range of $[0, 2.5]$, which does not produce sharp deviations. The chosen weights reflect the above-elaborated goals. The score is doubled due to its significance, resulting in a value in the range of $[0, 2]$. The additional parameters were multiplied by weights, resulting in a mutual sum of 0.5 at most in order not to overshadow the score. According to conducted experiments, the writing_rate parameter resulted in the smallest survivor's improvement, thus receiving the lowest weight. Experiments also showed that evolution was able to independently learn the importance of the survivor's lifetime, on the contrary to the importance of performed writings, which it learned seldom. Thus, high and medium weights were given to written_bytes and lifetime parameters, respectively, to accelerate the improvement process.

4.3 Genetic Operators

We used Koza's standard mutation and crossover operators Koza et al. (1994) that operate on the survivors' parts, which are represented as trees. Specifically, we used the grow sub-tree (i.e., sub-part) mutation and the exchange sub-tree (sub-part) crossover. We added two more operators. The *duplicate-tree* (part) mutation takes the best tree (part) of a survivor and replaces the second part with it. The *exchange-trees* (parts) crossover replaces one of the trees (parts) of the first individual with one of the trees (parts) of the second.

4.4 Improvements

Below are several improvements to the basic setup that we tested.

4.4.1 Random Generator Pattern

We wish to add randomness to the BNF to allow our survivors to be unpredictable. Thus, we add Pseudo-Random Number Generator (PRNG) patterns to our BNF. Specifically, we added Linear Congruential Generator (LCG) and XOR-Shift Generators implementation to our grammar as shown in Listing 6 in the appendix.

4.4.2 Fitness Approximation

One of our main hold-backs is the fitness calculation time. Raising the Java engine, running 200 battles, and outputting the results into a file takes non-negligible time, which prolongs the evolution that itself requires many hours to complete a run due to G3P's stochastic nature. To handle this issue, we combined a machine learning model, as presented in Tzruia et al. (2023), that learns how to approximate the fitness value without evaluating the individuals. Since their approach is designed for vector-based representation, we first create a mapping between our AST representation into a float vector form. For that, we use the AST's size and the engine's parameters (elaborated

in Section 4.2). For example, if an individual has reached a score of 0.3, a lifetime of 3.21827, wrote 0.69897 bytes in a rate of 0.01, and its larger tree contains 25 nodes, its vector representation will be [0.3, 3.21827, 0.69897, 0.01, 25]. During evolution, the individuals' vector representations and their actual fitness value are collected and used for training the ML model. The ML model switches between actual and approximated fitness according to defined conditions. In the actual fitness phase, it performs learning on truly evaluated individuals until it reaches a sufficient level of correct prediction and switches to approximation. In our case, we chose the switch condition to be based on cross-validation (CV) error as it reaches 5% of the maximal possible fitness, which equals 0.125 out of 2.5.

In each generation in the approximation phase, a certain percentage of the population is still evaluated regularly to maintain the model and save the accurate results of the best individual evolved so far. The percent which performed the best was 30%. Less sampling resulted in non-convergence of the evolution, and more sampling increased the evolution time.

4.4.3 Memetic Operators

As we will show, the use of fitness approximation dramatically reduced the computation time. Thus, we decided to utilize the fitness approximation to create smart memetic operators that use local search for further improving evolution.

Whenever we apply the basic genetic operators described above, we run them five times and choose the best individual according to the approximated fitness.

5 Experiments and Results

We carried out a comprehensive set of experiments aimed at winning the top human-written survivors. Our code is written in Python, using the EC-KitY toolkit Sipper et al. (2023). Our code and data are at Assembly code generation. The code for the human-written survivors we compete with can be found at Leshem and Eyzenberg (2012).

Experiments were conducted on a shared cluster of 96 nodes and a total of 5,408 CPUs (the most powerful processors are AMD EPYC 7702P 64-core, although most have lesser specs). 64 CPUs and 150 GB RAM were allocated for each evolutionary run (against one human adversary) to parallelize the evaluation. In practice, each run without the fitness approximation took approximately two days.

The specific hyperparameters utilized in the experiments and their chosen values are detailed in Table 1. The population size was chosen to be 192 to optimize the need for diversity, considering the resources of 64 CPUs and assigned time per evolutionary run. The Grow mutation probability was chosen to be 0.7 as the experiments that were conducted showed a clear tendency to better results with a high mutation rate, yet this rate allows evolution to perform a significant learning process.

The operators were sequentially applied to individuals with different probabilities (Table 1). The evolution was set to terminate when 2K generations were reached, or before, depending on whether convergence between best and average fitness values was achieved in addition to a monotonic non-increasing winning strike of 200 generations.

We repeated each experiment ten times to test consistency. Our individuals' average fitness and standard deviation against each of the past years' winners are in Table 2. We consider an average engine score higher than 0.5 a winning result for our individual. Notably, evolution managed to evolve Assembly programs, which won almost 78% of past years' human-written winners.

Table 1: Evolutionary hyper-parameters.

Representation	Grammar-based GP
Mutation	Grow sub-tree and duplicate tree †
Recombination	Exchange of sub-trees and trees †
Grow mutation probability	0.7
Duplication mutation probability	0.2
Exchange sub-tree recombination probability	0.3
Replacement recombination probability	0.2
Parent selection	Tournament with $k = 4$
Survivor selection	Generational replacement
Population size	192
Termination	2,000 generations or convergence with a winning strike

† The operators are described in Section 4.3.

Table 2: Test average fitness and standard deviation over ten experiments of our best individuals against past years' winners.

Year	**Human survivor**	**#Wins**	**Avg. Engine's Score**	**SD**
2006	Zeus	8/10	0.675	0.162
2007	HutsHuts	10/10	0.960	0.048
2008	APOCALYPSE	9/10	0.741	0.170
2009	XLII	9/10	0.891	0.174
2010	FSM	3/10	0.481	0.147
2011	Mamaliga	9/10	0.738	0.132
2012	Zorg	9/10	0.692	0.171
2013	Snake	10/10	0.736	0.136
2014	IamAA	6/10	0.478	0.220
2014	Paranoia	9/10	0.890	0.190
2015	SilentError	9/10	0.684	0.127
2016	LoudBugFix	2/10	0.402	0.078
2017	Memz	10/10	0.997	0.006
2018	Barvaz'sAngles	10/10	0.991	0.008
2019	Nuki'sDemons	5/10	0.666	0.286
2020	GreeniEs	10/10	0.984	0.020
2021	BlocksOfGuru	10/10	0.753	0.118
2022	TheHeapMen	4/10	0.494	0.102

```
1  @start:
2  and cl, [bx + 0x68 + 0x104 + 0x246]
3  div WORD [bx]
4  l8293849:
5  rcl dl, cl
6  rcl ax, 1
7  rol si, cl
8  push WORD [si]
9  shl dh, cl
10 l8293850:
11 and WORD [di + 0x222], 0x92
12 wait
13 wait
14 mov WORD [di], 0x196
15 jns l8293850
16 @end:
```

(a) Part 1

```
1  @start:
2  sub bh, [bx + 0x30 + @start]
3  div WORD [bx]
4  l8293858:
5  rol dl, cl
6  shr di, 1
7  dw 0x144 inc sp
8  push ds add [0xDA80], bx
9  sbb dl, 0x34 xor al, 0xD3
10 sar ax, cl clc
11 l8293859:
12 and WORD [si + 0x246 + 0x230], 0x264
13 push bx
14 pop WORD [di]
15 and WORD [si + 0x206 + 0x202 + 0x220 + 0
       x-14 + 0x-20 + 0x32], 0x144
16 jmp l8293859
17 @end:
```

(b) Part 2

Listing 1: Evolved survivor against Zorg (2012 winner). The two parts of the evolved survivor utilized Zorg's Achilles' heel by writing data to a part of its program. Strike-through text denotes run-time changed code.

5.1 Qualitative Analysis of Evolved Survivors

In this section, we inspect the code of the evolved solvers. The inspection reveals that the evolved survivors managed to win complex and long survivors using a relatively small code fragment. Although GP frequently evolves long code, sometimes only a small part of it is used in the program run flow and yet manages to win. As we will demonstrate, this shows how evolution found the Achilles' heel in the opponents' code and utilized it for its benefit.

5.1.1 Utilizing Achilles' heel

One of the clearest examples is Zorg—the 2012 winner. Zorg writes an important code fragment for its future run on memory address zero. The evolution process noticed it in about 100 generations and overridden this memory by addressing `di`, which holds the value zero, depriving Zorg of winning (see line 14 in both parts of Listing 1, which includes the `dw` translation to Assembly commands and the effect on the following commands). Zorg's code is significantly longer and more complicated than the evolved fragment that overtook it. The evolved survivor manages to win Zorg in about 70% of the battles in a game (according to the average engine's score) despite the weakness finding due to the randomness in the game's execution order.

5.1.2 Concentrated vs. Scattered Memory Writes

During evolution, we noticed several spikes in the best fitness. For example, when training against BlocksOfGuru, there were spikes in the fitness of the best individuals in generations 206 and 256 (see Figure 2). To analyze these spikes, we ran a game with BlocksOfGuru against these individuals and the overall best individual (from generation 1,769). The results and memory image are depicted in Figure 3. We can see that most memory writes were made by the second part of the 1,769 and 256 individuals that cover the arena with scattered green and pink dots. 206's second part performed less, yet a significant number of writes in yellow are concentrated in several areas. All of the first part performed little to no new memory writes. Inspecting their cleared

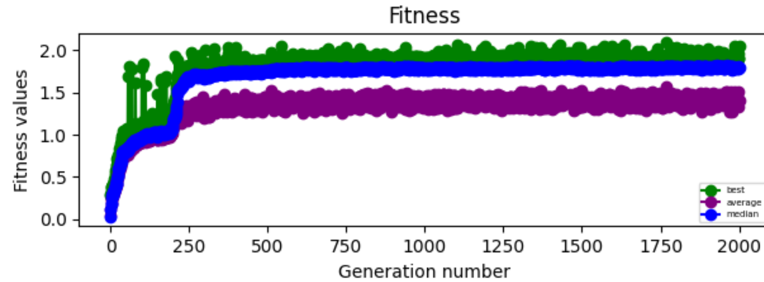


Figure 2: BlocksOfGuru (2021)

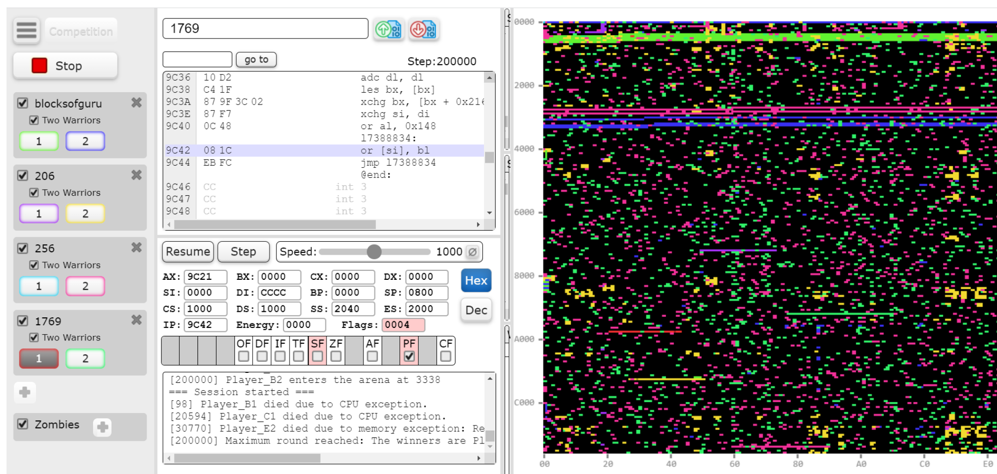


Figure 3: The memory image of BlocksOfGuru vs. best individuals from generations 206, 256, and 1,769. The scattered green and pink dots are memory bytes written by the 1,769 and 256 individuals, respectively. The concentrated yellow dots are memory bytes written by the 206 individual.

runnable code (Listings 2 and 3) reveals that the first parts of 256 and 1,769 run in the loop written in their bottom, which keeps the individual alive but does not perform attacking actions—as seen in the lack of their color in the arena. In 206, both parts run in a loop due to `jmp ax` at the end, which jumps into the beginning of the code. Most memory writes of all individuals are performed using addressing `si`, yet with adding different constants to `si`. 256 and 1,769 use special constants like 65,535 and @start, while 206 does not. The first two exceed the bounds of word data, and the computation is thus written to an address defined as the special constant modulo 2^{16}, which results in scattered writes. The evolutionary process discovered that scattered writing has more chances to encounter adversary code, as reflected in their higher scores, and it is reflected in runs against other survivors as well.

5.1.3 Vertical vs. Horizontal Memory Writes

Another interesting pattern we detected in evolved survivors was writing vertical bytes into memory. That is, sequentially writing a byte every 256 bytes, creating a vertical line in the memory state. This contrasts with human-written survivors, who usually write

```
1  @start:
2  not WORD [bx+65535]
3  cmc
4  and [si+0x252],di
5  xchg bx,[si+0x051E]
6  jnc l268089
7  l268089:
8  add [si+0x0802
        +65535],bx
9  sbb cl,0x-14
10 loop l268090
11 l268090:
12 mov [si+@start+0
        x03B4],cx
13 rcr si,1
14 and [si],ax
15 jmp ax
16 @end:
```

(a) 206 part 1

```
1  @start:
2  inc WORD [bx+65535]
3  cmc
4  and [si+0x252],cx
5  xchg bx,[si+0x051E]
6  jnc l268094
7  l268094:
8  add [si+0x0814
        +65535],bx
9  sbb cl,0x124
10 loop l268095
11 l268095:
12 mov [si+0x152],cx
13 rcr si,1
14 and [si],ax
15 jmp ax
16 @end:
```

(b) 206 part 2

```
1  @start:
2  dec WORD [bx
        +65535]
3  cmc
4  and [si+0x252],dx
5  xchg bx,[di+0
        x05FC]
6  l376145:
7  sub [si],cl
8  jmp l376145
9  @end:
```

(c) 256 part 1

```
1  @start:
2  inc WORD [bx+0x260
        +65535]
3  cmc
4  and [si+0x252],di
5  xchg bx,[si+0x0802
        +65535]
6  jnc l376150
7  l376150:
8  add [si+0x0802
        +65535],bx
9  sbb cl,0x-14
10 loop l376151
11 l376151:
12 mov [si+@start+0x03B4
        ],cx
13 rcr si,1
14 and [si],ax
15 jmp ax
16 @end:
```

(d) 256 part 2

Listing 2: Comparing the code of best individuals against the BlocksOfGuru survivor.

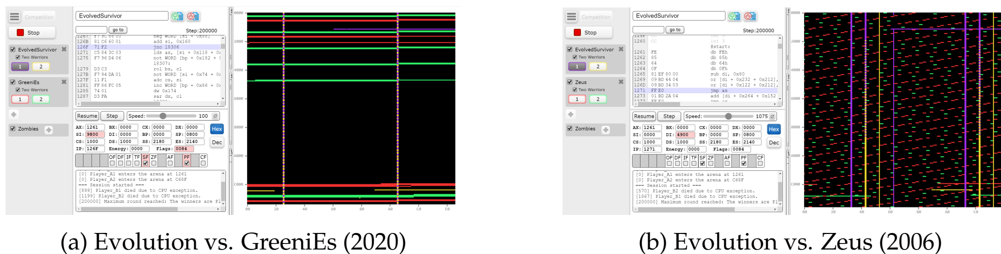
```
1  @start:
2  xchg bp,[bp+0x0130]
3  and [di+0x072C],dx
4  xchg di,[di+0x05FC]
5  l7388834:
6  or [si],bl
7  jmp l7388834
8  @end:
```

(a) 1,769 part 1

```
1  @start:
2  inc WORD [bx+0x260+65535]
3  cli
4  and [si+0x252],di
5  xchg bx,[si+0x0802+65535]
6  jnc l7388849
7  l7388849:
8  add [si+0x0802+65535],bx
9  sbb cl,0x-14
10 loop l7388850
11 l7388850:
12 mov [si+@start+0x03B4],cx
13 rcr si,1
14 and [si],ax
15 jmp ax
16 @end:
```

(b) 1,769 part 2

Listing 3: Comparing the code of best individuals against the BlocksOfGuru survivor.



(a) Evolution vs. GreeniEs (2020)   (b) Evolution vs. Zeus (2006)

Figure 4: Vertical vs. horizontal memory write. Horizontal writings of evolved survivors (in purple and yellow) “cut” the human-written adversaries that write in horizontal lines (green and red) by writing on their code before they reach their code.

Year	Human Survivor	w/o Random		w/ Random	
		Avg. Score	#Victories	Avg. Score	#Victories
2010	FSM	0.481	3/10	0.483	5/10
2014	IamAA	0.478	6/10	0.377	3/10
2016	LoudBugFix	0.402	2/10	0.483	6/10
2022	TheHeapMen	0.494	4/10	0.496	5/10

Table 3: Test game results with and without randomness.

memory in horizontal lines (consecutive bytes). As a result, the evolved survivors were able to “cut” the adversary by writing on its code before the adversary reached its code. This is depicted, for example, in the memory state of our evolved individuals against Zeus (2006) and GreeniEs (2020) (see Figure 4). The evolved survivors, in purple and yellow, write their code in vertical lines, thus cutting their adversaries’ horizontal writing (in red and green). Writing vertically assures reaching the opponent’s code faster because most submitted survivors take advantage of the maximum allowed code size, thus filling at least one memory row entirely. Therefore, filling one or a few columns will be faster than filling complete rows. A similar pattern was found in many runs against other survivors.

The presented patterns of utilizing weak points, scattered and vertical writings are expressed in a significant part of the performed evolutionary runs against all the survivors, although not all the runs of each survivor have used the same pattern.

5.2 Random Generator Pattern

The original program had difficulties overtaking a few previous years’ winners, specifically FSM 2010, IamAA 2014, LoudBugFix 2016, and TheHeapMen 2022, resulting in an average score lower than 0.5. We assumed that the BNF extension of random generator patterns may improve our evolved survivors. We ran the evolution against the above adversaries again for ten runs each.

As Table 3 shows, using randomness improved the number of games evolution won and the average score in three out of four cases. The majority of the best-evolved individuals contained at least one of the random patterns. However, in some, the pattern appears in an unreachable code segment or outside the loop, meaning it only executes once. We believe it helped the evolution process, even though the winning survivor does not actively use it.

The use of randomness enhanced the use of scattered writing patterns for some of the survivors and evolved a combined horizontal-vertical writing pattern for others, in contrast to the vertical-only pattern. The random pattern allowed for the combination of the described patterns together, resulting in scattered writing in horizontal lines that expand vertically, as seen in Figure 5. We can see the yellow and purple memory cells that are being filled horizontally at the beginning. Afterward, the created lines expand vertically, and everything is done using scattered writing.

5.3 Fitness Approximation and Memetic Operators

For the ML-based fitness approximation approach (Section 4.4.2), we used the Ridge model over Lasso (see Tzruia et al. (2023)) as it helps reduce overfitting that results from model complexity and doesn’t set the value of the coefficient to absolute zero. The complete fitness-approximation hyperparameters are given in Table 4, and they

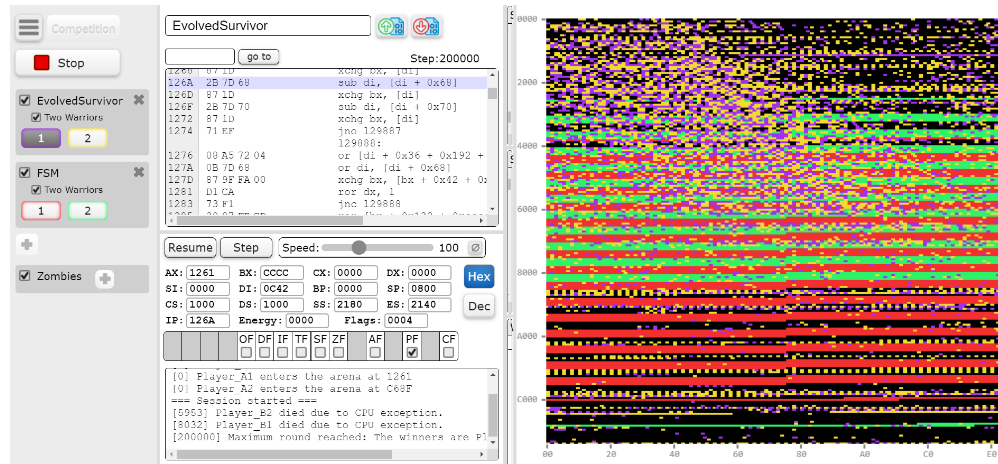


Figure 5: Scattered, horizontal, and vertical writing in the survivor, which evolved using randomness patterns against FSM (2010).

Table 4: ML hyper-parameters.

switch_condition	CV Error
switch_threshold	0.125
sample_rate	30%
model	Ridge
alpha	0.3
gen_weight	sqrt

are used with the parameters described in Table 1.

The combination of the approximation model in our evolution led to about a 30% decrease in total evolution time while preserving the winning results achieved before (see Table 2). For example, the evolution vs Mamaliga (2011) previously took about 77 hours and was decreased to 56 hours with the same amount of generation until winning was achieved.

As the approximated fitness was relatively accurate, we decided to utilize it against the human-written survivors we were not able to overtake Table 3. We did that using the memetic operators described in Section 4.4.3, running 10 evolutionary processes against each opponent to establish our results, similar to the method described in Section 4.2. We also used the random-pattern extension in these experiments.

As Table 5 shows, there was a great improvement, and a win was achieved against all human-written survivors we were not able to win before. Furthermore, in all games in the test phase that resulted in winning, they achieved the best possible score of 1.0. They did not win all games, hence the average score of 0.8 and 0.9.

During the experiment, the ML model was able to learn the parameter weights in each run. The highest weight was given consistently to the score parameter, as expected, and the size parameter was approximately ignored in all cases. We also noticed that the writing_rate weight varied among the survivors, while the other weights had close values.

Table 5: Test game results with the memetic operators and random patterns.

Year	Human Survivor	w/o memetic operators		w/ memetic operators	
		Avg. Score	#Victories	Avg. Score	#Victories
2010	FSM	0.481	3/10	0.8	8/10
2014	IamAA	0.478	6/10	0.9	9/10
2016	LoudBugFix	0.402	2/10	0.9	9/10
2022	TheHeapMen	0.494	4/10	1	10/10

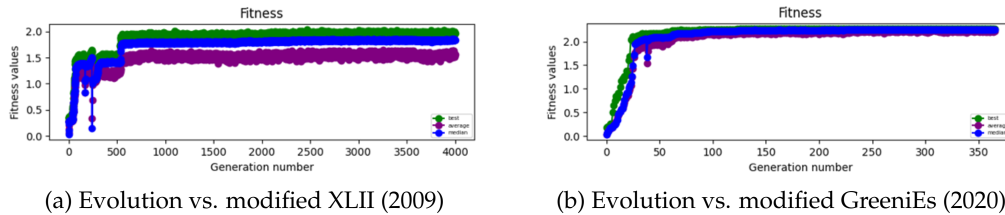
6 Application to Cyber Security

Signatures are widely used in anti-virus programs as a technique to detect known malware. They work by comparing specific patterns or characteristics of files against a database of known malware signatures. If a match is found, the anti-virus program can take appropriate action to quarantine or remove the malicious file. To bypass those techniques, malware developers have devised methods to alter their code in a manner that avoids detection by the signatures while still maintaining the malware's functionality. This perpetuates the ongoing battle between protectors and attackers in an endless cycle.

The adversarial environment of CodeGuru is an interesting platform for analyzing the ability of evolution to overcome tools designed especially against it, like the ability of malware developers to overcome signatures. As mentioned in Section 3, there is a special opcode `INT 0x87` in the game. It looks for a 4-byte sequence identical to the values stored in the registers `AX:DX` and replaces them with the values stored in `BX:CX`. The search is performed in the memory starting from address `DI:ES` moving up or down, determined by the direction flag. Therefore, if a command from a survivor is stored in `AX:DX`, it can be found and replaced by the adversary to an illegal command stored in BX:CX, causing the survivor to run it and be disqualified. This simulates the identification of malware by a signature.

We conducted an experiment utilizing the `INT 0x87` command to simulate signature-based anti-virus. The adversaries chosen were XLII (2009) and GreeniEs (2020), both of which we had previously defeated and that incorporate `INT 0x87` in their code. We subjected them to a complete evolutionary process, with a minor adjustment of halting evolution once the evolved population achieved a consistent win, indicated by an average fitness of 1.1, signifying a winning score (refer to Section 4.2). During this halt, we manually altered the adversaries' `AX:DX` values to match the commands relied upon by the best-evolved individuals. Subsequently, we resumed the evolution, introducing the evolved population to a bespoke adversary.

In both cases shown in Figure 6, the fitness initially dropped in the following generation due to encountering an improved adversary. However, as evolution progressed, it managed to recover and even improve the fitness results to levels achieved before. The evolutionary process successfully replaced the targeted commands with different ones that exhibited similar behavior. These results demonstrate the capability of evolution to confront tools specifically designed to counter it. Furthermore, the use of a domain-independent Assembly grammar, coupled with the prevalence of malware written in Assembly, suggests that these findings could be leveraged to modify malware for evading signature-based anti-virus systems.



(a) Evolution vs. modified XLII (2009)

(b) Evolution vs. modified GreeniEs (2020)

Figure 6: The fitness dropped in generation 241 and 38, respectively, after the tailor-made adversary was created.

7 Conclusion

This work focused on evolving Assembly code for the CodeGuru competition with the objective of creating a survivor program that can run the longest in shared memory while withstanding attacks from adversary survivors. By defining a Backus Normal Form (BNF) for the Assembly language and employing Genetic Programming (GP) to synthesize the code, the study aimed to develop top-notch solvers. The evaluation of the evolved programs involved running CodeGuru games against human-written winning survivors, leading to the identification and exploitation of weaknesses in the opponent programs.

To enhance the evolutionary process, we presented three enhancements. Specifically, we added random patterns to the BNF to allow our survivors to be less predictable; we dramatically cut the overall training time approximating the fitness using a machine-learning (ML) approach; and finally, by developing ML-based memetic operators that allowed our survivors to outperform all pervious-years human-written survivors.

The significance of this research extends to the realm of cyber-security, where the evolved Assembly programs were adept at detecting vulnerabilities in the opponent survivors, showcasing the potential for utilizing evolutionary algorithms to identify and rectify code weaknesses. Our domain-independence Assembly BNF opens up possibilities for adapting the approach to various contexts by adjusting the fitness function to target specific code vulnerabilities.

Moreover, the CodeGuru competition serves as a valuable platform for studying Genetic Programming and code evolution within adversarial environments. The thorough qualitative analysis conducted on the evolved survivors and the vulnerabilities uncovered contributes to the body of knowledge in this area. By shedding light on the efficacy of evolutionary techniques in enhancing program robustness and resilience against attacks, this research paves the way for further investigations into evolutionary strategies for cybersecurity applications and code optimization in competitive settings.

A Our Grammar

⟨*reg*⟩ ::= ‘ax’, ‘bx’, ‘cx’, ‘dx’, ‘si’, ‘di’, ‘bp’, ‘sp’

⟨*half_reg*⟩ ::= ‘ah’, ‘al’, ‘bh’, ‘bl’, ‘ch’, ‘cl’, ‘dh’, ‘dl’

⟨*addres*⟩ ::= ‘[bx]’, ‘[si]’, ‘[di]’, ‘[bp]’

⟨*pop_reg*⟩ ::= ‘ax’, ‘bx’, ‘cx’, ‘dx’, ‘si’, ‘di’, ‘bp’, ‘WORD [bx]’, ‘WORD [si]’, ‘WORD [di]’, ‘WORD [bp]’, ‘ds’, ‘es’

⟨*push_reg*⟩ ::= ‘ax’, ‘bx’, ‘cx’, ‘dx’, ‘si’, ‘di’, ‘bp’, ‘WORD [bx]’, ‘WORD [si]’, ‘ WORD [di]’, ‘ WORD [bp]’, ‘ds’, ‘es’, ‘cs’, ‘ss’

⟨*const*⟩ ::= [(2*i) for i in range(-10, 133)]’, ‘@start’, ‘@end’, ‘65535’, ‘0xcccc’

⟨*op*⟩ ::= ‘nop’, ‘stosw’, ‘lodsw’, ‘movsw’, ‘cmpsw’, ‘scasw’, ‘pushf’, ‘popf’, ‘lahf’, ‘stosb’, ‘lodsb’, ‘movsb’, ‘cmpsb’, ‘scasb’, ‘xlat’, ‘xlatb’, ‘cwd’, ‘cbw’, ‘cmc’, ‘clc’, ‘stc’, ‘cli’, ‘sti’, ‘cld’, ‘std’

⟨*op_single*⟩ ::= ‘div’, ‘mul’, ‘inc’, ‘dec’, ‘not’, ‘neg’

⟨*op_double*⟩ ::= ‘cmp’, ‘mov’, ‘add’, ‘sub’, ‘and’, ‘or’, ‘xor’, ‘adc’, ‘sbb’, ‘test’

⟨*op_jmp*⟩ ::= ‘jmp’, ‘jcxz’, ‘je’, ‘jne’, ‘jp’, ‘jnp’, ‘jo’, ‘jno’, ‘jc’, ‘jnc’, ‘ja’, ‘jna’, ‘js’, ‘jns’, ‘jl’, ‘jnl’, ‘jle’, ‘jnle’, ‘loopnz’, ‘loopne’, ‘loopz’, ‘loope’, ‘loop’

⟨*op_rep*⟩ ::= ‘rep’, ‘repe’, ‘repz’, ‘repne’, ‘repnz’

⟨*op_function*⟩ ::= ‘call’, ‘call near’, ‘call far’

⟨*op_special*⟩ ::= ‘wait wait wait wait’, ‘wait wait’, ‘int 0x86’, ‘int 0x87’

⟨*op_pointer*⟩ ::= ‘lea’, ‘les’, ‘lds’

⟨*op_ret*⟩ ::= ‘ret’, ‘retn’, ‘retf’, ‘iret’

⟨*op_push*⟩ ::= ‘push’

⟨*op_pop*⟩ ::= ‘pop’

⟨*op_double_no_const*⟩ ::= ‘xchg’

⟨*op_shift*⟩ ::= ‘sal’, ‘sar’, ‘shl’, ‘shr’, ‘rol’, ‘ror’, ‘rcl’, ‘rcr’

⟨*section*⟩ ::= ‘ ’

Listing 4: Terminals definitions

⟨*section*⟩ ::= ⟨*label*⟩ ⟨*section*⟩ ⟨*backwards_jump*⟩ ⟨*section*⟩
| ⟨*label*⟩ ⟨*section*⟩ ⟨*backwards_jump*⟩
| ⟨*section*⟩ ⟨*forward_jmp*⟩⟨*section*⟩ ⟨*label*⟩ ⟨*section*⟩
| ⟨*label*⟩ ⟨*section*⟩ ⟨*call_func*⟩ ⟨*backwards_jump*⟩ ⟨*label*⟩ ⟨*section*⟩ ⟨*return*⟩

```
  | ⟨op_double⟩ ⟨reg⟩ ⟨reg — const — address⟩ ⟨section⟩
  | ⟨op_double⟩ ⟨address⟩ ⟨reg — half_reg⟩ ⟨section⟩
  | ⟨op_double⟩ ⟨half_reg⟩ ⟨half_reg — const — address⟩ ⟨section⟩
  | ⟨op_double⟩ ⟨WORD — BYTE⟩ ⟨address⟩ ⟨const⟩ ⟨section⟩
  | ⟨op_pointer⟩ ⟨reg⟩ ⟨address⟩ ⟨section⟩
  | ⟨op_double_no_const⟩ ⟨reg⟩ ⟨reg — address⟩ ⟨section⟩
  | ⟨op_double_no_const⟩ ⟨half_reg⟩ ⟨half_reg — address⟩ ⟨section⟩
  | ⟨op_single⟩ ⟨reg — half_reg⟩ ⟨section⟩
  | ⟨op_single⟩ ⟨WORD — BYTE⟩ ⟨address⟩ ⟨section⟩
  | ⟨op_function⟩ ⟨address⟩ ⟨section⟩
  | ⟨op — op_special⟩ ⟨section⟩
  | ⟨op_rep⟩ ⟨op⟩ ⟨section⟩
  | ⟨op_push⟩ ⟨push_reg⟩ ⟨section⟩
  | ⟨op_pop⟩ ⟨pop_reg⟩ ⟨section⟩
  | 'jmp' ⟨reg — address⟩ ⟨section⟩
  | 'dw 0x'⟨const⟩ ⟨section⟩
  | ⟨op_shift⟩ ⟨reg — half_reg⟩ ⟨cl — 1⟩ ⟨section⟩

⟨call_func⟩ ::= 'call l'⟨const⟩ ⟨section⟩

⟨return⟩ ::= ⟨op_ret⟩ ⟨section⟩

⟨label⟩ ::= 'l'⟨const⟩ ⟨section⟩

⟨forward_jump⟩ ::= ⟨op_jmp⟩ 'l'⟨const⟩ ⟨section⟩

⟨backwards_jump⟩ ::= ⟨op_jmp⟩ 'l'⟨const⟩'-1' ⟨section⟩

⟨address⟩ ::= [⟨address⟩ + ⟨const⟩]
```

Listing 5: Functions definitions

```
⟨section⟩  ::= mov ax, timestamp
               mov ⟨reg⟩, 1664525
               mul ⟨reg⟩
               add ax, 1013904223 ⟨section⟩

⟨section⟩  ::= mov ⟨reg⟩, randint(0, 65,535)
               mov ⟨reg⟩, randint(0, 65,535)
               xor ⟨reg⟩, ⟨reg⟩
               shl ⟨reg⟩, 7
               shr ⟨reg⟩, 5
               xor ⟨reg⟩, ⟨reg⟩ ⟨section⟩
```

Listing 6: Functions definitions for the random patterns.

References

Andersen, D. G. (2001). The garden: Evolving warriors in core wars. https://api.semanticscholar.org/CorpusID:17099745.

Banzhaf, W. (2018). Some Remarks on Code Evolution with Genetic Programming. In Inspired by Nature, pages 145–156. Springer.

Castro, R. L., Schmitt, C., and Dreo, G. (2019). Aimed: Evolving malware with genetic programming to evade detection. In 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), pages 240–247.

Corno, F., Sánchez, E., and Squillero, G. (2003). Exploiting co-evolution and a modified island model to climb the core war hill. In CEC'03., volume 3, pages 2217–2221, Torino, Italy. IEEE.

Dewdney, A. K. (1984). Recreational mathematics–core wars. Scientific American, 2:43–98.

Dominik Sobania, Dirk Schweim, F. R. (2021). Recent developments in program synthesis with evolutionary algorithms. https://arxiv.org/pdf/2108.12227.pdf.

Ferrel, W. and Alfaro, L. (2020). Genetic programming-based code generation for arduino. International Journal of Advanced Computer Science and Applications, 11.

Hu, H., Liu, Y., Chen, C., Zhang, H., and Liu, Y. (2020). Optimal decision making approach for cyber security defense using evolutionary game. IEEE Transactions on Network and Service Management, 17(3):1683–1700.

Karpuzcu, U. R. (2005). Automatic verilog code generation through grammatical evolution. In Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation, GECCO '05, pages 394—397, New York, NY, USA. Association for Computing Machinery.

Koza, J. R. et al. (1994). Genetic programming II, volume 17. MIT press Cambridge, San Mateo, CA.

Leshem, D. and Eyzenberg, T. (2012). Codeguru repository. https://github.com/codeguru-il.

Murali, R. and Velayutham, C. S. (2022). Adapting novelty towards generating antigens for antivirus systems. In Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '22, page 1254–1262, New York, NY, USA. Association for Computing Machinery.

Noreen, S., Murtaza, S., Shafiq, M. Z., and Farooq, M. (2009). Evolvable malware. In Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO '09, page 1569–1576, New York, NY, USA. Association for Computing Machinery.

Orlov, M. and Sipper, M. (2011). Flight of the finch through the java wilderness. IEEE Transactions on Evolutionary Computation, 15(2):166–182.

O'Reilly, U.-M., Toutouh, J., Pertierra, M., Sanchez, D., Garcia, D., Luogo, A., Kelly, J., and Hemberg, E. (2020). Adversarial genetic programming for cyber security: a rising application domain where gp matters. Genetic Programming and Evolvable Machines, 21.

Pantridge, E., Helmuth, T., McPhee, N. F., and Spector, L. (2017). On the difficulty of benchmarking inductive program synthesis methods. In Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17, pages 1589—-1596, New York, NY, USA. Association for Computing Machinery.

Petke, J., Haraldsson, S. O., Harman, M., Langdon, W. B., White, D. R., and Woodward, J. R. (2018). Genetic Improvement of Software: A Comprehensive Survey. IEEE Transactions on Evolutionary Computation, 22(3):415–432.

Rosin, C. D. (2019). Stepping stones to inductive synthesis of low-level looping programs. In Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence and Thirty-First Innovative Applications of Artificial Intelligence Conference and Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, AAAI'19/IAAI'19/EAAI'19, Honolulu, Hawaii, USA. AAAI Press.

Serruto, W. F. and Casas, L. A. (2017). Automatic code generation for microcontroller-based system using multi-objective linear genetic programming. In 2017 International Conference on Computational Science and Computational Intelligence (CSCI), pages 279–285, Las Vegas, NV, USA. IEEE.

Sipper, M., Halperin, T., Tzruia, I., and Elyasaf, A. (2023). EC-KitY: Evolutionary computation tool kit in Python with seamless machine learning integration. SoftwareX, 22:101381.

Sysman, D. and Leibu, A. (2009). Darwin8086. https://code.google.com/archive/p/darwin8086/.

Tzruia, I., Halperin, T., Sipper, M., and Elyasaf, A. (2023). Fitness approximation through machine learning. https://doi.org/10.48550/arXiv.2309.03318.