arXiv:2403.19489v1 [cs.NE] 28 Mar 2024

Evolving Assembly Code in an Adversarial Environment

Irina Maliukov
Gera Weiss
Oded Margalit
irinamal@post.bgu.ac.il
geraw@bgu.ac.il
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 competition's 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. In addition, we compare our approach with a Large-Language Model, demonstrating that the latter cannot generate a survivor that can win at any competition. 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.

CCS CONCEPTS

• **Software and its engineering** → **Assembly languages**; **Genetic programming**; **Source code generation**; *Search-based software engineering*; • **Security and privacy** → *Vulnerability management*; • **Computing methodologies** → **Genetic programming**.

KEYWORDS

genetic programming, assembly, code generation, cyber-security, CodeGuru Xtreme

ACM Reference Format:
Irina Maliukov, Gera Weiss, Oded Margalit, and Achiya Elyasaf. 2024. Evolving Assembly Code in an Adversarial Environment. In *Proceedings of The Genetic and Evolutionary Computation Conference 2024 (GECCO '24)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '24, July 14–18, 2024, Melbourne, Australia
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

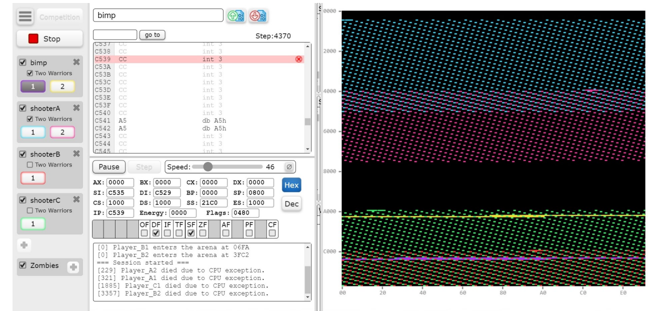


Figure 1: The CodeGuru Xtreme game. On the left are the survivors; on the center is the code of the selected survivor; and on the right is the arena, i.e., the memory status. Each survivor gets a different color in the arena, representing the bytes it wrote to the shared memory.

1 INTRODUCTION

CodeGuru Xtreme [9] 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. The screen-shot of the game screen 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.

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 BNF we defined, where the nodes and leaves are the BNF's functions and terminals, respectively. Since our

BNF and the operators we used are 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 [18] and some early work on the "Core War" game [1, 3], which served as the basis for 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 utilized large-language models (LLM) 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 [2, 14]. To support that, we compared our approach to using LLM in Section 5.2, demonstrating that the latter fails to create a survivor that wins any competition, even when playing against simple survivors.

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

This work has implications for cyber-security. As demonstrated below, we utilize evolution to detect weaknesses in other survivors. In addition, understanding the assembly language is a necessity in 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. Finally, our approach can evolve high-quality individuals with a small number of examples and without detailed information.

The CodeGuru Xtreme competition provides a unique opportunity to analyze *Genetic Programming* (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.

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 [7] 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 [11] 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 assembly, which has 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 [15] synthesized simple programs with loops from input/output examples. They targeted 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. In [16], multi-objective linear genetic programming is applied for the automatic generation of some specific assembly driver routines. The architecture used was an 8051 microcontroller assembly, which is similar to the 8086 we use. 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, [6] presented a methodology for writing Arduino board programs using an automatic generator of assembly language routines based on a cooperative co-evolutionary multi-objective linear genetic programming algorithm. 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 in our case using an input-output table. We need to evolve from scratch, a winning program, including jumps and loops, completely automatically. The presented method [6] requires more general implementation and adaptations to match our use case, yet prove the ability to achieve it.

Overview of Program Synthesis Methods. Several articles compare recent program synthesis and evolution techniques. [12] compares Flash Fill (Microsoft Excel feature which uses version-space algebras to perform program synthesis from examples on string manipulation tasks), MagicHaskeller (synthesizes functional Haskell programs through an exhaustive search of programs with the correct type signatures. Uses Monte-Carlo algorithm to remove semantically equivalent programs from the search space), TerpreT (a probabilistic programming language that is designed for inductive program synthesis), and two forms of GP—PushGP (evolves programs in a Turing complete, stack-based language called Push) and Grammar Guided Genetic Programming (a GP that uses context-free grammar as a guideline for syntactically correct programs throughout the evolution). The comparison focused on the possibility of producing a program that passes all tests for all training and unseen testing inputs. 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 system that performed the best was TerpreT, while sadly, there were no gathered results for G3P. In the high-level field, both GP approaches outperformed the others.

A survey of recent developments in program synthesis with evolutionary algorithms [5], found that the most influential approaches in the field are stack-based GP (usually PushGP), G3P, and Linear

GP (uses registers for calculations and direct memory access). Encouraged by this review and by the fact that G3P has not been widely tested on low-level languages, we want to use G3P for the evolution of assembly programs.

Google DeepMind created AlphaDev, a deep-reinforcement learning agent trained to improve a sorting algorithm [10]. They trained AlphaDev with up to 16 Tensor Processing Units (TPU) v.3, with a total batch size of 1,024 per TPU core and for 1 million iterations. In practice, across all tasks, training took them, in the worst case, two days to converge. Aside from the high resource demands for representing and training the neural network, their approach assumes a given working program to improve. It is not designed to create programs from scratch.

A day after AlphaDev publication, a user on X (formerly Twitter) tried to use ChatGPT to optimize the same sorting algorithm [13]. As noted before, improving an existing code is simpler than generating one from scratch. Nevertheless, we evaluate this approach in Section 5.2 and test whether it is applicable to our domain.

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.

3 CODEGURU

CodeGuru Xtreme [9] is a coding competition based on "Core War"—a 1984 programming game created by D. G. Jones and A. K. Dewdney [4]. 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. 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 that collaborate together. 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` implements "heavy bombing" by writing 256 bytes into memory, and `INT 0x87` implements "smart bombing" by re-writing a pattern of 4 bytes.

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. The AST follows a grammar defined by a BNF—a meta-syntax notation for context-free grammar consisting of derivation rules. 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 defined our types and derivation rules based on assembly language constraints. For example, we defined 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 Evaluation

We evaluated survivors by running a CodeGuru game of 200 battles against the selected human-written survivor. 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.

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

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. We also defined a bloat weight parameter, which equals 10^{-5}. It slightly lowers the fitness of large evolved trees in order to prevent trees 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.

4.3 Genetic Operators

We used Koza's standard mutation and crossover operators [8] 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.

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 [17]. Our code and data are at github.com/anonymous-submission. The code for the human-written survivors we compete with can be found at [9].

Experiments were conducted on a cluster of 96 nodes and 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 run to parallelize the evaluation. In practice, each run took approximately two days.

The specific hyperparameters utilized in the experiments and their chosen values are detailed in Table 1.

The operators were sequentially applied to individuals with different probabilities (Table 1). The evolution was set to terminate when 2,000 generations are reached, or before, depending on whether convergence between best and average fitness values is achieved in addition to a monotonic non-increasing winning strike of 200 generations.

We repeated each experiment ten times to prove 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.

In the following sections, we inspect the code of the evolved solvers. The inspection revealed 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.

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.

5.1 Qualitative Analysis of Evolved Survivors

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 from 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 cases 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 runnable code (Listings 2 and 3) reveals that

```
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 + 0x-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.

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

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

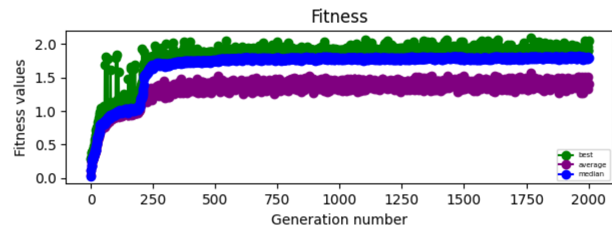


Figure 2: BlocksOfGuru 2021

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 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

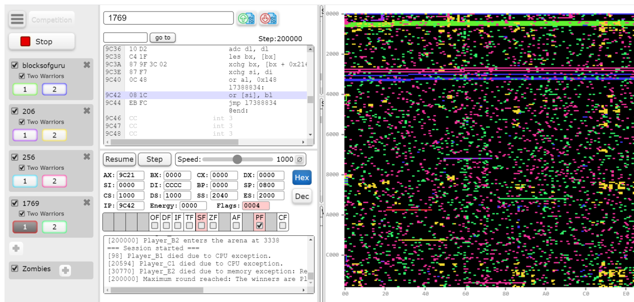


Figure 3: The memory image of BlocksOfGuru vs. best individuals from generations 206, 256, and 1,769. The scattered green and pink dots are memory bytes written by the 1,769 and 256 individuals, respectively. Non-reachable code is stroke through.

a few columns will be faster than filling complete rows. A similar pattern was found in a significant part of 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.

5.1.4 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 human-written survivors may use randomness to be unpredictable, while our BNF has no element of randomness. Thus, we decided to add Pseudo-Random Number Generator (PRNG) patterns to our BNF. We added Linear Congruentional Generator (LCG) and XOR-Shift Generators implementation to our grammar as shown in Listing 6 and ran the evolution against the above adversaries again for ten runs each.

As shown in Table 3, using randomness improved the number of games evolution won and the average score in 3/4 of the 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 (see Section 5.1.2) for some of the survivors and evolved a combined horizontal-vertical writing pattern for others, in contrast to the described in Section 5.1.3. The innovation the random pattern caused is 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. Afterwards, the created lines expand vertically, and everything is done using scattered writing.

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 result with and without randomness.

5.2 Comparison to LLMs

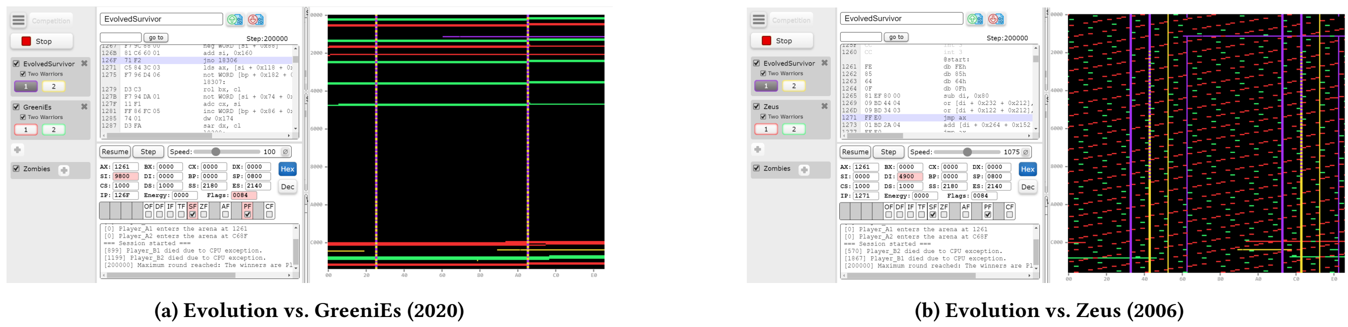
We compared our results to ChatGPT, one of today's leading LLMs. We explained the CodeGuru game, the special opcodes, and the rules using prompt engineering. We asked the chat for a survivor with the best chance to overtake others. The suggested code was tested against simple competition survivors that are given as examples in the game's web engine. At first, the provided survivors did not compile. After supplying corrections, they compiled; however, they achieved poor scores. Even when we supplied the opponents' code to ChatGPT, its produced survivor did not win. The prompts we used can be found at https://chat.openai.com/share/5ebda97c-06f8-4430-bd04-78a5abfc74ea.

6 CONCLUSION

This work presented the evolution of assembly code from scratch in an adversarial environment, specifically the CodeGuru Xtreme competition. Through the use of GP, we synthesized assembly code that outperformed human-written winning survivors from past years. Our evolved programs were able to identify and exploit weaknesses in the programs they were trained against. Additionally, we compared our approach with the use of a Large-Language Model (LLM) and demonstrated that the LLM was unable to generate a survivor capable of winning any competition.

The CodeGuru Xtreme competition serves as a valuable platform for studying GP and code evolution in adversarial environments. To facilitate further research in this direction, we have provided a comprehensive qualitative analysis of the evolved survivors and the weaknesses they have identified. While we were able to overtake most of the top human-written survivors, the domain is far from being solved. There are still some survivors that evolution failed to win, and the evolved survivors were only able to win against one survivor at a time.

The implications of this work extend beyond the CodeGuru Xtreme competition. Utilizing evolution to detect weaknesses in other survivors has important applications in cyber-security. Furthermore, the assembly language used in this research is also relevant in the context of computer viruses. We believe that by modifying the fitness function, our approach can be employed to identify weaknesses in code and aid in their resolution.



(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.

```
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+0x03B4],
       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+0x05FC]
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.

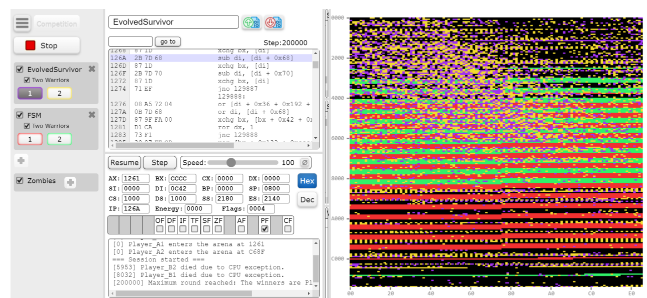


Figure 5: Scattered, horizontal and vertical writing in the survivor which evolved using randomness patterns against FSM (2010).

REFERENCES

[1] David G. Andersen. 2001. The Garden: Evolving Warriors in Core Wars. https://api.semanticscholar.org/CorpusID:17099745.

[2] Wolfgang Banzhaf. 2018. Some Remarks on Code Evolution with Genetic Programming. In Inspired by Nature. Springer, 145–156.

[3] Fulvio Corno, Ernesto Sánchez, and Giovanni Squillero. 2003. Exploiting co-evolution and a modified island model to climb the core war hill. In The 2003 Congress on Evolutionary Computation, 2003. CEC'03., Vol. 3. IEEE, Torino, Italy, 2217–2221.

[4] Alexander Keewatin Dewdney. 1984. Recreational Mathematics–Core Wars. Scientific American 2 (1984), 43–98.

[5] Franz Rothlauf Dominik Sobania, Dirk Schweim. 2021. Recent Developments in Program Synthesis with Evolutionary Algorithms. https://arxiv.org/pdf/2108.12227.pdf.

[6] Wildor Ferrel and Luis Alfaro. 2020. Genetic Programming-Based Code Generation for Arduino. https://thesai.org/Downloads/Volume11No11/Paper_68-Genetic_Programming_Based_Code_Generation.pdf. International Journal of Advanced Computer Science and Applications 11 (01 2020). https://doi.org/10.14569/IJACSA.2020.0111168

[7] Ulya R. Karpuzcu. 2005. Automatic Verilog Code Generation through Grammatical Evolution. https://dl.acm.org/doi/pdf/10.1145/1102256.1102346. In Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation (Washington, D.C.) (GECCO '05). Association for Computing Machinery, New York, NY, USA, 394––397. https://doi.org/10.1145/1102256.1102346

[8] John R Koza et al. 1994. Genetic programming II. Vol. 17. MIT press Cambridge.

[9] Danny Leshem and Tomer Eyzenberg. 2012. CodeGuru repository. https://github.com/codeguru-il.

[10] Daniel Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, and David Silver. 2023. Faster sorting algorithms discovered using deep reinforcement learning. Nature 618 (06 2023), 257–263. https://doi.org/10.1038/s41586-023-06004-9

[11] Michael Orlov and Moshe Sipper. 2011. Flight of the FINCH through the Java wilderness. IEEE Transactions on Evolutionary Computation 15, 2 (2011), 166–182.

[12] Edward Pantridge, Thomas Helmuth, Nicholas Freitag McPhee, and Lee Spector. 2017. On the Difficulty of Benchmarking Inductive Program Synthesis Methods. https://doi.org/10.1145/3067695.3082533. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (Berlin, Germany) (GECCO '17). Association for Computing Machinery, New York, NY, USA, 1589—-1596. https://doi.org/10.1145/3067695.3082533

[13] Dimitris Papailiopoulos. 2023. GPT-4 "discovered" the same sorting algorithm as AlphaDev. https://twitter.com/DimitrisPapail/status/1666843952824168465, last accessed on 2024-01-31.

[14] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. IEEE Transactions on Evolutionary Computation 22, 3 (June 2018), 415–432. https://doi.org/10.1109/tevc.2017.2693219

[15] Christopher D. Rosin. 2019. Stepping Stones to Inductive Synthesis of Low-Level Looping Programs. https://doi.org/10.1609/aaai.v33i01.33012362. 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). AAAI Press, Honolulu, Hawaii, USA, Article 292, 9 pages. https://doi.org/10.1609/aaai.v33i01.33012362

[16] Wildor Ferrel Serruto and Luis Alfaro Casas. 2017. Automatic Code Generation for Microcontroller-Based System Using Multi-objective Linear Genetic Programming. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8560802. In 2017 International Conference on Computational Science and Computational Intelligence (CSCI). IEEE, Las Vegas, NV, USA, 279–285. https://doi.org/10.1109/CSCI.2017.47

[17] Moshe Sipper, Tomer Halperin, Itai Tzruia, and Achiya Elyasaf. 2023. EC-KitY: Evolutionary computation tool kit in Python with seamless machine learning integration. SoftwareX 22 (2023), 101381. https://doi.org/10.1016/j.softx.2023.101381

[18] Dean Sysman and Amir Leibu. 2009. Darwin8086. https://code.google.com/archive/p/darwin8086/.

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.