WOJSKOWA AKADEMIA TECHNICZNA
im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

kpt. mgr inż. Marcin Pachnik

DESTYLACJA KORPUSU DANYCH TESTOWYCH W PROCESIE FUZZINGU Z WYKORZYSTANIEM ALGORYTMU GENETYCZNEGO

ROZPRAWA DOKTORSKA

Promotor: dr hab. inż. Kazimierz Worwa, profesor WAT

Promotor pomocniczy: płk dr inż. Rafał Kasprzyk

Warszawa 2022

Dziękuję wszystkim osobom wspierającym mnie w trakcie opracowywania niniejszej rozprawy. *W szczególności dziękuję mojej żonie –* Klaudynie.

Spis treści

Wykaz symboli i oznaczeń

a_{ij} – zmienna decyzyjna w problemie MCSCP;

A – macierz pokrycia korpusu (algorytm *MoonLight*);

A' – tymczasowa macierz pokrycia korpusu (algorytm *MoonLight*);

bin – bity będące wartością wykładnika funkcji w kodowaniu logarytmicznym;

$[bin]_{10}$ – wartość dziesiętna wykładnika w kodowaniu logarytmicznym;

b – indeks pomocniczy w czasie obliczania funkcji R w selekcji rankingowej;

b_u – binarna zmienna decyzyjna, określająca czy *u*–ty przypadek testowy wchodzi w skład zredukowanego korpusu;

B – liczba wybieranych osobników w selekcji rankingowej;

c_j – koszt *j*-tej podkolekcji w problemie MCSCP;

c_j^t – wektor kosztów *j*-tej podkolekcji w problemie WSCP;

c_u – czas obsługi *u*–tego przypadku testowego przez badany program;

ch, ch_g – chromosom w algorytmie genetycznym;

ch'_y, ch''_y – *y*–te locusy z chromosomów rodzicielskich (krzyżowanie heurystyczne);

ch_y^S – *y*–ty locus chromosomu potomnego (krzyżowanie heurystyczne);

C – czas obsługi wszystkich przypadków testowych składających się na zredukowany korpus;

d – indeks bajtów (entropia uproszczona);

D – liczba występujących w pliku unikalnych bajtów (grup znaków);

e_i – *i*-ty element zbioru E w problemach SCP, MSCP, WSCP, MCSCP;

E – zbiór elementów w problemach SCP, MSCP, WSCP, MCSCP;

f – jedna z funkcji w badanej bibliotece (algorytm *Minerva*);

F – zbiór unikalnych indeksów przypadków testowych wzbudzanych przez zredukowany korpus;

g – indeks osobnika w algorytmach ewolucyjnych (w tym algorytmach *VEGA*, epi*VEGA*);

G – rozmiar populacji w algorytmach ewolucyjnych (w tym algorytmach *VEGA*, epi*VEGA*);

h – indeks pomocniczy przy normalizacji funkcji R w selekcji rankingowej;

H – schemat w chromosomach algorytmu genetycznego;

i – indeks elementów zbioru E w problemach SCP, MSCP, WSCP, MCSCP;

j – indeks podzbiorów kolekcji K w problemach SCP, MSCP, WSCP, MCSCP;

k – indeks kryteriów w algorytmach *VEGA*, epi*VEGA*;

k^* – egzotyczna kolumna w macierzy pokrycia korpusu (algorytm *MoonLight*);

K – skończona kolekcja podzbiorów K_j w problemach SCP, MSCP, WSCP, MCSCP;

$\overline{K}$ – podkolekcja w problemie MCSCP odpowiadająca rozwiązaniu optymalnemu w sensie Pareto;

K_j – j-ty podzbiór w problemach SCP, MSCP, WSCP, MCSCP;

K^* – podkolekcja kolekcji K, której suma wynosi E w problemach SCP, MSCP, WSCP, MCSCP;

l – indeks chromosomów poddawanych operacji epigenetycznej;

L – liczba wszystkich krawędzi przejść w grafie przepływu sterowania badanego oprogramowania;

m – liczebność zbioru E w problemach SCP, MSCP, WSCP, MCSCP;

m_κ – mediana głównej cechy w κ-tej populacji (algorytm epiVEGA);

$m(H,t)$ – średnia liczba osobników pasujących do schematu H w chwili t;

M – zbiór argumentów i ich typów wykorzystywanych przez funkcję badanej biblioteki (algorytm *Minerva*);

n – liczebność kolekcji K w problemach SCP, MSCP, WSCP, MCSCP;

o_k – k-ta funkcja oceny (przystosowania) algorytmu ewolucyjnego;

O – zbiór indeksów zebranych przypadków testowych (początkowy korpus);

p_a – prawdopodobieństwo mutacji pojedynczego allelu;

p_c – prawdopodobieństwo krzyżowania pojedynczego osobnika;

p_c – prawdopodobieństwo krzyżowania pojedynczego osobnika;

p_I – prawdopodobieństwo dziedziczenia genu w krzyżowaniu równomiernym;

p_m – prawdopodobieństwo mutacji pojedynczego osobnika;

$p(H)$ – prawdopodobieństwo, iż schemat H pozostanie nienaruszony w wyniku mutacji i krzyżowania;

$p(d)$ – prawdopodobieństwo wylosowania d–tego bajtu;

P – moc rodziny podzbiorów indeksów krawędzi wzbudzanych przez zredukowany korpus;

P_κ – populacja, zbiór osobników wchodzących w skład κ - tego pokolenia;

P'_κ – populacja pomocnicza;

P''_κ – populacja pomocnicza;

$P_\kappa{}'''$ – populacja pomocnicza;

P_y – populacja algorytmu ewolucyjnego;

P_l – populacja pomocnicza (odrzucone osobniki);

P'_l – populacja pomocnicza (odrzucone osobniki poddane operacji epigenetycznej);

q – dokładność obliczeń (liczba miejsc po przecinku) w kodowaniu binarnym;

Q – współczynnik selekcji (selekcja rankingowe);

r^* – wiersz w macierzy pokrycia korpusu (algorytm *MoonLight*);

r_κ – rozstęp głównej cechy w κ-tej populacji (algorytm epiVEGA);

R – funkcja prawdopodobieństwa przejścia selekcji rankingowej przez osobnika;

R'_b – znormalizowana wartość funkcji;

s_u – zbiór indeksów krawędzi wybudzany przez u–ty przypadek testowy;

S_p – presja selekcyjna;

t – chwila czasu w algorytmach ewolucyjnych, rozumiana jako numer pokolenia;

T – maksymalna liczba pokoleń (algorytmy VEGA, epiVEGA);

T_{max} – maksymalna liczba pokoleń bez poprawy (algorytmy VEGA);

u – indeks przypadku testowego (destylacja korpusu jako MCSCP) ;

U – długość badanego ciągu (pliku) w bajtach;

v_R – wartość prawego końca przedziału liczbowego w kodowaniu binarnym;

v_L – wartość lewego końca przedziału liczbowego w kodowaniu binarnym;

v_B – wartość zakodowanej liczby rzeczywistej w kodowaniu binarnym;

v_D – dziesiętna wartość ciągu binarnego stanowiącego zakodowaną postać liczby v_B w kodowaniu binarnym;

V – zbiór indeksów krawędzi przejść w grafie przepływu sterowania badanego oprogramowania;

w – wycinek koła w selekcji kołem ruletki, wyrażany procentowo;

w_u – rozmiar u–tego przypadku testowego;

w_d – liczba wystąpień d–tego bajtu (entropia uproszczona);

W – rozmiar zredukowanego korpusu;

x – wektor binarny będący potencjalnym rozwiązaniem w problemie MCSCP;

x_j – binarna zmienna decyzyjna w problemach SCP, MSCP, WSCP, MCSCP;

$\bar{x}$ – wektor binarny będący rozwiązaniem dopuszczalnym w problemie MCSCP;

X – zbiór przypadków testowych stanowiących rozwiązanie problemu destylacji (algorytm *MoonLight*);

X' – zbiór przypadków testowych stanowiących tymczasowe rozwiązanie problemu destylacji (algorytm *MoonLight*);

y – indeks locusów w chromosomie;

Y – zbiór rozwiązań niezdominowanych (algorytmy ewolucyjne, algorytm *VEGA*, epi*VEGA*);

Y' – tymczasowy zbiór rozwiązań niezdominowanych w algorytmach *VEGA* i epi*VEGA*;

Y_κ – najlepszy zbiór rozwiązań niezdominowanych wyznaczony przez algorytm do pokolenia κ w algorytmach *VEGA* i epi*VEGA*;

z_d – liczba wystąpień *d*–tego bajtu;

Z – liczba zebranych przypadków testowych (początkowy korpus);

α – bit znaku funkcji wykładniczej w kodowaniu logarytmicznym;

β – bit znaku wykładnika funkcji wykładniczej w kodowaniu logarytmicznym;

$\gamma(H)$ – rząd schematu H;

$\delta(H)$ – rozpiętość schematu H;

ϵ – entropia uproszczona;

$\bar{\epsilon}$ – średnia entropia przypadków testowych w zredukowanym korpusie;

ε_u – entropia *u*–tego przypadku testowego;

ζ – pozycja osobnika na liście rankingowej;

η – liczba kryteriów w MCSCP;

κ – indeks pokolenia, algorytmy *VEGA*, epi*VEGA*;

κ_p – liczba pokoleń bez poprawy w algorytmie epi*VEGA*;

θ – zmienna przyjmująca wartość [0,1];

ϑ – histon, wektor epigenetyczny w algorytmie epi*VEGA;*

ι – długość chromosomu *ch;*

λ – liczba potomków powstałych w wyniku krzyżowania pary rodzicielskiej (selekcja metodami (μ, λ), (μ+ λ));

μ – liczebność populacji rodzicielskiej (selekcja metodami (μ, λ), (μ+ λ));

ξ – długość ciągu przeznaczonego do zakodowania liczby z przedziału $[v_L, v_R]$ w kodowaniu binarnym;

τ – argument wykorzystywany przez funkcję z badanej biblioteki (algorytm *Minerva*);

$\bar{\phi}$ – średnia wartość funkcji celu w populacji;

$\phi(H)$ – średnia wartość funkcji celu osobników pasujących do H;

Ψ – zbiór wszystkich funkcji w badanej bibliotece (algorytm *Minerva*).

Wykaz akronimów oraz pojęć

Allel – w naukach biologicznych określenie opisujące warianty genu. W algorytmach genetycznych - wartość genu.

Blackbox – proces testowania oprogramowania bez znajomości kodu źródłowego.

Crawler – oprogramowanie służące do zbierania wybranych treści udostępnionych w sieci WWW.

CVE – (ang. *Common Vulnerabilities and Exposures*) lista identyfikatorów podatności i zagrożeń zarządzany przez organizację non-profit MITRE.

Deduplikator – moduł fuzzera służący do usuwania powtarzających się elementów w zbiorze znalezionych błędów.

Destylacja – proces redukcji korpusu danych testowych wykorzystywany w *fuzz* testach.

Epigenetyka – nauka, której przedmiotem jest podlegająca dziedziczeniu zmiana ekspresja genów.

epi*VEGA* – autorski algorytm, modyfikacja algorytmu *VEGA*, wzbogacona o operator wzorowany na zjawiskach epigenetycznych oraz sterowanie zbieżnością.

Exploit – program wykorzystujący podatność w innym oprogramowaniu.

Fenotyp – ogół cech osobnika, wynikających z genotypu oraz oddziaływania zjawisk zewnętrznych.

Fuzzer – oprogramowanie służące do automatycznego testowania oprogramowania pod kątem występowania w nim podatności.

Gen – w naukach biologicznych fragment kodu DNA ulegający ekspresji. W algorytmach genetycznych - wartość w chromosomie znajdująca się pod poszczególnymi locusami.

Genotyp – zestaw genów osobnika, które definiują jego cechy dziedziczne.

Graf przepływu sterowania – teoretyczna struktura danych wykorzystywana w celu prezentacji procedur aplikacji przez kompilator.

Greybox – technika testowania łącząca elementy zarówno testów białoskrzynkowych i czarnoskrzynkowych, najczęściej opiera się o znajomość formatu wejściowego plików przetwarzanych przez projekt.

Harness – (pol. uprząż) program zawierający minimalną funkcjonalność, wystarczającą jedynie do przetestowania projektu/biblioteki.

Histon – w naukach biologicznych zasadowe białko neutralizujące i wiążące kwas deoksyrybonukleinowy. W autorskim algorytmie ewolucyjnym, wektor binarny służący do wyciszania i aktywacji poszczególnych locusów w chromosomie.

Instrumentacja – inaczej też instrumentalizacja, wzbogacenie kodu oryginalnego projektu o dodatkowe wstawki, mające na celu umożliwienie uzyskania informacji zwrotnej o pokryciu kodu lub wydajności.

Korpus – wstępny zbiór plików będący wzorcem do generowania nowych przypadków testowych przez fuzzery.

Locus – w naukach biologicznych obszar chromosomu, który zajmuje pojedynczy gen. W algorytmach genetycznych - pozycja danego genu w chromosomie.

MCSCP – (ang. *Multi–Criteria Set Cover Problem)* wielokryterialny problem pokrycia zbioru.

Mitoza – podział komórki na dwie komórki potomne, posiadające identyczny materiał genetyczny.

MSCP – (ang. *Minimum Set Cover Problem)* problem minimalnego pokrycia zbioru.

Testy jednostkowe – testy pisane przez programistów, sprawdzające poprawność wyniku działania pojedynczej, odizolowanej funkcji z założonym rezultatem.

Testy regresyjne – testy oprogramowania, weryfikujące jego zamierzoną funkcjonalność (brak występowania niezamierzonych efektów) po wprowadzeniu zmian w kolejnych wersjach.

Parser – inaczej analizator składniowy. Oprogramowanie służące do określania struktury gramatycznej danych wejściowych.

Podatność – błąd oprogramowania pozwalający na przejęcie kontroli nad atakowanym urządzeniem (środowiskiem).

Pokrycie kodu – procedura mająca na celu ustalenie, które fragmenty kodu (linie, krawędzie w grafie przepływu sterowania, bloki w schemacie) zostały aktywowane przez przypadek testowy.

Sanityzacja – sposób sprawdzania, oczyszczania i filtrowania danych wejściowych wprowadzanych przez użytkowników do programów, interfejsów API, usług internetowych.

SCP – (ang. *Set Cover Problem)* problem pokrycia zbioru.

VEGA – (ang. *Vector Evaluated Genethic Alghoritm)* wielokryterialny algorytm genetyczny autorstwa D. J. Schaffera.

Whitebox – proces testowania oprogramowania z dostępnym kodem źródłowym, najczęściej wzbogacony mechanizmami instrumentacji.

WSCP – (ang. *Weight Set Cover Problem)* ważony problem pokrycia zbioru.

1. Wstęp

Fuzzing, inaczej zwany również *fuzz* testami, jest dojrzałą [1] metodą zautomatyzowanego poszukiwania podatności. Jest stosunkowo skuteczny [2], przez co zyskuje coraz większą popularność wśród analityków bezpieczeństwa oraz deweloperów oprogramowania. Jego efektywność jest na tyle istotna, że pojawiają się propozycje [3] bazujących na fuzzingu technik wytwarzania oprogramowania, które pozwalałyby na zapewnianie jakości i bezpieczeństwa [4] na etapie implementacji aplikacji.

Obecnie *fuzzing* przeprowadza się przy pomocy istniejących ogólnodostępnych narzędzi otwartoźródłowych [5, 6, 7, 8], jak i za pomocą dedykowanych dla poszczególnych projektów programów [9]. Z jego wykorzystaniem testuje się oprogramowanie, algorytmy kryptograficzne [10], urządzenia sieciowe [11], a nawet systemy kontroli dostępu [12]. Główną zaletą fuzzingu jest fakt, że przy jego użyciu sprawdza się znaczną liczbę przypadków testowych, które generowane są na istotnie różniące się od siebie sposoby [13]. W efekcie dane wejściowe przyjmują formy trudne do przewidzenia dla osób, które odpowiedzialne są za sanityzację oraz weryfikację plików, sygnałów czy strumieni danych obsługiwanych przez programy lub urządzenia. Większość z metod generowania danych testowych oparta jest na uprzednio przygotowanym zbiorze – korpusie, którego zarządzanie jest istotnym elementem całego procesu fuzzingu, tuż obok detekcji oraz zarządzania błędami czy innymi niepożądanymi zachowaniami testowanego oprogramowania.

Fuzzing jest odpowiedzią na czysto inżynierskie problemy procesu wytwarzania oprogramowania. Jego zadaniem jest wsparcie osób zapewniających jakość i bezpieczeństwo aplikacji. W związku z rosnącą popularnością opisywanej techniki, staje się ona również przedmiotem coraz liczniejszych badań naukowych. Przykładowo w publikacji [14] *fuzz* testy rozpatrywano jako łańcuchy Markowa. Taka interpretacja przedmiotowego procesu pozwoliła na opracowanie fuzzera, który umożliwił dokładniejszą eksplorację przestrzeni stanów badanego oprogramowania. Przygotowywanie zbioru danych testowych wykorzystywanych w fuzzingu sprowadza się również do procesów optymalizacyjnych [15] (np. ważony problem minimalnego pokrycia zbioru). Część fuzzerów czerpie zaś inspiracje z algorytmów sztucznej inteligencji [16] czy algorytmów genetycznych [17], co również ma usprawniać przeszukiwanie przestrzeni stanów badanego oprogramowania.

Wszystkie wyżej wskazane badania, prace, projekty czy próby umieszczenia fuzzingu w pewnych naukowych ramach były inspiracją do napisania niniejszej dysertacji. W rozprawie zaproponowano zakwalifikowanie procesu destylacji korpusu jako problemu

wielokryterialnego pokrycia zbioru, efektywnie rozwiązywanego z wykorzystaniem autorskiego algorytmu *epiVEGA*, który wzbogaca algorytm *VEGA* o oryginalny operator epigenetyczny oraz sterowanie zbieżnością.

1.1 Stan wiedzy w dziedzinie rozprawy

Pomimo wysokiej popularności fuzzingu, destylacja korpusu nie jest przedmiotem zbyt licznych badań naukowych [15, 18, 19]. Proces redukcji zbioru danych testowych rozumiany jest jako *WSCP (ang. weight set cover problem*) lub *MSCP* (*ang. minimal set cover problem*) – ważony problem pokrycia zbioru, gdzie jako wagę przyjmuje się rozmiar plików lub czas obsługi przez badany program. W przypadku wykorzystywania najbardziej popularnego obecnie fuzzera – *AFL++* [5], w czasie testów stosuje się dedykowany destylator *afl–cmin* (cmin) [20], którego zasada działania opiera się na algorytmie dedykowanym do destylacji plików. Rezultatem jego pracy jest wyłącznie przybliżone rozwiązanie problemu *WSCP*.

Zarządzanie korpusem danych testowych jest bardzo istotnym etapem w procesie fuzzingu. Odpowiednio przedestylowany korpus pozwala na przyśpieszenie testów, co przekładać się ma bezpośrednio na liczbę znalezionych błędów. Obecnie nie wykorzystuje się algorytmów optymalizacji wielokryterialnej, skupiając się wyłącznie na pojedynczej istotnej cesze zebranych plików.

Jedne z najbardziej zaawansowanych badań dotyczących destylacji przeprowadzono w 2019 r. [15] a następnie rozwinięto je dwa lata później [19]. W pracy poza formalnym zdefiniowaniem problemu destylacji korpusu, zaproponowano wysokowydajny algorytm programowania dynamicznego *Moonlight*.

Innym podejściem do destylacji korpusu jest wykorzystanie algorytmów sztucznej inteligencji [21], które zapewniają zbiory danych testowych dostosowane do popularnych fuzzerów mutacyjnych, jak wspomniany wcześniej *AFL++*. Tak jak w poprzednich przypadkach, stosowane jest wyłącznie jedno kryterium destylacji, którym jest rozmiar plików.

Najczęściej redukcja zbioru danych testowych odbywa się z wykorzystaniem algorytmów programowania dynamicznego bądź pokrewnych. Obecnie do rozwiązywania problemu destylacji korpusu nie wykorzystuje się podejścia wielokryterialnego. Jest to prawdopodobnie związane z własnościami algorytmów programowania dynamicznego, które cechują się rosnącą wraz z ilością rozpatrywanych parametrów złożonością pamięciową i czasową (adekwatnie do iloczynu maksymalnych wartości wszystkich kryteriów) [22].

Wspominanej wady pozbawione są algorytmy ewolucyjne, które jako heurystyki przeszukują równolegle przestrzeń rozwiązań z różnych jej punktów, a ukierunkowanie tego

procesu zależy głównie od uzyskiwanych danych, dotyczących aktualnie wyznaczanych rozwiązań.

Jednym ze współcześnie rozwijanych trendów w rozwoju algorytmów ewolucyjnych jest stosowanie operatorów, które inspirowane są zjawiskami epigenetycznymi, takimi jak dziedziczenie białkami prionowymi, metylacja cytozyny czy acetylacja histonu [23]. Wymienione mechanizmy odpowiadają w przyrodzie za zróżnicowanie fenotypu osobnika w następstwie występowania zewnętrznych czynników stresowych, co nie jest aktualnie uwzględniane w próbach ich odwzorowywania w algorytmach ewolucyjnych [23, 24].

Każdy z rozdziałów stanowiących część teoretyczną niniejszej rozprawy stara się wyczerpująco poszerzać wszystkie obszary poruszone w opisie stanu wiedzy. Część oryginalna przedstawia destylację korpusu danych testowych rozszerzoną o dodatkowe wagi. Jako zagadnienie obliczeniowe redukcja ta została autorsko zdefiniowana jako problem wielokryterialnego pokrycia zbioru. Sam proces destylacji pierwotnego zestawu plików testowych wykorzystywanych w procesie fuzzingu został przeprowadzony z wykorzystaniem wielokryterialnego algorytmu genetycznego, wzbogaconego o modyfikator epigenetyczny (wyzwalany czynnikiem stresowym) oraz mechanizm sterowania zbieżnością.

1.2 Cel i zakres pracy

Analizowany w niniejszej rozprawie problem badawczy dotyczy skutecznego przygotowywania korpusu danych testowych w procesie fuzzingu. W rozdziale drugim przedstawiona została historia, podział i zastosowanie *fuzz* testów. Opisano popularne, obecnie stosowane fuzzery oraz destylatory. Omówiono rolę poprawnie przygotowanego i zarządzanego początkowego zbioru danych testowych w procesie automatycznego poszukiwania podatności. Rozważono wady i zalety fuzzingu.

Rozdział trzeci został poświęcony algorytmom ewolucyjnym, ich klasyfikacji oraz historii. Poruszona została tematyka dotycząca kodowania chromosomów oraz odmian klasycznych operatorów genetycznych – mutacji i krzyżowania. Opisano metody selekcji oraz rolę zbieżności w zakończeniu pracy algorytmu ewolucyjnego. Wskazano współczesne kierunki rozwoju obliczeń ewolucyjnych.

Czwarty rozdział skupia się na tematyce zjawisk epigenetycznych i dotychczasowych próbach ich odwzorowania w postaci operatorów stosowanych w algorytmach ewolucyjnych. Na bazie przeglądu obecnych trendów w przeprowadzaniu procesu fuzzingu oraz dotychczasowych rozwiązań stosowanych w algorytmach ewolucyjnych sformułowany został

w rozdziale piątym problem badawczy, który definiuje destylację korpusu danych testowych jako problem *MCSCP* (*ang. multi-criteria set cover problem*).

Przeprowadzona analiza literatury oraz badania zrealizowane w ramach dotychczasowych publikacji pozwalają postawić tezę, że ***wykorzystanie w procesie destylacji wielokryterialnego algorytmu genetycznego wzbogaconego o operator epigenetyczny oraz mechanizm sterowania zbieżnością pozwoli na efektywną redukcję korpusu danych testowych w procesie fuzzingu***.

W pracy zaproponowany został autorski operator epigenetyczny, wzbogacający algorytm *VEGA* [25], który rozszerzono również o sterowanie zbieżnością. Tak rozbudowane rozwiązanie posłużyło do przeprowadzenia destylacji korpusów plików w formatach **.gif*, **.lz4*, **.xml* oraz **.pdf*. Ustalanie prawdopodobieństwa wystąpienia zjawiska epigenetycznego, ocena skuteczności zaproponowanego rozwiązania i wykorzystanie w procesie fuzzingu wyznaczonych z jego użyciem korpusów zostały opisane w rozdziale szóstym. Skupiono się przy tym na wskazaniu różnic między korpusem otrzymanym przez zaproponowany autorski algorytm *epiVEGA* (*ang. epigenetic Vector Evaluated Genetic Algorithm*), a zbiorem niezredukowanym (pełnym), pustym, wytypowanymi za pomocą destylatora bazującego na oryginalnym algorytmie *VEGA* (*ang. Vector Evaluated Genetic Algorithm*) i destylatora *afl–cmin*.

Praca zakończona została wnioskami na temat przeprowadzonych eksperymentów oraz propozycjami przyszłych kierunków i potencjalnego rozwoju badań zrealizowanych w trakcie pisania niniejszej rozprawy.

2. Fuzzing

Fuzzing, (inaczej - *fuzz* testy), jest automatyczną, pseudolosową metodą testowania oprogramowania. Polega na wprowadzaniu na wejście programu losowych, zmodyfikowanych lub błędnych danych testowych w celu znalezienia błędów nieprzewidzianych przez autora aplikacji (nieobsłużonych za pomocą mechanizmów wykorzystywanego języka programowania) [26]. Dane testowe są generowane bądź powstają w wyniku mutowania plików wchodzących w skład zebranych wcześniej zbiorów danych testowych, tzw. korpusów. Zazwyczaj *fuzzing* stosuje się w celu testowania programów, które przyjmują na wejściu dane o określonej strukturze (np. pliki o ustalonym formacie) lub programów realizujących protokoły komunikacyjne.

Za pomocą fuzzingu nie poszukuje się w oprogramowaniu błędów logicznych, lecz tych związanych z niepoprawnym lub niebezpiecznym wykorzystaniem języka programowania przez autora aplikacji.

Współcześnie *fuzzing* stosuje się najczęściej podczas audytów bezpieczeństwa oprogramowania pisanego w językach takich jak C/C++, nieposiadających wystarczających mechanizmów bezpieczeństwa. Uniwersalność tej metody pozwala także na testowanie rozwiązań sprzętowych i systemów operacyjnych [27]. *Fuzz* testy odnoszą również sukcesy w trakcie badania jakości implementacji algorytmów kryptograficznych [28].

Charakterystycznymi błędami znajdowanymi za pomocą fuzzerów są błędy ochrony pamięci (czyli sytuacje, kiedy program uzyskuje dostęp do obszaru pamięci nie zaalokowanej dla niego), przepełnienia buforów, stosów czy zmiennych (kiedy rozmiar struktury przekroczy rozmiar pamięci dla niej zaalokowanej) [29]. Błędy te pozwalają na nieautoryzowany dostęp do obszarów pamięci programu, wbrew intencji autora. Z wykorzystywaniem fuzzingu atakujący najczęściej badają moduły oprogramowania, które nie są już wspierane bądź są jeszcze w procesie wdrażania. Głównym przedmiotem testów są parsery danych, biblioteki multimedialne, tekstowe, interpretery języków, zależne podprogramy większych projektów i biblioteki zewnętrze, na których bezpieczeństwo nie mają wypływu deweloperzy oprogramowania, będących przedmiotem badań lub potencjalnego ataku.

2.1 Historia fuzzingu

Pierwszymi próbami zautomatyzowania testów oprogramowania poprzez wykorzystanie losowych danych testowych były tzw. testy małpie (*ang. monkey tests*) [30]. W roku 1983

Steve Capps, jeden z współautorów komputerów osobistych *Apple Macintosh*, pracował nad demonstracyjną płytą „*Guided Tour*”, której uruchomienie miało sprawić, iż system operacyjny dokonywał autoprezentacji swoich możliwości. Podczas pracy nad projektem Capps stwierdził, że wykorzystywane do prezentacji „haki” (ang. *hook* – techniki umożliwiające ingerencje w zachowanie oprogramowania, poprzez kontrolę nad wywołaniami funkcji) mogą zostać wykorzystane do automatyzacji testów i dać podstawy do stworzenia narzędzia „*The Monkey*” (pol. małpa). Była to niewielka aplikacja, która wprowadzała do uruchomionego programu losowe zdarzenia, wyglądające dla pobocznego obserwatora na próbę obsługi komputera przez rozjuszone zwierzę. Narzędzie pozwoliło na znalezienie szeregu błędów w aplikacjach takich jak *MacPaint* i *MacWrite*.

W 1975 roku [31] w sposób eksperymentalny wykazano słuszność wykorzystywania losowych testów przy weryfikowaniu poprawności zaprojektowania dużych obwodów logicznych.

Same *fuzz* testy zostały opracowane i zdefiniowane na Uniwersytecie *Wisconsin Madison* w 1989 roku. Celem projektu pierwotnie było wykrywanie błędów linii poleceń oraz badanie interfejsu użytkownika systemów operacyjnych [32]. Projekt pozwolił ostatecznie na wykrycie również błędów o całkowicie odmiennej naturze, które mogły być klasyfikowane jako podatności. Narzędzia wykorzystane w czasie eksperymentów zostały udostępnione publicznie (a dokładniej – ich kod źródłowy), dając początki współczesnemu fuzzingowi. Opisywany projekt dalej jest wspierany, co pozwala na regularne znajdowanie błędów w oprogramowaniu. Jego rozwój na przestrzeni lat prezentuje się następująco:

- W 1995 roku na ww. uniwersytecie opublikowano czteroczęściowy przegląd wykorzystania fuzzingu w procesie zapewniania jakości [33]. Odtworzono *fuzz* testy, których przedmiotem były aktualne wersje systemów *UNIX* oraz popularne narzędzia konsolowe. Badania wykazały, iż niezawodność oprogramowania na z czasem znacznie się pogorszyła. Były to pierwsze prace badawcze dotyczące niezawodności, które obejmowały narzędzia na licencji *GNU* i systemy *Linux* o otwartym kodzie źródłowym. Publikacja prezentowała również wykorzystanie fuzzingu w badaniach aplikacji z graficznym interfejsem użytkownika (dla systemu *Windows*) zgodnych z *X–Windows*. W testach wykorzystano zarówno nieustrukturyzowane (niepoprawne), jak i ustrukturyzowane (prawidłowe sekwencje użycia myszy i klawiatury) dane wejściowe. W efekcie udało się doprowadzić do zawieszenia 25% badanych aplikacji. Testom poddano również sam serwer *X–Windows*. *Fuzzing* modułu wykazał, że jest on odporny na tego typu „ataki”. Trzecia część artykułu

opisywała próbę testowania usług sieciowych opartą na ustrukturyzowanych (zgodnych z badanym protokołem) testowych danych wejściowych. Żadna z nich nie uległa awarii. Ostatni rozdział publikacji poświęcono losowemu testowaniu wartości zwracanych przez wywołania bibliotek systemowych (w szczególności funkcji z rodzinny *malloc*). Prawie połowa standardowych programów *UNIX* nie potrafiła poprawnie zweryfikować otrzymywanych wartości.

- W 2000 roku zastosowano *fuzzing* do testowania systemu operacyjnego *Windows NT* [34], weryfikując aplikacje wykorzystujące interfejs *Win32*. Badaczom udało się znaleźć nieprawidłowości w 21% badanych aplikacji oraz zawiesić 24% programów. Analiza znalezionych błędów pozwoliła ustalić, że ich źródło na przestrzeni lat nie zmieniło się. Za występowanie większości błędów odpowiedzialne było ustawienie losowej wartości wskaźników bądź błędne zastosowanie wywołań zwrotnych funkcji (ang. *callback*).
- *Fuzzing* okazał się również skuteczny w testowaniu oprogramowania uruchamianego w systemie *Apple Mac OS X* [35]. W roku 2006 przeprowadzono testy oprogramowania konsolowego oraz posiadającego graficzny interfejs użytkownika. Badania te zakończyły się znalezieniem błędów w 7% z 135 sprawdzanych aplikacji obsługiwanych z wiersza poleceń. Spośród 30 aplikacji uruchamianych w środowisku graficznym *MacOS Aqua* w 73% znaleziono błędy.
- W trakcie testów przeprowadzonych na uniwersytecie *Wisconsin–Madison* w 2020 roku po raz kolejny wykorzystano pierwotne techniki fuzzingu [36]. Ponownie zastosowane zostały klasyczne testy czarnoskrzynkowe, przyjmujące na wejściu nieustrukturyzowane dane. Przedmiotem badań było oprogramowanie uruchamiane w obecnych systemach *UNIX*: *Linux*, *FreeBSD* i *MacOS*. Celem badań było sprawdzenie, czy oryginalny *fuzzing* nadal jest skuteczny i czy obecne programy są odporne na tego typu ataki. Przetestowano ok. 75 narzędzi na każdej z ww. platform, ze wskaźnikami awaryjności wynoszącymi 12% dla systemu *Linux*, 16% na *Apple MacOS* i 19% na *FreeBSD*. Ich analiza wykazała, że klasyczne awarie, takie jak błędy związane z implementacją wskaźników, tablic czy zastosowaniem wywołań zwrotnych funkcji nadal występują stosunkowo często we współczesnym oprogramowaniu. Dodatkowo zostały wykryte błędy, które wynikały z wysokiego stopnia złożoności współczesnego oprogramowania i stosowanych w nim algorytmów. Testom poddano również programy napisane w języku *Rust* [37], które

nie okazały się wolne od błędów, chociaż ich liczba była mniejsza niż w programach napisanych w języku C, co jest zgodne z założeniami autorów tego języka.

Równolegle *fuzzing* rozwijany był przez podmioty prywatne jak i niezależnych badaczy, jako skuteczna metoda zapewniania jakości. W kwietniu 2012 [38] firma *Google* na swoim blogu ogłosiła uruchomienie infrastruktury chmurowej, przeznaczonej do testowania przeglądarki *Chrome*, nazwanej „*ClusterFuzz*". Była ona zbudowana na bazie klastra kilkuset maszyn wirtualnych, za pomocą której uruchamiano ok. sześć tysięcy instancji *Chrome*. „*ClusterFuzz*" skonfigurowany został tak, aby automatycznie pobierał najnowszą wersję przeglądarki, która miała przetwarzać ok. 50 milionów przypadków testowych dziennie. Przepustowość infrastruktury od czasu jej uruchomienia została zwiększona szesnastokrotnie.

Kilka miesięcy później zaprezentowana [39] została grupa błędów bezpieczeństwa powłoki *bash* systemów *UNIX* – „*Shellshock*". Większość z tych podatności znaleziono przy użyciu fuzzera *AFL* [17]. Wiele usług dostępnych z poziomu Internetu, takich jak serwery WWW, używa powłoki *bash* do przetwarzania części żądań, umożliwiając atakującemu wykorzystanie podatnych wersji ww. oprogramowania do wykonania dowolnych poleceń, co mogło doprowadzić do nieautoryzowanego dostępu do systemu operacyjnego.

W tym samym roku uruchomiono również projekt *Stagefright fuzzer* [40] (inaczej *MFFA* – ang. *Media Fuzzing Framework for Android*). Główną jego ideą było tworzenie uszkodzonych, ale strukturalnie prawidłowych danych multimedialnych. Pliki te następnie kierowane były do odpowiednich komponentów w systemie *Android* w celu ich odkodowania lub odtworzenia. W tym czasie monitorowano testowane oprogramowanie pod kątem występowania potencjalnych problemów, które mogłyby zostać wykorzystane jako luki bezpieczeństwa. Projekt został napisany w języku *Python* [41]. Skrypty były używane do wysyłania zniekształconych danych w zrównoleglonej infrastrukturze urządzeń z systemem *Android*, rejestrowania wyników i monitorowania wystąpienia ewentualnych błędów w sposób zautomatyzowany. Właściwe dekodowanie plików multimedialnych na urządzeniach mobilnych odbywało się za pomocą uruchamianego z wiersza poleceń programu *Stagefright*. Od jego nazwy pochodzi określenie całej grupy podatności systemu *Android*. Najbardziej znanym jest błąd CVE–2015–1538 [42], którego prezentacja (*ang. PoC, proof of concept*) odbyła się poprzez przesłanie spreparowanej wiadomości MMS. Samo jej otrzymanie, bez ingerencji użytkownika, pozwalało na przejęcie kontroli nad urządzeniem.

W grudniu 2016 roku, firma *Google* ogłosiła uruchomienie projektu *OSS–Fuzz* [43], pozwalającego na przeprowadzanie zaawansowanego fuzzingu popularnych projektów typu

open–source. Celem *OSS–Fuzz* jest zapewnienie większego bezpieczeństwa programów otwartoźródłowych, których twórcy często nie posiadają odpowiedniej infrastruktury do przeprowadzania fuzzingu na dużą skalę. Projekt wspiera aktualnie wykorzystywane fuzzery służące do testowania oprogramowania napisanego w językach, C/C++, Rust, Go [44], Python i Java [45].

Podobne do wcześniej opisanego rozwiązanie zaproponowała firma *Microsoft* prezentując w 2020 roku projekt *OneFuzz* [46]. Jest to platforma, która w przystępny sposób pozwala programistom na przeprowadzenie fuzzingu w skalowalnym, rozporoszonym środowisku.

Ostatnim znaczącym wydarzeniem związanym z fuzzingiem jest znalezienie z jego wykorzystaniem podatności CVE–2020–16747 przez Michała Jurczyka [47]. Podobnie jak błędy z rodziny *Stagefright* pozwalała ona na przejęcie telefonu z systemem *Android*, również bez ingerencji użytkownika. Problem ten dotykał wszystkich smartfonów firmy Samsung wyprodukowanych po 2015 roku.

2.2 Podział fuzzingu ze względu na znajomość kodu

Fuzzing można dzielić ze względu na poziom znajomości kodu źródłowego:

- *Fuzzing* czarnoskrzynkowy (ang. *blackbox*) [48] cechuje się brakiem znajomości kodu źródłowego przez testera. Bazuje wyłącznie na możliwości oceny poprawności wykonania programu.
- *Fuzzing* szaroskrzynkowy (ang. *greybox*) [49] umożliwiają analizę kodu po znalezieniu błędu, bądź dokładną analizę zachowania programu po wprowadzeniu przypadku testowego (np. pokrycia kodu). Znana jest dobrze struktura plików wejściowych.
- *Fuzzing* białoskrzynkowy (ang. *whitebox*) [50] przeprowadza się w sytuacji, kiedy znany jest kod programu, dokładne informacje o jego zachowaniu w trakcie jego wykonywania oraz istnieje możliwość bieżącej analizy kodu źródłowego.

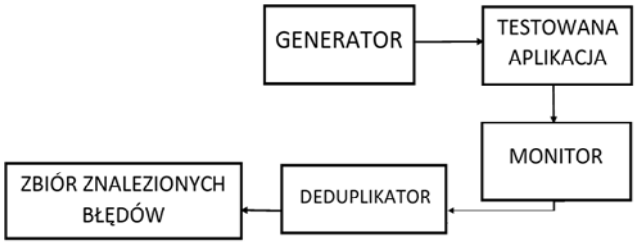
2.3 Podział fuzzingu ze względu na sposób generowania przypadków testowych

Istotną częścią *fuzz* testów jest generowanie danych testowych. Odpowiedni dobór metody znacząco wpływa na skuteczność testów. Algorytm nie może tworzyć zestawów testowych, które byłyby odrzucone przez testowany program. Jednocześnie wskazane jest,

aby zawierał on w sobie jak najwięcej elementów, których obsłużenia nie przewidział autor badanego programu.

2.3.1 Dumb fuzzing

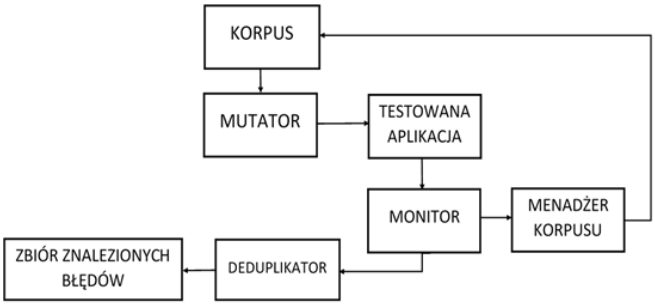
Metoda pseudolosowa – polega na tworzeniu danych od podstaw z wykorzystaniem generatora pseudolosowego i podawaniu ich testowanej aplikacji jako wartości wejściowej. Rozwiązanie to sprawdza się w weryfikacji poprawności walidacji oraz implementacji przechowywania danych w pamięci (np. kiedy wygenerowane przez *fuzzer* dane są zbyt duże). Cechuje się jednak znaczną liczbą odrzuconych przypadków testowych, co zmniejsza skuteczność samego fuzzingu.



Rys. 1 Schemat fuzzera bazującego na generatorze przypadków testowych.

2.3.2 Guided fuzzing

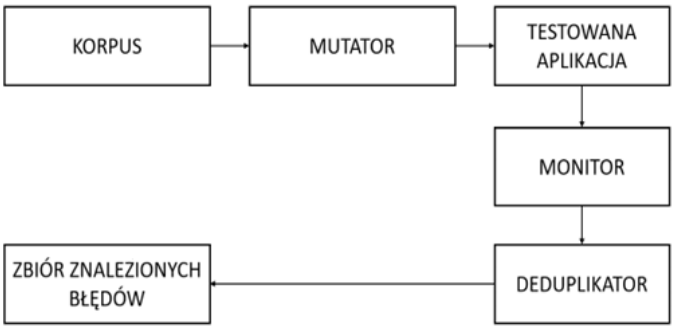
Fuzzing sterowany wykorzystuje instrumentalizację badanego oprogramowania. Na podstawie wybranych parametrów pracy programu wygenerowane przypadki testowe oceniane są pod kątem dalszej ich użyteczności oraz dokonywana jest ich selekcja. Podczas fuzzingu oceniany może być czas obsługi przypadku testowego, wykorzystanie zasobów sprzętowych czy pokrycie kodu [51]. Pokrycie kodu można mierzyć w pokryciu linii kodu, liczby aktywowanych przez przypadek testowy bloków kodu lub aktywowanych krawędzi w diagramie przepływu sterowania.



Rys. 2 Schemat fuzzera wykorzystującego mechanizm instrumentalizacji.

2.3.3 Mutation based fuzzing

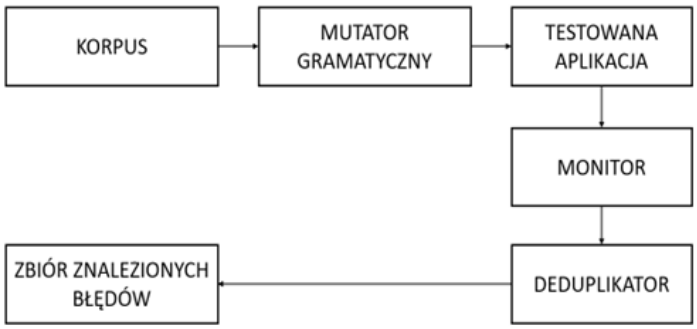
Metoda mutacyjna bazuje na zbiorze przypadków testowych – korpusie. Przypadki te mogą być publikowane przez autorów badanego oprogramowania lub samodzielnie zebrane przez osobę przeprowadzająca testy, np. z wykorzystaniem Internetu. Mogą także być wyselekcjonowanymi przypadkami testowymi, które zostały wybrane w czasie poprzednio przeprowadzanych testów. Pliki te się mogą cechować się znacznym pokryciem kodu badanej aplikacji bądź tym, że wywoływały błąd innego wcześniej badanego oprogramowania (lub poprzedniej wersji obecnie testowanego programu). Przypadki te modyfikowane są losowo, (w niewielkim stopniu), bez uwzględniania poprawności ich struktury.



Rys. 3 Schemat fuzzera wykorzystującego mutator.

2.3.4 Generation based fuzzing

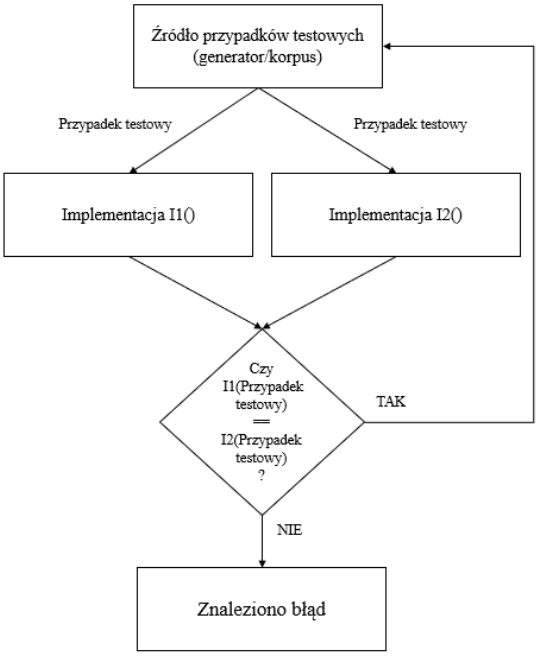
Metoda gramatyczna w przeciwieństwie do metody mutacyjnej, uwzględnia strukturę danych testowych. Modyfikacje przeprowadzane są w taki sposób, aby nowopowstały przypadek testowy był poprawny pod względem budowy formatu pliku. Technika gramatyczna może bazować na korpusie przypadków testowych lub jak w przypadku metody pseudolosowej, generować całkowicie nowe, lecz poprawnie strukturalnie dane. W celu tworzenia przypadków testowych zgodnych z formatem pliku stosowane są specjalne „słowniki” składające się z fragmentów plików, których wykorzystanie pozwala na wygenerowanie poprawnych danych.



Rys. 4 Schemat fuzzera gramatycznego opartego o korpusie.

2.3.5 Differential fuzzing

Testowanie różnicowe [52], znane również jako *fuzzing* różnicowy, to technika testowania oprogramowania, która wykrywa błędy poprzez dostarczenie tych samych danych wejściowych do serii podobnych aplikacji (lub różnych implementacji tego samego programu) i obserwację różnic w wyniku ich wykonania. Stosowany jest najczęściej w przypadku przepisywania projektu na nowy język programowania lub porównywania implementacji sprzętowych do programowych.



Rys. 5 Schemat działania fuzzingu różnicowego.

Istnieją odmiany fuzzingu różnicowego [53], które poza różnicami w danych wynikowych oprogramowania porównują paramenty jego pracy, takie jak wykorzystanie zasobów sprzętowych czy czas obsługi przypadku testowego.

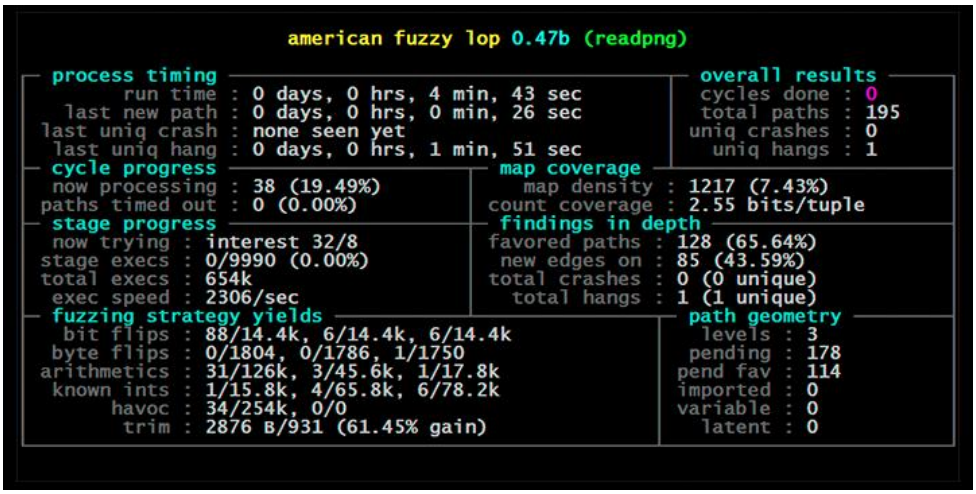
2.4 Współczesne fuzzery

2.4.1 Fuzzer AFL

Oprogramowanie *American fuzzy lop* [17] to fuzzer wykorzystujący instrumentalizację kodu oraz mechanizm podobny do algorytmu ewolucyjnego (często AFL błędnie jest określany fuzzerem bazującym na algorytmie genetycznym). Algorytm ten umożliwia automatyczne wykrywanie wartościowych przypadków testowych, które wybudzają nowe krawędzie w grafie przepływu sterowania badanego pliku wykonywalnego. Działa on według następującej listy kroków:

1. Ładowanie początkowych przypadków testowych do kolejki.
2. Wczytanie pierwszego/kolejnego przypadku z kolejki.
3. Przycięcie pliku do najmniejszego rozmiaru, który nie zmieni zachowania programu.
4. Wykonanie modyfikacji pliku zgodnie z wybraną strategią klasyczną (zazwyczaj mutacyjną).
5. Jeżeli któryś z nowo wygenerowanych przypadków zwiększa pokrycie kodu lub wywołuje błąd, dodanie go do kolejki przypadków testowych.

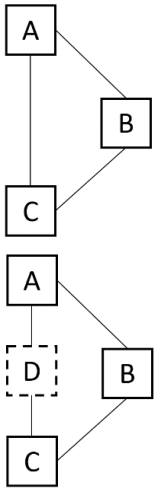
Korpusy wytwarzane przez opisywane narzędzie są przydatne do przeprowadzania testów w przyszłości, ponieważ cechują się wysokim pokryciem kodu.



Rys. 6 interfejs american fuzzy lop

Instrumentalizacja programu odbywa się poprzez podzielenie każdej występującej w nim krawędzi krytycznej w grafie przepływu sterowania (czyli takiej, której aktywacji nie da się jednoznacznie wykryć z wykorzystaniem informacji o aktywnych blokach) na dwie za

pomocą nadmiarowych, „sztucznych” bloków. W ten sposób można określić jaką ścieżkę aktywował dany przypadek testowy.



Rys. 7 Schemat przedstawiający sposób instrumentalizacji krytycznej krawędzi „A–C” poprzez dodanie bloku „D”

2.4.2 Fuzzer AFL++

AFLplusplus, *AFL++*, [5] jest następcą fuzzera *AFL* autorstwa Michała Zalewskiego i został stworzony, aby uwzględnić wszystkie najlepsze modyfikacje opracowane przez lata dla różnych rozwidleń (ang. *fork*) *AFL*, które ostatecznie nie zostały wdrożone do programu, ponieważ jego wsparcie skończyło się w listopadzie 2017 r. *Fuzzer AFL++* składa się z następujących modułów:

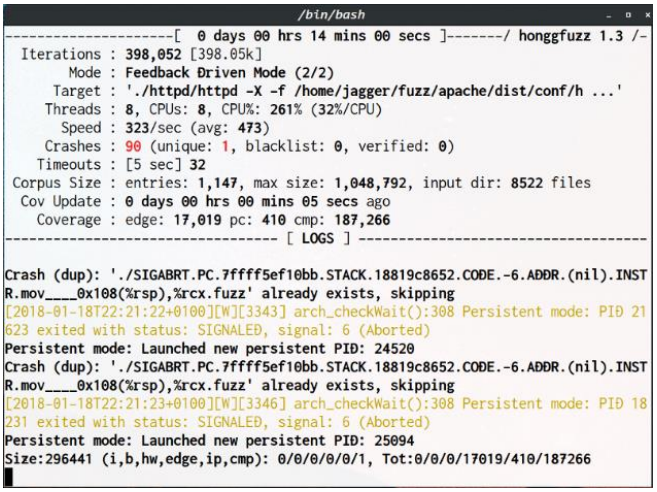
- *afl–fuzz* – fuzzer z wieloma mutatorami i konfiguracjami;
- moduły instrumentacji kodu źródłowego: tryb *LLVM*, *afl–as*, wtyczka *GCC*;
- moduły instrumentacji kodu binarnego: tryb *QEMU*, tryb *Unicorn*, tryb *QBDI*;
- narzędzia do minimalizacji przypadków/korpusów testowych*: afl–tmin*, *afl–cmin*;
- biblioteki pomocnicze: *libtokencap*, *libdislocator*, *libcompcov*.

2.4.3 Fuzzer HonggFuzz

HonggFuzz to *fuzzer* „ewolucyjny” (podobnie jak AFL) [7] przeznaczony do wyszukiwania błędów bezpieczeństwa, posiadający rozbudowane opcje analizy danych wynikowych testów przeprowadzonych za jego pomocą.

HonggFuzz jest oprogramowaniem wieloprocesowym i wielowątkowym, dzięki czemu nie ma potrzeby uruchamiania znacznej liczby jego instancji, ponieważ *fuzzer* ten

wykorzystuje w pełni zasoby sprzętowe platformy, na której jest uruchamiany. Korpus plików testowych jest automatycznie udostępniany i ulepszany między wszystkimi procesami *HonggFuzza*.



Rys. 8 Interfejs programu HonggFuzz.

HonggFuzz można wykorzystywać do testowania oprogramowaniu uruchamianego w systemach *Linux*, *NetBSD*, *FreeBSD*, *Microsoft Windows*, *Apple Mac OS X* oraz *Android*.

Aplikacja używa niskopoziomowych interfejsów do monitorowania procesów (np. *ptrace* [54] w systemach *Linux* i *NetBSD*). W przeciwieństwie do innych fuzzerów, wykrywa i zgłasza przechwycone sygnały awarii (potencjalnie ukryte przez program poddany testom).

HonggFuzz ma bogatą historię wykrytych błędów bezpieczeństwa (w tym m.in. 25 podatności zakwalifikowanych jako CVE). Jednym z bardziej znanych jest luka CVE–2016–6309 w znaleziona *OpenSSL* [55].

HonggFuzz cechuje się intuicyjnym i prostym sposobem użycia. Można za jego pomocą uruchamiać testy wykorzystujące korpus lub rozpocząć testowanie bez początkowego zbioru przypadków testowych. Do projektu dołączony jest szereg przykładów rzeczywistych ustawień fuzzera dla powszechnie używanego oprogramowania (np. *Apache HTTPS*, *OpenSSL*, *libjpeg* itd.).

2.4.4 Fuzzer LibFuzz

Libfuzzer [8] jest klasycznym fuzzerem ewolucyjnym, który na bieżąco wykorzystuje dane dotyczące pracy testowanego oprogramowania w celu zwiększenia pokrycia kodu. Zasada jego działania zbliżona jest do testów jednostkowych [56]. Testom poddawana jest jedna z funkcji oprogramowania, dla której pisany jest test, nazywany uprzężą (ang. *harness*).

Główną cechą wyróżniającą *Libfuzzer* jest sposób uruchamiania testowanego oprogramowania – po teście nie jest ono wyłączane, lecz po obsłużeniu danych wejściowych podawane są mu kolejne przypadki testowe. Podejście to pozwala na zwiększenie liczby wykonanych testów, ponieważ pomija wszystkie czynności związane z przygotowaniem pamięci dla procesów i wątków, co w efekcie oszczędza czas potrzebny do obsługi pojedynczego przypadku testowego. Informacje o pokryciu kodu są zbierane przez moduł *SanitizerCoverge LLVM* [57], a szczegółowe informacje o wykrytych błędach uzyskuje się dzięki modułom *Memory Sanitizer* i *UndefinedBehaviourSanitizer*.

2.4.5 Fuzzer Minerva_lib (Polish Fuzzy Lop)

Minerva_lib (lub inaczej: *Polish Fuzzy Lop*) to *fuzzer* autorstwa polskiego badacza bezpieczeństwa, Macieja Kocielskiego [6], będący we wczesnej fazie rozwoju. Bazuje na algorytmie *Minerva*, który pierwotnie wykorzystywany był do fuzzingu interpreterów języka *PHP*. Zasada działania algorytmu jest następująca: tworzony jest zbiór Ψ wszystich funkcji, które wchodzą w skład badanej biblioteki oraz inicjalizowany jest zbiór argumentów i ich typów M. Ze zbioru Ψ wybierana jest losowo funkcja f, do której wywołania potrzebne są elementy należące do M. W pierwszej iteracji wybierane są funkcje, które nie przyjmują argumentów, ponieważ na tym etapie zbiór $M = \emptyset$. Wynik wywołania f, τ dodawany jest do zbioru M. Procedurę powtarza się do znalezienia błędu.

Minevra_lib wykorzystywany jest głównie do testowania bezpieczeństwa oprogramowania typu *open source*. Na tle obecnie stosowanych fuzzerów wyróżnia go brak zastosowanej instrumentalizacji.

2.5 Rola korpusu

Dla fuzzerów, które nie opierają się na generatorze przypadków testowych, korpus stanowi istotną bazę dla rozpoczęcia testów. Odpowiednio przygotowane pliki zwiększają liczbę wykonanych testów oraz znalezionych błędów [13]. Wyniki przywołanego porównania wskazują na istotną rolę korpusu danych w procesie fuzzingu. W pełni losowo wygenerowane dane są mało skuteczne, ze względu na dużą liczbę odrzuconych przypadków testowych. Zbyt rozbudowane korpusy skutkują dużą liczbą braków odpowiedzi (ang. *time–out*), co negatywnie wpływa na liczbę wykonanych testów. Zbiory przypadków testowych, które są zbyt małe, nie są w stanie znaleźć zadowalającej liczby błędów bezpieczeństwa.

2.6 Tworzenie korpusu i destylacja

2.6.1 Źródło pików

Przypadki testowe wchodzące w skład korpusu mogą być publikowane przez autorów oprogramowania (np. [58]) lub samodzielnie zbierane przez osobę przeprowadzająca testy (np. z wykorzystaniem crawlerów internetowych). Mogą być także wyselekcjonowanymi przypadkami testowymi, które zostały wybrane w czasie poprzednio przeprowadzanych testów (np. [59]). Zbiory takie cechują się znacznym pokryciem kodu badanej aplikacji lub tym, że wywoływały błąd w innym wcześniej badanym oprogramowaniu (bądź w poprzedniej wersji obecnie testowanego programu).

2.6.2 Destylacja

Destylacja korpusu polega na jego redukcji w taki sposób, aby nowo uzyskany zbiór pozwalał na zwiększenie efektywności *fuzz* testów [15, 18, 19]. Proces ten może zostać oparty na różnorodnych kryteriach redukcji, takich jak rozmiar przypadków testowych, liczebność całego korpusu, czas obsługi poszczególnych przypadków testowych czy sumaryczne pokrycie kodu badanego oprogramowania przez korpus. Na podstawie wybranych parametrów pracy programu przypadki testowe oceniane są pod kątem dalszej ich użyteczności oraz dokonywana jest ich selekcja. Poza redukcją liczebności zbioru stosuje się także skrypty, które „przycinają” przypadki testowe (np. *afl-tmin* [60]). Z plików usuwane są fragmenty, których brak nie wpływa negatywnie na stopień pokrycia kodu przez przypadek testowy. Proces destylacji korpusu, który jako kryterium optymalizacji zbioru przyjmuje pokrycie krawędzi w grafie przepływu sterowania, można utożsamiać z problemem minimalnego pokrycia zbioru [15]. Obecnie wykorzystywane do destylacji korpusów programy dodatkowo wprowadzają wagi dla podzbiorów, rozumiane jako rozmiar poszczególnych przypadków testowych (*cmin* [20], *Moonight* [15]), rozszerzając problem do problemu ważonego minimalnego pokrycia zbioru. Poza zachowaniem stopnia pokrycia krawędzi na poziomie pierwotnego zbioru, celem destylacji ważonej jest redukcja rozmiaru ostatecznego korpusu. Zamiennie do rozmiaru plików stosuje się kryterium czasu obsługi pliku przez testowany program. Redukcja czasu obsługi ma na celu zwiększenie liczby wywołań programu w czasie fuzzingu, co przekłada się bezpośrednio na większą liczbę znalezionych błędów w tym samym czasie [61].

2.6.3 Współczesne metody destylacji

2.6.3.1 Destylator afl–cmin

Skrypt *afl–cmin* (inaczej: *cmin*) [20] jest narzędziem służącym do wyznaczania najmniejszego podzbioru plików dla wskazanego zbioru przypadków testowych (korpusu) w taki sposób, aby podzbiór ten wybudzał taką samą liczbę krawędzi w grafie przepływu sterowania, co pierwotny zbiór. Skrypt ewaluuje początkowy korpus danych testowych przed rozpoczęciem procesu fuzzingu. Następnie usuwa pliki, których obecność w korpusie nie zmienia pokrycia krawędzi. Dodatkowo może minimalizować korpusy generowane podczas pracy fuzzera *AFL*, poprzez usuwanie wszystkich plików, które wzbudzają unikalne ścieżki, jednak co do których istnieje duże prawdopodobieństwo, że mogą zostać zastąpione poprzez pliki wygenerowane dynamicznie w czasie fuzzingu. Zasada działania programu jest następująca. Program sortuje wszystkie pliki pod kątem liczby wybudzanych krawędzi. Oznacza jako przeznaczone do umieszczenia w przedestylowanym korpusie te pliki, które wzbudzają unikalne krawędzie, ponieważ są one niezbędne do stworzenia najlepszego korpusu. Dla krawędzi, które są wzbudzane przez więcej niż jeden przypadek testowy, wybiera te, które cechują się najmniejszym rozmiarem. Autor sugeruje, że empirycznie dowiódł, że w ten sposób powstają mniejsze zbiory danych, niż przy zastosowaniu bardziej skomplikowanych algorytmów. Sam skrypt w żaden sposób nie modyfikuje przypadków testowych. W celu dodatkowego zmniejszenia poszczególnych plików stosuje się dodatkowo skrypt *afl–tmin*, którego zadaniem jest usunięcie z przypadków testowych tych ich fragmentów, bez których nie zmienia się pokrycie krawędzi w grafie przepływu sterowania.

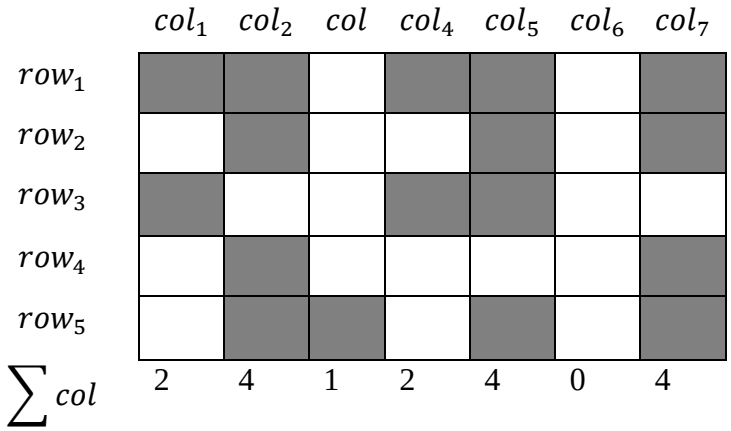
2.6.3.2 Destylator Moonlight

Oprogramowanie *MoonLight* [15] jest destylatorem, który utożsamia destylację jako ważony problem minimalnego pokrycia zbioru i oblicza rozwiązanie przy użyciu programowania dynamicznego.

MoonLight interpretuje pokrycie korpusu jako macierz: każdy wiersz jest wektorem bitowym odpowiadającym jednemu przypadkowi testowemu z pierwotnego korpusu, a każda kolumna reprezentacją zbioru krawędzi między blokami w grafie przepływu sterowania badanego oprogramowania. Jeżeli dana krawędź jest wzbudzana przez przypadek testowy reprezentowany przez wiersz, to odpowiadająca mu pozycja w kolumnie wynosi 1, w przeciwnym przypadku 0. Zgodnie z założeniami autorów współczesnych fuzzerów (w szczególności *AFL*), autorzy algorytmu *MoonLight* przyjęli, że zachowanie pierwotnego

pokrycia krawędzi w przedestylowanym korpusie pozwoli na znalezienie takiej samej liczby błędów co podczas fuzzingu z wykorzystaniem pierwotnego korpusu, jednak w krótszym czasie.

Celem algorytmu jest znalezienie najmniejszego podzbioru przypadków testowych (wierszy), które pokrywają wszystkie niezerowe kolumny (krawędzie) w macierzy A. Rozwiązaniem problemu minimalnego pokrycia zbioru jest najmniejszy zestaw wierszy (tj. przypadków testowych), który obejmuje wszystkie kolumny. Algorytm *Moonlight* dodatkowo uwzględnia wagę poszczególnych wierszy, która może reprezentować rozmiar poszczególnych przypadków testowych bądź czas niezbędny do ich obsługi przez badany program.



Rys. 9 Przykład macierzy pokrycia [15].

W tym celu algorytm *Moonlight* bazuje na następujących operacjach:

- Usuwanie osobliwości (ang. *singularities*) – osobliwości są kolumnami lub wierszami w macierzy A, których suma reprezentowanych wartości wynosi zero. Według autorów występują one rzadko, ponieważ reprezentują przypadki testowe, które nie wybudzają żadnej krawędzi w grafie przepływu sterowania badanego oprogramowania. Przypadki takie należy usunąć z rozwiązania, ponieważ ich brak nie zmniejsza pokrycia.
- Wyszukiwanie egzotycznych wierszy (ang. *exotic rows*) – egzotyczna kolumna k^* w A składa się z jednego elementu. Egzotyczny wiersz to unikalny wiersz r^* w A, który zawiera k^*. Wszystkie przypadki testowe związane z egzotycznymi wierszami zawsze są częścią ostatecznego przedestylowanego korpusu.
- Wyszukiwanie wierszy dominujących (ang. *dominant rows*) – niektóre wiersze w A mogą być podzbiorami innych wierszy, więc mogą zostać zdominowane. Dominujące wiersze, które same nie są zdominowane, określa się mianem dominujących

wierszy pierwotnych. Jeśli wiersz będący podzbiorem wiersza dominującego ma większą wagę, może zostać usunięty. W przeciwnym przypadku pozostawia się go w macierzy *A*, ponieważ może to ostatecznie doprowadzić do zmniejszenia rozmiaru korpusu.

- Usuwanie dominujących kolumn (ang. *dominant columns*) – dominacja kolumn jest relacją podobną do dominacji wierszy. Niektóre kolumny w *A* mogą być podzbiorem innych kolumn. Jednak w tej operacji kolumna dominująca zostaje usunięta, a kolumny podległe zostają pozostawione. Kolumna dominująca jest zbędna i można ją bezpiecznie usunąć.
- Usuwanie kolumn zawierających (ang. *contained columns*). Wybierając wiersz, który ma być składową rozwiązania, należy wyeliminować wszystkie kolumny, które się z nim przecinają. Kolumny można bezpiecznie usunąć, ponieważ zostaną one pokryte przypadkami testowymi skojarzonymi z wierszem. Operacja ta zmniejsza wielkość rozwiązywanego problemu.
- Heurystyczna redukcja wierszy (ang. *heuristic row reduction*) – wykonanie wszystkich opisanych dotychczas operacji doprowadza do sytuacji, w której w macierzy *A* pozostają przypadki testowe gwarantujące znalezienie rozwiązania końcowego. Nie ma jednak gwarancji, że jest to rozwiązanie optymalne. Z tego powodu należy zastosować heurystykę, która pozwoli na ponowne efektywne wykorzystanie ww. operacji w celu redukcji rozwiązania. Autorzy metody *Moonlight* wyszli z założenia, że wydajnym wydaje się wybieranie wierszy z największym pokryciem, ponieważ pozwala to na ponowną redukcję problemu, poprzez eliminacje kolumn, które się z nimi przecinają.

Algorytm 1 MoonLight

```
function MOONLIGHT(A, X)
  if isEmpty(A) then
     return X
  end if
  Cols ← Singularities(A)
  if Cols ≠ 0 then
     A' ← RemoveRowsCols(A,0,Cols)
     return MOONLIGHT(A', X)
  end if
  Rows ← Exotic(A)
  if Rows ≠ 0 then
     Cols ← ContainedCols(A, Rows)
     A' ← RemoveRowsCols(A, Rows, Cols)
     X' ← X ∪ Rows
     return MOONLIGHT(A', X')
```

```
    end if
    (DomRows, SubRows) ← DominantRows (A)
    if DomRows ≠ 0 then
        Cols ← ContainedCols(A, DomRows)
        DelRows ← DomRows ∪ SubRows
        A' ← RemoveRowsCols(A, DelRows, Cols)
        X' ← X ∪ DomRows
        return MOONLIGHT(A' ,X')
    end if
    Cols ← DominantCols(A)
    if Cols ≠ 0 then
        A' ← RemoveRowsCols(A,0,Cols)
        return MOONLIGHT(A' , X)
    end if
        Rows ← Heuristic(A)
        Cols ← ContainedCols(A, Rows)
        A'← RemoveRowsCols(A, Rows, Cols)
        V'← V ∪ Rows
        return MOONLIGHT(A', X')
    end function
    Solution ← MOONLIGHT(A, 0)
```

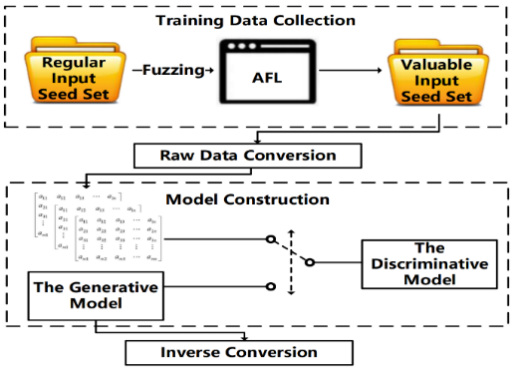
2.6.3.3 Destylator SmartSeed

Zaprezentowany w 2018 roku *SmartSeed* [21] to ciągle rozwijana metoda przygotowywania korpusu danych testowych, wykorzystująca algorytmy sztucznej inteligencji. Destylację wykonuje się w następujących etapach:

1. **Gromadzenie danych szkoleniowych:**
   Ustalane jest kryterium służące do ewaluacji wartości plików wejściowych i wybierana jest metoda uzyskania zestawu uczącego dla *SmartSeed*.
2. **Konwersja surowych danych:**
   W celu poprawnej interpretacji plików o niestałych formatach lub niestandardowych rozmiarach, dokonuje się ich konwersji do jednolitego typu macierzy.
3. **Konstrukcja modelu:**
   Interpretując macierze jako dane uczące, konstruowany jest model generatywny w oparciu o *Wasserstein Generative Adversarial Networks* [62].
4. **Konwersja odwrotna:**

Na podstawie modelu generatywnego generowane są macierze, konwertowane na odpowiednie pliki wejściowe, co jest procesem odwrotnym do kroku nr 2.

SmartSeed można łączyć z większością istniejących narzędzi do fuzzingu opartych na generatorach mutacyjnych. Domyślnie stosowany jest z *AFL*.



Rys. 10 Architektura Destylatora SmartSeed [21].

2.6.4 Zarządzanie korpusem w procesie fuzzingu

Utrzymanie odpowiedniej jakości danych testowych jest jedynym etapem w procesie fuzzingu, który łączy poszczególne iteracje automatycznych testów bezpieczeństwa. Dlatego zaleca się stosować procedurę zarządzania korpusem, która pozwala utrzymywać jego jakość na wysokim poziomie. Składa się ona z następującej listy kroków:

1. Zebranie początkowego korpusu (*crawling*, generowanie pseudolosowe, generowanie gramatyczne).
2. Przeprowadzenie fuzzingu.
3. Zebranie przypadków testowych wygenerowanych mutacyjne w procesie fuzzingu.
4. Deduplikacja ww. przypadków testowych.
5. Minimalizacja plików, destylacja całego korpusu.
6. Badanie pokrycia kodu przez zaktualizowany korpus.
7. Uzupełnianie brakującego pokrycia dodatkowymi przypadkami.
8. Przeprowadzenie fuzzingu i powrót do punktu nr 3.

Opisana powyżej procedura bazuje na fragmencie materiałów pochodzących z komercyjnych warsztatów „*Od zera do pierwszego 0–day'a. Warsztaty automatycznego wyszukiwania podatności*”, autorstwa Kamila Frankowicza[1].

[1] https://crossweb.pl/en/training/instytut-pwn-od-zera-do-pierwszego-0-daya-listopad-2019

2.7 Antyfuzzing

W związku z rosnącą popularnością wykorzystania fuzzerów w procesie wyszukiwania podatności, zaczęto stosować pierwsze mechanizmy blokujące ich stosowanie. Głównym celem antifuzzingu [63] jest utrudnianie atakującym skutecznej identyfikacji podatności, które mogłyby być wykorzystane w późniejszych atakach. Znajdowanie błędów bezpieczeństwa w programach implementujących to zabezpieczenie jest dalej możliwe, jednak wymaga od badaczy poświęcenia dłuższego czasu oraz zaangażowania większych zasobów sprzętowych. Głównym celem zastosowania tego typu mechanizmów jest umożliwienie autorom oprogramowania (lub osobom wykonującym jego zlecony audyt) znalezienie błędów bezpieczeństwa oraz wdrożenia niezbędnych poprawek, zanim zrobią to potencjalni atakujący. Najbardziej popularnymi metodami są pakowanie [64] i zaciemnianie [65] kodu źródłowego oraz wyzwalanie błędów w mechanizmach instrumentalizacji. Techniki te wprowadzają wyższe zapotrzebowanie na zasoby, co nie tylko utrudnia *fuzzing*, ale w efekcie może wpływać negatywnie na doświadczenia korzystania z oprogramowania przez zwykłych użytkowników.

Fuzzing może być skutecznie przerywany poprzez wyzwalanie błędów w dynamicznych narzędziach instrumentacyjnych. Zastosowanie tej techniki wymaga jednak dużej znajomości potencjalnie wykorzystywanego fuzzera, więc nie jest metodą ogólną.

W celu utrudnienia fuzzingu wzbogaca się chroniony program o dodatkowy moduł, którego uruchomienie przez użytkownika jest niezwykle trudne, jednak dla fuzzera (nierozumiejącego struktury programu) nie powinno stanowić problemu. Mechanizmy instrumentalizacyjne mogą „utknąć” w takim module, przeszukując jego przestrzeń stanów, co ostatecznie nie pozwoli na wyszukiwanie błędów w pozostałej, kluczowej z punktu widzenia bezpieczeństwa, części oprogramowania.

Zakłada się, że wszystkie spośród zastosowanych mechanizmów bezpieczeństwa nie powinny ostatecznie wpływać na wydajność chronionego oprogramowania. Sam moduł zabezpieczający powinien być trudny do odróżnienia od głównych fragmentów pliku wykonywalnego, aby jego blokada lub usunięcie było niemożliwe. Obecnie żadna z wyżej opisywanych metod nie jest w stanie spełnić wszystkich tych założeń.

2.8 Wady i zalety fuzzingu

Główną wadą stosowania fuzzingu w procesie wyszukiwania podatności jest to, że za jego pomocą znajduje się w większości nieskomplikowane do wywołania błędy. Jest to spowodowane tym, iż niemal wszystkie strategie fuzzingu starają się maksymalizować liczbę wywołań testowanego oprogramowania [66], promując w tym celu małe i niezłożone przypadki testowe.

Fuzzery wykorzystujące generatory pseudolosowe cechują się niskim pokryciem kodu. Jeżeli testowany program obsługuje pliki, które zawierają sumę kontrolną pozostałych danych, *fuzzer* w praktyce przetestuje wyłącznie moduł odpowiedzialny za jej wyliczanie.

Bardziej zaawansowane fuzzery, wykorzystujące instrumentalizacje do oceny jakości generowanych przypadków testowych, tylko pozornie zyskują przewagę nad klasycznymi technikami. Dopiero wykorzystanie reguł gramatycznych pozwala na znaczne zwiększenie pokrycia, jednak ich zastosowanie może ostatecznie doprowadzić do pominięcia części błędów.

Biorąc wyżej wymienione pod uwagę można wysnuć wniosek, iż losowe generowanie danych testowych w procesie fuzzingu jest istotną wadą, ponieważ wygenerowanie w ten sposób wartości brzegowej jest wysoce nieprawdopodobne. Efekt skali pozwala jednak na wygenerowanie przypadków testowych powodujących istotne błędy bezpieczeństwa, które mogą być skutecznie wykorzystywane w atakach komputerowych.

Ostatecznie stosowanie fuzzingu zdecydowanie zwiększa bezpieczeństwo oprogramowania, ponieważ pozwala na znalezienie błędów, których wyszukać nie byliby w stanie ani testerzy, ani osoby odpowiedzialne za audyt oprogramowania, ponieważ wymagałoby to od nich zastosowania metody zbliżonej do pełnego przeglądu wszystkich przypadków testowych. Zasadność stosowania fuzzingu jest na tyle istotna, iż duże korporacje, takie jak *Google* czy *Microsoft*, wprowadziły tę metodę w cykl wytwarzania oferowanego przez nie oprogramowania.

3. Algorytmy ewolucyjne

3.1 Historia algorytmów ewolucyjnych

Najwcześniejsze propozycje systemów inspirowanych ewolucją biologiczną zaprezentowane zostały w latach 60. XX wieku [67, 68, 69]. Głównym celem ww. badań było symulowanie zjawiska ewolucji biologicznej. Przez następne trzy dekady wyodrębniły się kluczowe kierunki obliczeń ewolucyjnych: programowanie ewolucyjne [70], strategie ewolucyjne [71] oraz algorytmy genetyczne [72].

Lata 90. XX wieku były okresem badań [73, 74] dotyczących istotności stosowania poszczególnych operatorów genetycznych, w celu wykluczenia pozostałych. Ostatecznie wnioski [75] z empirycznej komparacji operatorów mutacji i krzyżowania nie pozwoliły na wykluczenie tego pierwszego, którego zasadność stosowania była wcześniej podważana [76].

Aktualnie rozwój algorytmów ewolucyjnych i prowadzone nad nimi badania skupiają się na dostosowywaniu ogólnych metod ewolucyjnych do poszczególnych problemów obliczeniowych w celu ich sprawniejszego rozwiązywania [77, 78].

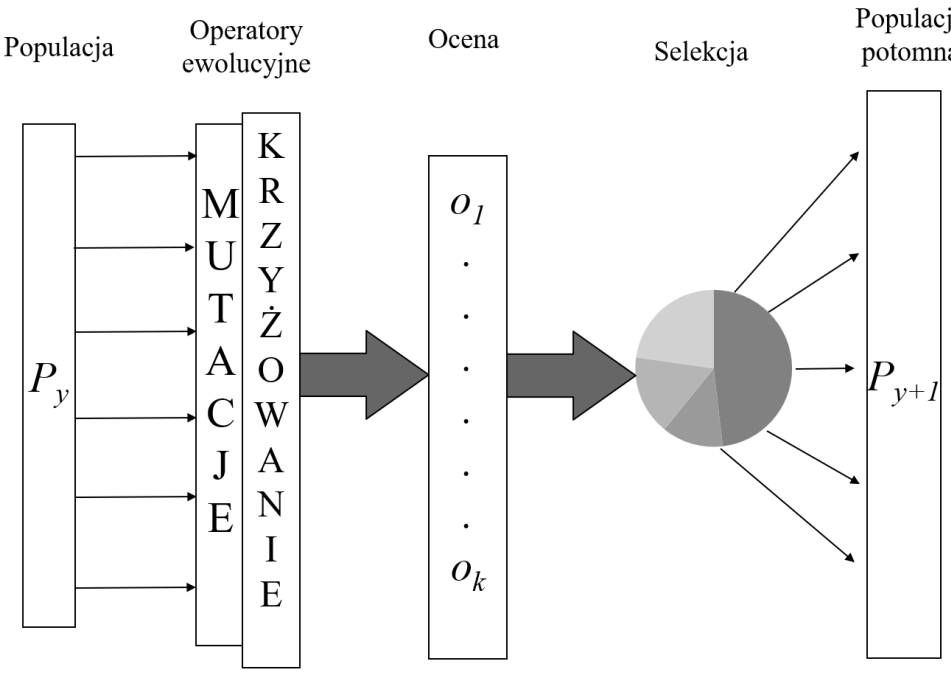
3.2 Algorytmy ewolucyjne

Algorytmy ewolucyjne, lub inaczej obliczenia ewolucyjne to rodzina algorytmów optymalizacji inspirowana ewolucją biologiczną. Pod względem technicznym są rodziną algorytmów populacyjnych służących do rozwiązywania problemów o charakterze stochastycznym lub metaheurystycznym.

W obliczeniach ewolucyjnych początkowy zestaw proponowanych rozwiązań jest generowany losowo, po czym aktualizowany iteracyjnie. Każda nowa generacja rozwiązań powstaje poprzez losowe usuwanie mniej pożądanych solucji i wprowadzanie niewielkich, losowych zmian. W terminologii biologicznej populacja rozwiązań poddawana jest więc „selekcji" i „mutacji" [79]. W rezultacie zbiór potencjalnych rozwiązań będzie stopniowo ewoluować, zwiększając swoje dopasowanie.

Ewolucyjne techniki obliczeniowe mogą tworzyć wysoce dopasowane rozwiązania dla szerokiego zakresu zadanych parametrów wybranych problemów. Istnieje wiele wariantów i rozszerzeń dostosowanych do bardziej szczegółowych rodzin problemów i struktur danych. Dodatkowo obliczenia ewolucyjne używane są w biologii ewolucyjnej jako eksperymenty „*in*

silico” [80] (pol. „w krzemie”, w analogii do określeń „*in vitro”*, pol. „w szkle” oraz „*in vivo”*, pol. „na żywym”) do badania ogólnych aspektów procesów ewolucyjnych.



Rys. 11 Schemat pojedynczego cyklu algorytmu ewolucyjnego.

3.3 Algorytmy genetyczne

Algorytmy genetyczne są najdłużej rozwijanym działem obliczeń ewolucyjnych. Zostały opracowane przez J. H. Hollanda [81] w latach 60–tych XX wieku, na potrzeby badań nad procesami adaptacyjnymi w odbiornikach sygnałów binarnych. Holland dopuszczał również wykorzystanie algorytmów genetycznych m. in. w celu optymalizacji funkcji wielu zmiennych [72], co w praktyce zostało zastosowane w 1975 r. przez K. A. De Jonga [82]. Zaprezentowana przez Hollanda idea została rozwinięta przez D. E. Goldberga w latach 80–tych XX wieku w celu optymalizacji pracy gazociągu [83].

Tym co wyróżnia algorytmy genetyczne od pozostałych metod symulowanej ewolucji, jest zastosowanie kodowania binarnego w celu reprezentacji chromosomów oraz odmiennego zastosowania poszczególnych operatorów ewolucyjnych (np. zwyczajowo stosowana jest selekcja rankingowa, która wyparła selekcję kołem ruletki).

3.4 Programowanie ewolucyjne

Programowanie ewolucyjne zostało zaproponowane przez L. J. Fogla i innych [84] w latach 60. XX wieku. Zaprezentowano mechanizm predykcji łańcuchów znaków generowanych procesami Markowa, bazujący na automatach skończonych (ang. *finite state machine, FSM*) [85]. Główną różnicą między programowaniem genetycznym, a algorytmami ewolucyjnymi i genetycznymi było zastosowanie wyłącznie operatora mutacji tak, aby „materiał genetyczny" nie był wymieniany między poszczególnymi osobnikami. Mutacja służyła do modyfikowania macierzy przejść, co było utożsamiane z wygenerowaniem nowego potomka. Z tak wygenerowanych osobników konstruowano łańcuchy znaków, które były nową populacją potomną. Wraz z populacją rodzicielską tworzyły tymczasową populację pośrednią. Połowa osobników z populacji pośredniej, cechujących się największymi wartościami funkcji przystosowania, tworzyła nową populację rodzicielską dla kolejnego pokolenia.

Programowanie ewolucyjne zostało rozwinięte przez syna L. J. Fogla, B. D. Fogla [86] poprzez umożliwienie ewolucji chromosomów, których allele miały przyporządkowane wartości rzeczywiste.

3.5 Kodowanie potencjalnych rozwiązań

Kodowanie w algorytmach ewolucyjnych wykorzystywane jest do zapisania potencjalnych rozwiązań zadania w formie chromosomów. Od wybranego kodowania zależy jakość otrzymanych wyników. Powinno ono być na tyle uniwersalne, aby nie faworyzować żadnego rozwiązania bądź grupy rozwiązań. Powszechnie stosuje się kodowanie: rzeczywiste, binarne oraz logarytmiczne. Jednak w zależności od wybranego algorytmu oraz klasy problemu możliwe jest odmienne interpretowanie wartości składających się na chromosom, np. jako poszczególne wierzchołki grafu w ścieżce [87].

3.5.1 Kodowanie binarne

Allele chromosomów kodowanych binarnie przyjmują wartości zapisane w systemie dwójkowym $\{0,1\}$. W celu zakodowania liczb rzeczywistych dla algorytmów genetycznych [88] wykorzystuje się liniowe odwzorowanie $[v_L, v_R]$ na przedział $[0, 2^{\xi} - 1]$ ($2^{\xi} - 1$ jest liczbą dziesiętną zakodowaną w postaci ciągu binarnego o długości ξ) w następujący sposób:

$$v_B = v_L + v_D * \frac{v_R - v_I}{2^{\xi} - 1} \qquad (1)$$

gdzie:

v_R – wartość rzeczywista prawego końca przedziału liczbowego;

v_L – wartość rzeczywista lewego końca przedziału liczbowego;

v_B – wartość zakodowanej liczby rzeczywistej;

v_D – dziesiętna wartość ciągu binarnego stanowiącego zakodowaną postać liczby v_B.

Długość ξ liczby $2^{\xi} - 1$ wyznaczana jest zaś z nierówności:

$$(v_R - v_L) * 10^q \leq 2^{\xi} - 1 \qquad (2)$$

gdzie:

q – liczba miejsc po przecinku (przyjęta dokładność obliczeń).

3.5.2 Kodowanie rzeczywiste

Kodowanie rzeczywiste [89] jest wykorzystywane najczęściej w strategiach ewolucyjnych, gdzie chromosomy składają się z liczb zmiennoprzecinkowych, których precyzja jest limitowana przez wykorzystywany w obliczeniach język programowania. Podejście to ułatwia kodowanie problemu i poprawia dokładność końcowego rozwiązania.

3.5.3 Kodowanie logarytmiczne

Kodowanie logarytmiczne [90] wykorzystuje się w celu skrócenia chromosomów kodowanych w notacji binarnej za pomocą funkcji ekwipotencjalnej. Stosowane jest najczęściej przy rozwiązywaniu problemów wielokryterialnych, charakteryzujących się dużą przestrzenią potencjalnych rozwiązań. Ciąg binarny kodowany logarytmiczne podzielony jest na trzy części [α β bin] będący reprezentacją liczby:

$$(-1)^{\beta} e^{(-1)^{\alpha}[bin]_{10}} \qquad (3)$$

gdzie:

$[bin]_{10}$ – wartość dziesiętną wykładnika;

α – bit znaku funkcji wykładniczej;

β – bit znaku wykładnika funkcji wykładniczej;

bin – pozostałe bity będące wartością wykładnika funkcji wykładniczej.

3.5.4 Schematy

Zaproponowana przez J. H. Hollanda teoria schematów [91] była próbą uzasadnienia skuteczności algorytmów genetycznych. Schemat jest wzorcem opisującym podzbiór ciągów analogicznych ze względu na wyznaczone pozycje. Schematy definiuje się za pomocą alfabetu {0,1,*}. Symbol „*” reprezentuje wartość nieokreśloną (0 lub 1). Chromosom ch należy do schematu H wtedy i tylko wtedy, jeśli każdy symbol w ch odpowiada symbolowi określonemu w H.

Twierdzenie o schematach sformułowane przez J. H. Hollanda [91]:

Jeżeli schemat odpowiada rozwiązaniom przeciętnie lepszym niż inne, to osobniki do niego pasujące stanowią coraz liczniejszą grupę wraz ze wzrostem liczby pokoleń symulowanej ewolucji. Wzrost ich liczebności jest szacunkowo wykładniczy.

Badania prowadzone nad schematami wykazały istnienie zjawiska ukrytej równoległości. Zjawiskom takim jak tworzenie nowych osobników, modyfikacja, przenoszenie do następnego pokolenia, ocenie pod kątem dostosowania podlegają pojedyncze osobniki, jak i całe schematy. W związku z tym, że każdy osobnik może zostać przystosowany do wielu schematów, ostatecznie algorytm genetyczny przetwarza większą liczbę tych drugich [83, 92].

Rzędem schematu $\gamma(H)$ określamy liczbę ustalonych w nim pozycji (znaków niebędących „*”).

Rozpiętość schematu $\delta(H)$ rozumiana jest jako odległość między dwoma skrajnie położonymi, ustalonymi miejscami.

Wykorzystanie powyższych sformułowań umożliwia zdefiniować prawdopodobieństwo $p(H)$, iż schemat pozostanie niezmieniony w następstwie krzyżowania i mutacji:

$$p(H) \approx 1 - p_c * \frac{\delta(H)}{\iota - 1} - \gamma(H) * p_a \qquad (4)$$

gdzie:

p_c – jest prawdopodobieństwem krzyżowania pojedynczego osobnika;

p_a – jest prawdopodobieństwem mutacji pojedynczego allelu;

ι – jest długością chromosomu ch.

Wyżej opisane prawdopodobieństwo posłuży do oszacowania średniej liczby osobników $m(H,t)$, które w chwili t pasują do schematu H.

$\phi(H)$ – średnia wartość funkcji celu osobników pasujących do H;

$\bar{\phi}$ – średnia wartość funkcji celu w populacji.

Wtedy:

$$m(H,t+1) \geq m(H,t) * \frac{\phi(H)}{\bar{\phi}} * p(H) \qquad (5)$$

Schematy mogą odzwierciedlać pewne, ewentualnie „korzystne", cechy w przestrzeni potencjalnych rozwiązań. Twierdzenie o schematach pozwala na zauważenie, iż liczebność chromosomów pasujących do tych schematów rośnie z czasem w populacji. Implikuje to wzrost prawdopodobieństwa, że dla stałej populacji chromosom będzie pasował do większej liczby „korzystnych" schematów. Zjawisko to przekłada się bezpośrednio na wzrost szansy na wygenerowanie chromosomu reprezentującego rozwiązanie optymalne. Zakłada się [92, 93], że algorytmy genetyczne działają najskuteczniej, kiedy kodowanie problemu oraz funkcja celu umożliwiają zdefiniowanie krótkich i zwartych schematów, tzw. cegiełek (lub inaczej – bloków budujących).

3.6 Operatory genetyczne

3.6.1 Mutacje

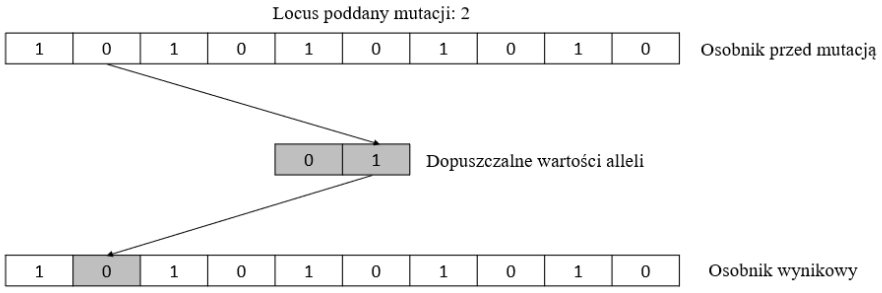
W naukach biologicznych mutacje rozumie się jako spontaniczne, skokowe modyfikacje materiału genetycznego, które mogą zostać odziedziczone w organizmach potomnych [94]. Pierwotnie definiowane jako zjawisko spontaniczne, obecnie wiadomo również, że wywołać je może wpływ czynników zewnętrznych, nazywanych mutagenami [95], takich jak promieniowanie jonizujące, elektromagnetyczne, ultrafioletowe czy niektóre związki chemiczne.

W algorytmach genetycznych głównym celem mutacji jest odtworzenie potencjalnie zniszczonego genu, który został utracony w wyniku działania innego operatora – krzyżowania. Mutacja zapobiegać ma również przedwczesnej zbieżności populacji. W związku z tym zakłada się, że mutacja nie bierze znaczącego udziału w procesie

generowania rozwiązania optymalnego (prawdopodobieństwo wystąpienia mutacji przyjmuje się zazwyczaj na poziomie $p_m < 0{,}2$) [96].

3.6.1.1 Mutacja jednopunktowa

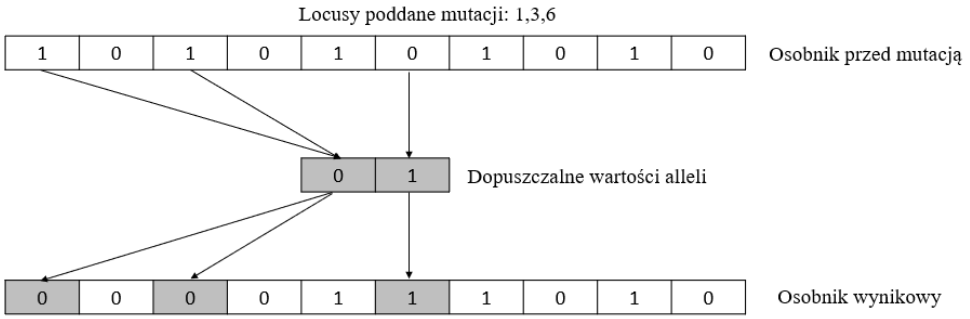
Mutacja jednopunktowa jest podstawową metodą losowej modyfikacji genotypu osobnika. W algorytmach genetycznych zazwyczaj losuje się gen, który jest następnie negowany. Czasami spotyka się rozwiązanie, które polega na wylosowaniu dwóch genów, zamienianych następnie miejscami w chromosomie [91].



Rys. 12 Schemat mutacji jednopunktowej.

3.6.1.2 Mutacja wielopunktowa

Wykonanie mutacji wielopunktowej co do zasady polega na tym samym co realizacja mutacji jednopunktowej z tą różnicą, że w celu jej zastosowania losuje się większą liczbę locusów poddawanych modyfikacji [90].



Rys. 13 Schemat mutacji wielopunktowej.

3.6.1.3 Pozostałe metody mutacji

Wyżej wymienione metody mutacji ze względu na swoją uniwersalność są najczęściej wykorzystywane w algorytmach ewolucyjnych. W literaturze można spotkać także przykłady następujących mutacji, których zastosowanie zależy od rozwiązywanego problemu oraz wybranej metody kodowania potencjalnych rozwiązań.

Mutacja jednolita – w mutacji jednolitej nowa wartość genu jest losowana ze zbioru liczb o rozkładzie normalnym [97].

Mutacja graniczna – locus poddany mutacji zamieniony jest na brzegową wartość genu. Mutacja stosowana jest w przypadkach, gdy spodziewa się rozwiązania bliskiemu wartościom brzegowym [98].

Mutacja gaussowska – polega na dodaniu losowej wartości z rozkładu Gaussa do każdego genu chromosomu w celu stworzenia nowego osobnika [99].

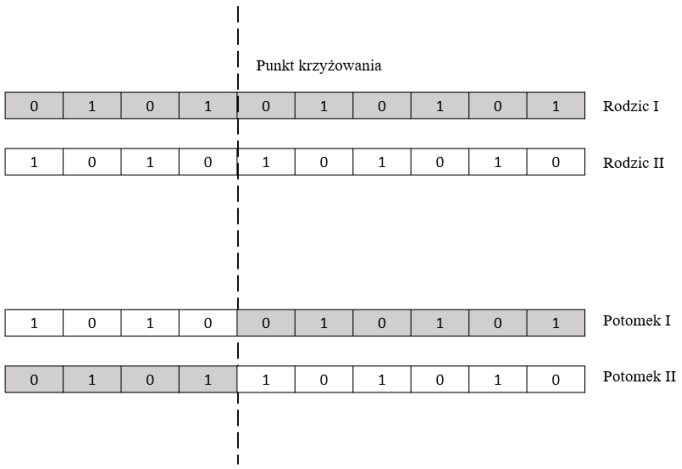
Mutacja zmieniająca – mutacja polegająca na zamianie dwóch wylosowanych genów locusami [100].

3.6.2 Krzyżowanie

Krzyżowanie, inaczej hybrydyzacja, w naukach biologicznych jest procesem, w którym na bazie dwóch organizmów o różnych genotypach powstaje osobnik potomny [101]. W algorytmach symulowanej ewolucji celem krzyżowania jest otrzymanie osobnika, który reprezentuje cechy osobników rodzicielskich (w następstwie wymiany informacji genetycznej). Prawdopodobieństwo krzyżowania zazwyczaj przyjmuje wartości z przedziału $0{,}5 < p_c < 1$ [102, 103].

3.6.2.1 Krzyżowanie jednopunktowe

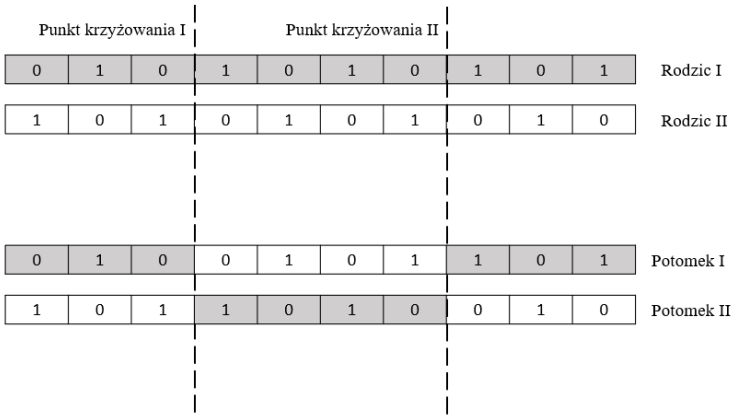
Krzyżowanie jednopunktowe polega na „rozcięciu” w tym samym miejscu dwóch chromosomów i połączenie „lewej” części pierwszego rodzica z „prawą” częścią drugiego rodzica i odwrotnie. W następstwie tej operacji powstają dwa osobniki potomne [104].



Rys. 14 Schemat krzyżowania jednopunktowego.

3.6.2.2 Krzyżowanie wielopunktowe

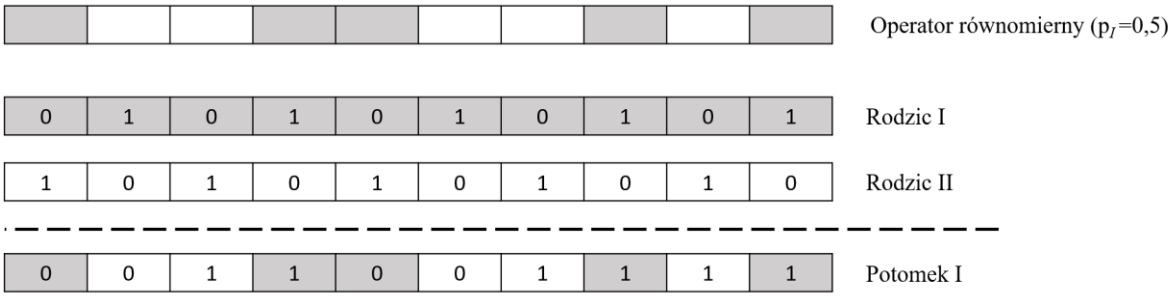
Krzyżowanie wielopunktowe [105] polega na „rozcięciu" dwóch chromosomów w wielu punktach krzyżowania. Chromosomy potomne dziedziczyć będą geny przedziałami naprzemiennie od każdego z rodziców.



Rys. 15 Schemat krzyżowania wielopunktowego (dwupunktowego).

3.6.2.3 Krzyżowanie równomierne

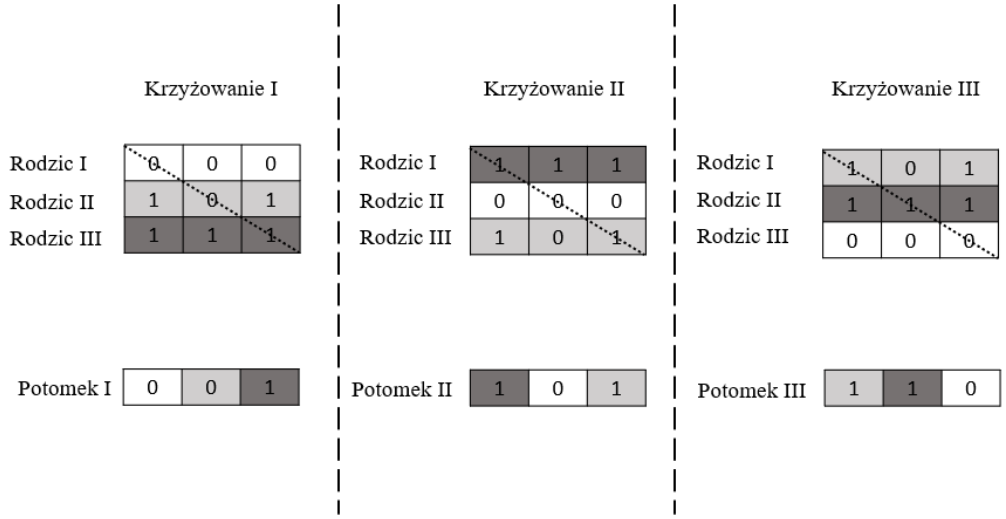
Krzyżowanie równomierne jest uogólnieniem krzyżowania wielopunktowego. Geny są dziedziczone przez potomka w sposób losowy z prawdopodobieństwem p_I od rodzica pierwszego, w przeciwnym razie od rodzica drugiego [106].



Rys. 16 Schemat krzyżowania równomiernego.

3.6.2.4 Krzyżowanie po przekątnej

Krzyżowanie ukośne [107] jako jedno z nielicznych nie jest wzorowane na procesie biologicznej hybrydyzacji. Odbywa się ono między większą niż dwa liczbą osobników. Chromosom potomny dziedziczy geny znajdujące się na przekątnej macierzy ułożonej z osobników populacji rodzicielskiej.



Rys. 17 Schemat krzyżowania po przekątnej.

3.6.2.5 Krzyżowanie rozmyte

Krzyżowanie rozmyte, zaproponowane w 2012 roku [108], uzależnia wybraną w algorytmie metodę hybrydyzacji do stopnia zróżnicowania populacji, stając się automatycznie swoistą metodą sterowania zbieżnością. W przypadku wysokiej różnorodności stosowane jest krzyżowanie dwupunktowe. Kiedy różnorodność populacji spada poniżej

ustalonego poziomu, dotychczasowa metoda krzyżowania zastępowana jest hybrydyzacją wielopunktową lub równomierną.

3.6.2.6 Krzyżowanie heurystyczne

Krzyżowanie heurystyczne [109] polega na założeniu, że pierwszy ch′ z chromosomów pary rodzicielskiej posiada wszystkie wartości alleli większe od tych, z których składa się drugi z rodziców ch′′. Wartości poszczególnych genów osobnika potomnego wyliczane są wg wzoru:

$$ch_y^S = \theta * \left(ch'_y - ch''_y\right) + ch'_y \qquad (6)$$

gdzie:

ch^s– osobnik potomny;

θ – zmienna przyjmująca wartość [0,1];

y – indeksy poszczególnych locusów.

3.7 Metody selekcji

3.7.1 Selekcja rankingowa

Metoda rangowa polega na sortowaniu populacji rodzicielskiej pod względem osiąganej przez poszczególne osobniki wartości funkcji oceny, w celu ustanowenia listy rankingowej. Każdemu z potencjalnych rozwiązań przypisywana jest ranga. Osobnikowi najbardziej dostosowanemu (posiadający największą wartość funkcji oceny) przypisuje się najwyższa rangę, a najgorzej przystosowany otrzymuje rangę najniższą. Minimalna wartość rangi wynosi 1, największa równa się liczebności rozpatrywanej populacji. Prawdopodobieństwo przejścia selekcji rankingowej osobnika może bazować na funkcji liniowej lub nieliniowej [110].

Funkcja liniowa:

$$R\left(\zeta, S_p, B\right) = 2 - S_p + \frac{2*\left(S_p - 1\right)*(\zeta - 1)}{B-1} \qquad (7)$$

Funkcja nieliniowa:

$$R(\zeta, Q, B) = \frac{B * Q^{(\zeta-1)}}{\sum_{b=1}^{B} Q^{(b-1)}} \qquad (8)$$

gdzie:

ζ– miejsce osobnika na liście rankingowej;

S_p – presja selekcyjna;

b – indeks poszczególnych osobników w populacji rodzicielskiej.

B – liczba wybieranych osobników;

Q – współczynnik selekcji, ustalany eksperymentalnie.

Wartość funkcji prawdopodobieństwa R wyznaczana jest dla każdego osobnika.

Znormalizowana wartość funkcji R:

$$R'_h = \frac{R_h}{\sum_{b=1}^{B} R_b}, h = \{1,2,..,B\} \qquad (9)$$

gdzie:

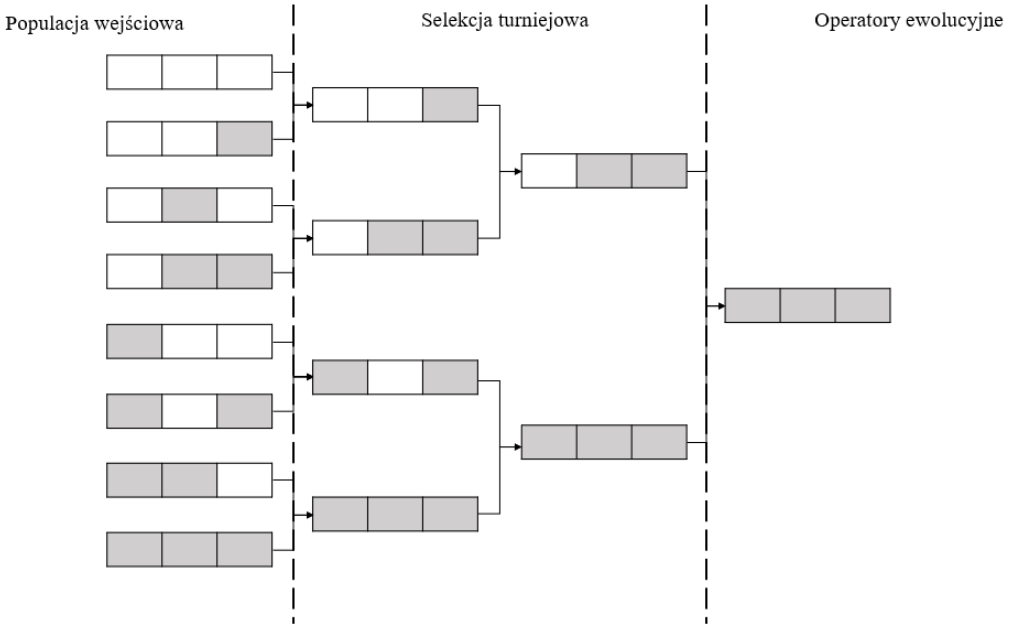
h – indeks „normalizowanego" osobnika populacji rodzicielskiej.

b – indeks poszczególnych osobników w populacji rodzicielskiej.

Wykorzystując ww. znormalizowaną wartość funkcji R określa się prawdopodobieństwo przeżycia osobnika z populacji rodzicielskiej. Chromosomy tworzące nowe pokolenie wybierane są poprzez wylosowanie, zgodnie z rozkładem równomiernym B liczb rzeczywistych z zakresu [0,1], które wyznaczają osobniki w nowej populacji.

3.7.2 Selekcja Turniejowa

Selekcja turniejowa [111] jest rodzajem selekcji nadającym się do poszukiwania rozwiązań zarówno minimalnych jak i maksymalnych. Proces selekcji zaczyna się od określenia rozmiaru turnieju. Do każdego etapu selekcji wyznacza się ustaloną liczbę osobników. Każdy etap (pojedynek) wygrywa jeden z chromosomów. Zwycięzcy pojedynków rywalizują następnie między sobą w kolejnych „potyczkach", aż do wyłonienia zwycięzcy. Triumfujący chromosom poddawany jest następnie operatorom mutacji i krzyżowania. W przypadku drugiego z operatorów, jeżeli będzie on stosowany, odbywa się kolejny turniej w celu wyłonienia partnera. Liczba odbywanych turniejów odpowiada ustalonej liczbie populacji.



Rys. 18 Przykładowy schemat selekcji turniejowej.

3.7.3 Selekcja kołem ruletki

Selekcja proporcjonalna kołem ruletki wykorzystuje populację pośrednią, równoliczną populacji rodzicielskiej [112]. Każdemu chromosomowi przyporządkowuje wycinek koła według wzoru:

$$w(ch_g) = \frac{o_k(ch_g)}{\sum_{g'=1}^{G} o_k(ch_{g'})} * 100\%, g, g' = \{1,2, \dots G\} \qquad (10)$$

gdzie:

G – liczebność populacji pośredniej;

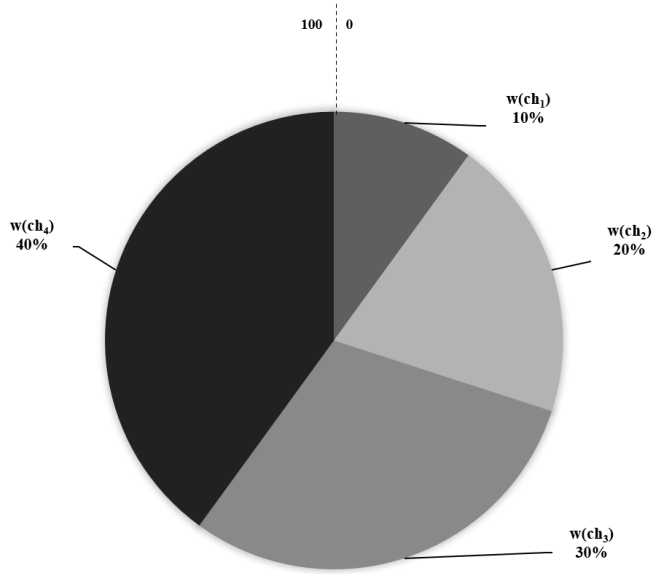
ch_g – g–ty chromosom należący do populacji pośredniej;

$w(ch_g)$ – wycinek koła wyrażany w %;

$o_k()$ – funkcja przystosowania.

Następnie losowane jest G liczb z przedziału [0, 100], z wykorzystaniem których wybierane są chromosomy przechodzące proces selekcji. W ten sposób istnieje możliwość wielokrotnego „rozmnożenia się" lepiej dostosowanych osobników, co uważane jest za wadę selekcji kołem ruletki, ponieważ powoduje przedwczesną zbieżność algorytmu oraz stagnację (sytuację gdy w końcowej fazie pracy algorytmu średnie przystosowanie niewiele różni się od maksymalnego proponowanego przez algorytm). Jedną z większych zalet tej metody jest fakt,

iż pozwala na „rozmnażanie się” osobników słabiej czy nawet najgorzej przystosowanych. Metoda ta stosowana jest przy dodatnich wartościach funkcji przystosowania oraz wyłącznie w zadaniach maksymalizacji.



Rys. 19 Przykładowy schemat selekcji kołem ruletki.

3.7.4 Selekcja metodami (μ, λ), (μ+ λ)

Metody selekcji (μ, λ) oraz $(\mu + \lambda)$ [113] wykorzystywane są w strategiach ewolucyjnych [90]. W obydwóch strategiach μ oznacza liczebność populacji rodzicielskiej a λ liczbę potomków powstałych w wyniku krzyżowania pary rodzicielskiej (zazwyczaj $\lambda \geq 2$). W strategii (μ, λ) liczebność populacji pośredniej podlegającej selekcji wynosi $\mu * \lambda$. Strategia $(\mu + \lambda)$ zakłada powiększenie populacji pośredniej o dotychczasową populację rodzicielską, dlatego też liczebność populacji pośredniej wyniesie $\mu * (1 + \lambda)$. Selekcja polega na wyborze μ najlepiej dostosowanych osobników z populacji pośredniej. W strategii (μ, λ) osobniki przeżywają tylko w jednym pokoleniu (generacji), gdzie w metodzie $(\mu + \lambda)$ mogą one przetrwać wielokrotnie proces selekcji.

3.8 Sterowanie zbieżnością algorytmu

Szybkość zbieżności [114] jest to liczba pokoleń, po których wygenerowaniu wartość najlepiej przystosowanego osobnika nie zmienia się. Można ją utożsamiać z czasem niezbędnym do wskazania przez algorytm genetyczny najlepszego rozwiązania.

Zbieżność wyników osiąganych przez algorytm jest następstwem zdominowania populacji przez niewielką liczbę genotypów. W efekcie działanie operatora krzyżowania przestaje być skuteczne. Biorąc pod uwagę fakt, iż działanie algorytmów ewolucyjnych opiera się na przeszukiwaniu przestrzeni potencjalnych rozwiązań, osiągniecie szybkiej zbieżności jest zjawiskiem niepożądanym, ponieważ znacząco tą przestrzeń ogranicza. Zbyt wolne osiągnięcie zbieżności wydłuża jednak czas poszukiwania rozwiązania, co czyni wykorzystywanie algorytmów ewolucyjnych mniej zasadnym. Sterowanie zbieżnością odbywa się za pomocą doboru parametrów pracy algorytmu ewolucyjnego, takich jak prawdopodobieństwa mutacji, krzyżowania czy wielkość populacji. Dopuszcza się również stosowanie technik dodatkowych takich jak liniowe skalowanie funkcji przystosowania [115] czy skalowanie Boltzmana [116].

3.9 Warunki stopu algorytmów ewolucyjnych

Warunki zatrzymania algorytmów ewolucyjnych zależne są od rodzaju rozwiązywanego zadania oraz dopuszczalnego odchylenia od rozwiązania optymalnego [117]. Prawidłowe ustalenie warunku stopu zapobiega wykonywaniu nadmiarowych obliczeń oraz wpływa na dokładność wyznaczonego rozwiązania.

Jednym z warunków stopu dla algorytmów ewolucyjnych jest określona „*a priori”* liczba generacji, w których nie uległo poprawie najlepsze rozwiązanie wskazane przez algorytm. Warunek ten jest zazwyczaj łączony z zadanym limitem czasu obliczeń, który ustanawia maksymalny okres poszukiwania rozwiązania optymalnego.

Odmiennym rozwiązaniem jest mierzenie odchylenia standardowego funkcji oceny, utożsamianego z różnorodnością wygenerowanych rozwiązań. Algorytm kończy obliczenia w momencie, kiedy pozostałe w populacji osobniki cechują się zbliżonym do siebie genotypem. Analogicznymi warunkami zakończenia poszukiwania rozwiązania najlepszego jest wartość wariancji lub poziomu ufności.

3.10 Dostrajanie algorytmu ewolucyjnego

Dostrajanie algorytmu ewolucyjnego [118, 119] jest procesem ustalania parametrów jego pracy tak, aby przeszukiwanie przestrzeni potencjalnych rozwiązań było najbardziej efektywne dla zadanego problemu. Przekłada się on bezpośrednio na jakość otrzymanego wyniku. Głównymi parametrami pracy algorytmów ewolucyjnych są:

- Długość osobnika – liczba genów składająca się na chromosom, który go reprezentuje. Zależna jest od danych wejściowych rozwiązywanego problemu oraz zastosowanego sposobu kodowania. Niepoprawny sposób doboru kodowania osłabia skuteczność operatorów ewolucyjnych. Kodowanie powinno być zgodne z twierdzeniem o schematach.
- Rozmiar populacji – liczba mająca znaczący wpływ na jakość i czas uzyskania rozwiązania. Zbyt mała populacja może utrudnić bądź nawet uniemożliwić znalezienie akceptowalnego wyniku. Zbyt duża liczba osobników w populacji może doprowadzić do utraty zbieżności i przekształcenia algorytmu genetycznego w algorytm losowego przeszukiwania wszystkich rozwiązań.
- Wybór poszczególnych operatorów genetycznych i prawdopodobieństwo ich wystąpienia.

Wszystkie wyżej wymienione parametry najczęściej ustalane są w drodze eksperymentów, polegających na empirycznej weryfikacji poszczególnych kombinacji właściwości pracy algorytmu.

3.11 Kierunki rozwoju obliczeń ewolucyjnych

Obecnie badania dotyczące algorytmów ewolucyjnych skupiają się na zagadnieniach związanych m.in. z automatyzacją dostrajania, np. poprzez dobór prawdopodobieństw krzyżowania i mutacji dla zadanego problemu [120, 121].

Kolejnymi z kierunków badań nad symulowaną ewolucją są metody redukcji problemu przedwczesnej zbieżności. W tym celu opracowywane są nowe odmiany znanych operatorów ewolucyjnych [122].

Złożone problemy wymagają zaawansowanego mapowania fenotypu na genotyp opisywany przez chromosom. W takich przypadkach trudnością może się okazać wyznaczenie schematów czy bloków budujących. Niezbędnym wydaje się więc opracowywanie metod kodowania dostosowanych do poszczególnych problemów optymalizacyjnych, aby było to możliwe [123, 124].

Biorąc pod uwagę fakt, iż symulowana ewolucja bazuje na mechanizmach zaczerpniętych z ewolucji biologicznej, cześć prac skupia się na dalszych próbach implementacji innych zjawisk biologicznych, takich jak mutacje wirusów [125], praca układu odpornościowego [126] czy zjawiska epigenetyczne [23].

4. Zjawiska epigenetyczne

4.1 Wprowadzenie

Epigenetykę [127], czyli dziedziczenie pozagenowe zdefiniował w 1942 roku C. H. Waddington, w celu wyjaśnienia mechanizmu oddziaływania pomiędzy genami oraz ich produktami w procesie powstawaniu fenotypu. Opisuje ona zjawisko, w którym ekspresja genów zależy od czynników zewnętrznych. Poprzez ekspresje rozumie się fakt przepisania informacji genetycznej z DNA (kwasu deoksyrybonukleinowego) na RNA (kwasy rybonukleinowe) i poszczególne białka. Zjawisko to ma wpływ na ostateczną postać fenotypu, bez wpływu na informację niesioną w DNA.

Epigenomem określa się informację genetyczną, która jest dziedziczona w trakcie podziałów komórkowych, lecz nie wynika ona z samej sekwencji DNA. Na jej postać wpływ mają wszystkie znaczniki molekularne wyciszające bądź aktywujące wybrane fragmenty DNA. Wartym podkreślenia jest fakt, iż informacja ta jest elastyczna i może być dynamicznie zmieniana. Epigenom podatny jest na modyfikację w odpowiedzi na zmiany środowiskowe.

W naukach biologicznych zmiany epigenetyczne obserwowane są najczęściej w rozwoju bliźniąt monozygotycznych (jednojajowych). Pomimo posiadania tego samego DNA, osobniki te posiadają znaczne różnice w fenotypie, pogłębiające się wraz z starzeniem się organizmu.

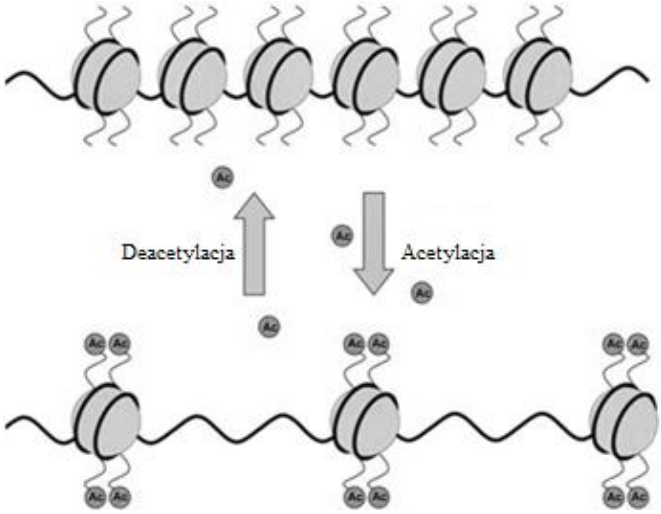
4.2 Wybrane zjawiska epigenetyczne

4.2.1 Metylacja DNA

Metylacja DNA [128] jest modyfikacją, która polega na dodaniu do zasad azotowych nukleotydów grupy metylowej. Następstwem tego zjawiska jest wycieszenie lub ograniczenie ekspresji genów. Ekspresja poszczególnych genów jest skorelowana z poziomem zmetylowania DNA. Wyższy stopień metylacji obniża ekspresję danego genu. Zaburzenia metylacji mogą prowadzić do zmian nowotworowych np. poprzez wyłączenie ekspresji genów supersorowych [129]. Zjawisko metylacji może być dziedziczone w komórkach potomnych.

4.2.2 Modyfikacje histonów

DNA umiejscowione jest w kompleksie makromolekularnym – chromatynie, która składa się z białek histonowych oraz niehistonowych. Główną rolą histonów [130] jest wiązanie DNA. Białka te podlegają różnorodnym modyfikacjom, wśród których najlepiej poznana jest acetylacja histonów. Odpowiada ona za rozluźnienie chromatyny, a w następstwie – zwiększenie ekspresji genów. Odwrotnością tego procesu jest deacetylacja histonów, której skutkiem jest hamowanie transkrypcji genów (wyciszenie).



Rys. 20 Schemat procesów acetylacji i deacetylacji histonów [131].

4.3 Znaczenie w procesie selekcji naturalnej

Badania prowadzone nad gatunkiem nicieni *C. elegans* [132] dowiodły, że modyfikacja czynników zewnętrznych, takich ilość i rodzaj dostępnego pożywienia czy temperatura otoczenia, powoduje znaczną zmianę w ekspresji ich genów. Cechy te mogą zostać przekazane międzypokoleniowo i utrzymywać się nawet przez 14 generacji. W przypadku, kiedy żyjące w tym samym środowisku co ich przodkowie osobniki zostaną poddane analogicznym czynnikom stresogennym, ich fenotyp bazujący na modyfikacjach genetycznych zwiększy znacząco szanse na przetrwanie. Z drugiej strony wywołane zmiany w budowie organizmu (np. większe nagromadzenie tkanki tłuszczowej), mogą doprowadzić do wystąpienia chorób, co zmniejsza szanse tych osobników na posiadanie potomstwa.

4.4 Odwzorowanie zjawisk epigenetycznych w algorytmach ewolucyjnych

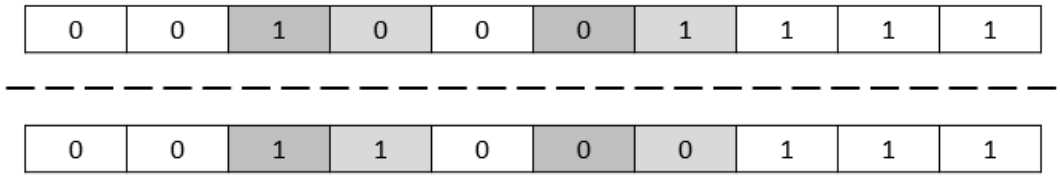
4.4.1 Odwzorowanie imprintingu

W naukach biologicznych *imprinting* genomowy polega na stopniowanej metylacji genów oraz histonów w komórkach – gametach (komórkach rozrodczych) [133]. Do zjawiska tego dochodzi w podczas gametogenezy (łączenia się komórek rozrodczych). Głównym zadaniem tego zjawiska jest zapobiegnięcie całkowitej dominacji genów jednego z rodziców, co mogłoby doprowadzić do wystąpienia partenogenezy, w efekcie czego zmniejszyłaby się bioróżnorodność populacji.

W algorytmach genetycznych *imprinting* genomowy jest operatorem ograniczającym ekspresje genów jednego z rodziców. Występuje na etapie krzyżowania. Może być utożsamiany z dynamiczną zmianą punktów krzyżowania tak, aby udział genów jednego z rodziców miał większy wpływ na kształt potomstwa.

4.4.2 Odwzorowanie paramutacji

Paramutacja jest zjawiskiem epigenetycznym, polegającym na interakcji między dwoma allelami w jednym locusie. Skutkuje powstaniem w jednym z nich zmian dziedzicznych. Zmiana może dotyczyć wzorca metylacji DNA lub modyfikacji histonów [134]. Allel inicjujący zmianę jest określany mianem allelu paramutagennego, podczas gdy allel, który został zmieniony epigenetycznie, jest określany jako allel paramutowalny [135]. Może on mieć zmienione poziomy ekspresji poszczególnych genów, które potrafią być przekazane potomstwu, nawet jeśli allel paramutagenny sam nie został przekazany.



Rys. 21 Schemat działania operatora bazującego na zjawisku paramutacji.

4.4.3 Odwzorowanie bookmarkingu

Bookmarking jest mechanizmem pamięci epigenetycznej, który polega na przekazywaniu w czasie mitozy informacji o genach aktywnych oraz tych, które mogą zostać potencjalnie aktywowane w komórkach potomnych [136]. *Bookmarking* gwarantuje zatem, że komórki

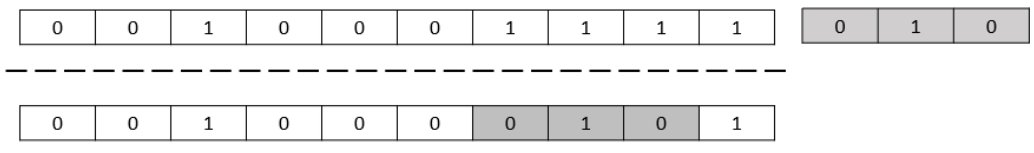
potomne mają taki sam wzór ekspresji genów jak komórka, z której pochodzą. Stosowany jest w połączeniu z innymi operatorami epigenetycznymi.

4.4.4 Odwzorowanie dziedziczenia za pomocą prionu

W rozprawie [23] autorstwa Kornela Chromińskiego zaproponowany został operator genetyczny inspirowany białkowymi cząstkami zakaźnymi, tzw. prionami (*PrPsc*). Jest to zmodyfikowana forma białka komórkowego (*PrPc*), kodowanego przez genom. Modyfikacja białek nieaktywnych w infekcyjne następuje pod wpływem czynników zewnętrznych. Zmodyfikowane białka zakaźne cechują się dużą odporności na działanie czynników fizycznych i chemicznych. Kontakt z białkiem pochodzącym od innego osobnika (np. poprzez drogę pokarmową) może prowadzić do wprowadzenia prionu do organizmu. W następstwie kontaktu z białkami zakaźnymi w błonach komórkowych w ciele gospodarza powstaje heterodimer (*PrPc–PrPsc*), przekształcający się następnie w homodimer (*PrPsc–PrPsc*). Cząstka ta rozpada się ostatecznie na dwie nowe cząstki prionowe (*PrPsc*). Jest to więc pozagenetyczna metoda na zmianę struktury białek, która prowadzi m.in. do chorób nazywanych pasażowalnymi enefalopatioami gąbczastymi [137].

Modyfikator epigenetyczny wzorowany na ww. cząstkach zakaźnych implementowany jest następująco: chromosomy w kolejnych iteracjach wystawiane są na oddziaływanie czynnika zewnętrznego w postaci krótkiego fragmentu losowego genotypu. Role czynnika zewnętrznego pełni określonej długości wektor binarny (symulacja prionu inwazyjnego), odpowiadający kodowaniu osobników w algorytmie genetycznym. Działanie prionu powoduje wystąpienie jednakowej zmiany w kodzie wszystkich osobników, które zostały w tym celu wylosowane przez algorytm.

W zaproponowanym rozwiązaniu zmianom epigenetycznym nie podlegają osobniki posiadające najwyższą wartość funkcji przystosowania.



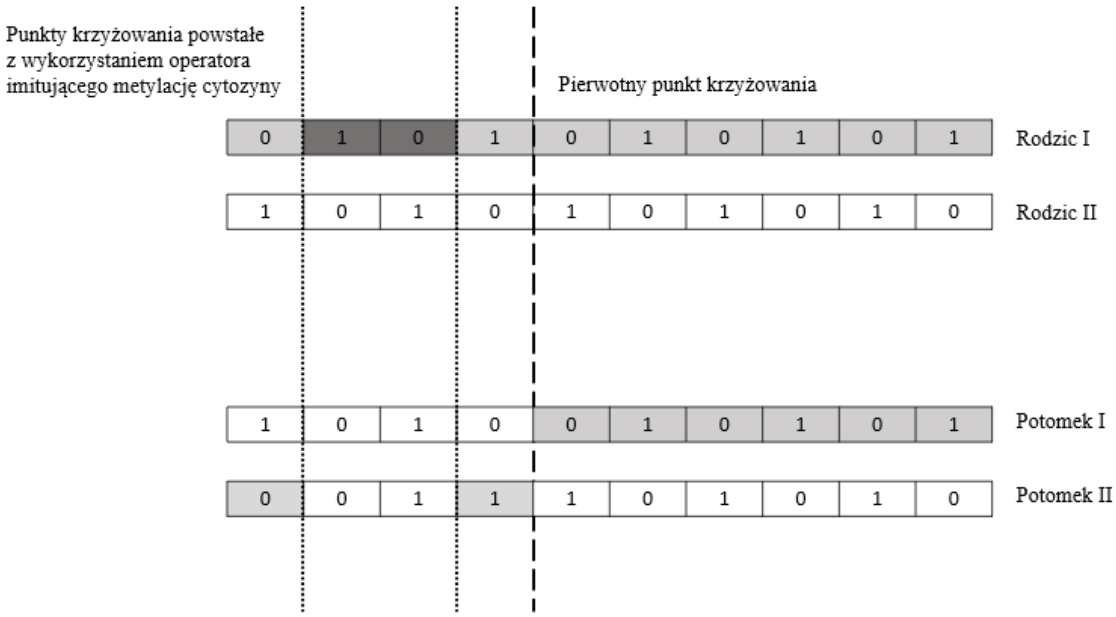
Rys. 22 Schemat działania operatora bazującego na dziedziczeniu za pomocą prionu.

4.4.5 Odwzorowanie metylacji cytozyny

Metylacja cytozyny [138] – w naukach biologicznych jest to proces przyłączania grup alkilowych (konkretnie grup metylowych do zasad azotowych nukleotydów,

w szczególności do cytozyny). Wyższy poziom metylacji genu może spowodować zanik jego ekspresji.

Zaproponowana w rozprawie [23] autorstwa Kornela Chromińskiego modyfikacja odwzorowująca metylację cytozyny symuluje blokowanie losowych sekwencji geneotypu. Modyfikacja została zintegrowana z operacją krzyżowania chromosomów. Jeżeli w danej pokoleniu wystąpi proces symulujący wyciszenie fragmentu genotypu, to sekwencja ta nie weźmie udziału w krzyżowaniu (nie zostanie przekazana osobnikom potomnym).



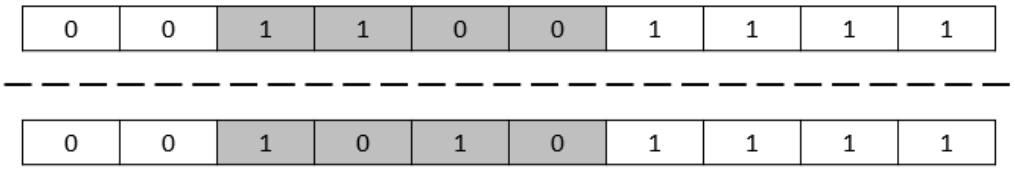
Rys. 23 Schemat działania operatora bazującego na zjawisku metylacji cytozyny.

4.4.6 Odwzorowanie wyłączenia allelicznego

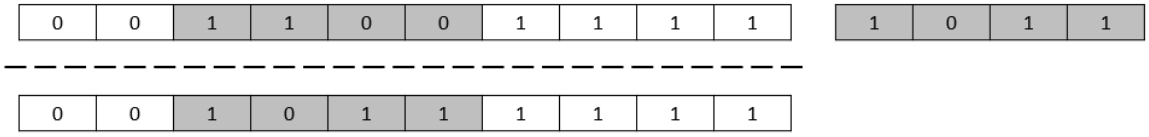
Wykluczenie alleli [139] to proces, w którym dochodzi do ekspresji tylko jednego allelu genu, podczas gdy drugi allel zostaje wyciszony. W biologii zjawisko to jest godne uwagi ze względu na rolę w rozwoju limfocytów B, gdzie wykluczenie alleli pozwala każdemu dojrzałemu limfocytowi B na ekspresję tylko jednego rodzaju immunoglobuliny. W następstwie każdy limfocyt B jest zdolny do rozpoznawania tylko jednego antygenu. Koekspresja obu alleli w limfocytach B jest związana z autoimmunizacją i wytwarzaniem autoprzeciwciał.

Proces symulujący wyłączenie alleliczne modyfikuje fragment sekwencji osobnika. Opisuje się go jako dezaktywację fragmentu genotypu oraz zamiany wyciszonego kodu na nowy. Mechanizm ten został zaimplementowany w dwóch wariantach, określanych jako

rearanżacja genomu i zastąpienie sekwencji. W pierwszym z nich wybrana sekwencja genotypu zostaje poddana permutacji. W przypadku zastąpienia sekwencji wybrany fragment zostaje zamieniony na nowy, losowy, zgodny z ustalonymi w algorytmie zasadami kodowania.



Rys. 24 Schemat działania operatora bazującego na zjawisku wyłączenia allelicznego I.



Rys. 25 Schemat działania operatora bazującego na zjawisku wyłączenia allelicznego II.

4.4.7 Ocena istniejących operatorów wzorowanych epigenetyką

Opisane w podpunktach 4.4.1–4.4.6 operatory wzorowane są na zjawiskach epigenetycznych. Modyfikują genotyp osobników w sposób mający naśladować procesy występujące w naturze. Nie robią tego jednak w sposób oddający faktyczną rolę epigenetyki w kształtowaniu fenotypu organizmów żywych.

Przykładowo, zaprezentowana „modyfikacja imitująca dziedziczenie za pomocą prionu" zakłada, że priony są losową, dziedziczną mutacją (a nie białkiem, jak ma to miejsce w naturze) występującą jednakowo na wszystkich osobnikach poza najlepiej przystosowanym i dotykającą kilku sąsiadujących genów. J. Manjrekar w pracy „*Epigenetic inheritance, prions and evolution.*" [140] uznał, że dziedziczenie oparte na prionach jest wyjątkowe wśród zjawisk epigenetycznych, ponieważ w przeciwieństwie do innych znanych mechanizmów epigenetycznych zachodzi całkowicie na poziomie białka, a nie poprzez bezpośrednią lub pośrednią modyfikację ekspresji genów. Zaproponowane rozwiązanie odbiega zatem w sposób znaczący od zjawiska biologicznego, które ma naśladować.

Opisywana modyfikacja inspirowana metylacją cytozyny zakłada, iż część genomu nie bierze udziału w procesie dziedziczenia, co skutkuje tym, że jest wyciszana. Analizując

podany w pracy przykład można zauważyć (rysunek nr 23), że zaproponowane „wyciszenie” nie różni się niczym od zwykłego krzyżowania dwupunktowego, które obecnie jest już stosowane w algorytmach genetycznych. Główna różnicą jest to, że „punkty przecięcia” dobierane są dynamicznie w taki sposób, aby osobnik lepiej przystosowany przekazał większą cześć genotypu. Mając powyższe na uwadze można uznać, że operator ten bliższy jest biologicznemu zjawisku imprintingu, chociaż także nie oddaje w pełni jego natury.

Metylacja cytozyny w naturze nie usuwa danej cechy z genomu, jak ma to miejsce w omawianym operatorze, a tylko ją wycisza. Geny dalej są przekazywane, ale nie biorą udziału w budowaniu organizmu. Z czasem, (po pewnej liczbie pokoleń) geny ponownie się aktywują do stanu pierwotnego. Zaproponowana w pracy modyfikacja ponownie tylko mutuje fragment genotypu o losowej długości. Informacja genetyczna jest tracona.

Poza błędnym założeniem, że modyfikacja epigenetyczna zmienia nieodwracalnie geny, a nie wyłącznie ich ekspresje, kluczowy w odwzorowywaniu zjawisk epigenetycznych jest powód, dla którego one występują. W znacznej mierze [141] zjawiska epigenetyczne zachodzą w następstwie oddziaływania na organizmy żywe zewnętrznych czynników, głównie stresowych. W efekcie organizm może przetrwać pomimo, iż jego pierwotny fenotyp na to nie pozwalał. W wyżej opisanych wypadkach fakt ten nie jest uwzględniony, a implementacja operatorów ogranicza się do ich losowego występowania w trakcie pracy algorytmu ewolucyjnego.

5. Problem badawczy

5.1 Problem pokrycia zbioru (SCP)

Dana jest skończona kolekcja podzbiorów $K = \{K_j: 1 \leq j \leq n\}$, gdzie każdy z nich składa się przynajmniej z jednego z elementu ze zbioru $E = \{e_i: 1 \leq i \leq m\}$. Celem jest znalezienie podkolekcji K^* takiej, że jej suma także wynosi E [142]:

$$E = \bigcup_{K_j \in K^*} K_j \qquad (11)$$

5.2 Problem minimalnego pokrycia zbioru (MSCP)

Dana jest skończona kolekcja podzbiorów $K = \{K_j: 1 \leq j \leq n\}$, gdzie każdy z nich składa się przynajmniej z jednego z elementu ze zbioru $E = \{e_i: 1 \leq i \leq m\}$. Celem rozwiązania problemu minimalnego pokrycia zbioru [143] jest znalezienie najmniejszej podkolekcji K^*:

$$min \sum_{K_j \in K} x_j \qquad (12)$$

gdzie:

$$x_j = \begin{cases} 1\ jeżeli\ K_j \in K^*, \\ 0\ w\ przeciwnym\ przypadku. \end{cases}$$

takiej że:

$$E = \bigcup_{K_j \in K^*} K_j \qquad (13)$$

5.3 Ważony problem minimalnego pokrycia zbioru (WMSCP)

Uogólnieniem problemu pokrycia zbioru jest uwzględnienie funkcji kosztu podzbiorów. Dana jest skończona kolekcja $K = \{K_j: 1 \leq j \leq n\}$ podzbiorów złożonych z elementów $E = \{e_i: 1 \leq i \leq m\}$. Przypiszemy każdemu $K_j \in K$ koszt c_j.

Rozwiązaniem problemu jest znalezienie takiego K^* , że:

$$E = \bigcup_{K_j \in K^*} K_j \qquad (14)$$

oraz

$$min \sum_{K_j \in K} c_j x_j \qquad (15)$$

gdzie:

$$x_j = \begin{cases} 1\ jeżeli\ K_j \in K^*, \\ 0\ w\ przeciwnym\ przypadku. \end{cases}$$

5.4 Wielokryterialny problem pokrycia zbioru (MCSCP)

W wielokryterialnym problemie pokrycia zbioru (ang. *Multi–Criteria Set Covering Problem*, *MCSCP*) [144] dana jest skończona kolekcja $K = \{K_j : 1 \leq j \leq n\}$ podzbiorów złożonych z elementów $E = \{e_i : 1 \leq i \leq m\}$. Przypiszemy każdemu $K_j \in K$ nieujemny i niezerowy wektor kosztów (kryteriów) $c_j^t = (c_{1j}, c_{2j}, \dots, c_{kj})$. Podkolekcja K^* kolekcji K nazywany jest pokryciem kolekcji (zbioru) K jeżeli $\bigcup_{K_j \in K^*} K_j = E$. Przypisany koszt tego pokrycia to wektor $\sum_{K_j \in K^*} c_j$. Celem jest znalezienie rozwiązania takiego, że poprawienie jakiegokolwiek parametru spowoduje pogorszenie któregokolwiek z pozostałych.

Zapis formalny :

$$min \begin{cases} \sum_{j=1}^{n} c_{1j} x_j \\ \sum_{j=1}^{n} c_{2j} x_j \\ \vdots \\ \sum_{j=1}^{n} c_{kj} x_j \end{cases} \qquad (16)$$

takie że:

$$\begin{matrix} \sum_{j=1}^{n} a_{1j}x_j \geq 1 \\ \cdot \\ \cdot \\ \cdot \\ \sum_{j=1}^{n} a_{mj}x_j \geq 1,\ x_j \in \{0,1\}, \forall j = 1,2,\dots,n. \end{matrix} \quad (17)$$

gdzie:

$$a_{ij} = \begin{cases} 1\ jeżeli\ e_i \in K_j, \\ 0\ w\ przeciwnym\ przypadku. \end{cases}$$

I rozwiązanie:

$$x_j = \begin{cases} 1\ jeżeli\ K_j \in K^*, \\ 0\ w\ przeciwnym\ przypadku. \end{cases}$$

Przyporządkujemy binarny wektor $\bar{x} = (\overline{x}_1, \overline{x}_2, \dots, \overline{x}_n)$ do podkolekcji $\overline{K}$ kolekcji K.

Dopuszczalne rozwiązanie $\bar{x} = (\overline{x}_1, \overline{x}_2, \dots, \overline{x}_n)$ problemu MCSCP nazywamy rozwiązaniem optymalnym w sensie Pareto (niezdominowanym) wtedy i tylko wtedy gdy nie ma innego rozwiązania $x = (x_1, x_2, \dots., x_n)$ takiego, że:

$\sum_{j=1}^{n} c_{ij}x_j \leq \sum_{j=1}^{n} c_{ij}\overline{x}_j$ dla każdego $i = 1,2,..,\eta$

i

$\sum_{j=1}^{n} c_{ij}x_j < \sum_{j=1}^{n} c_{ij}\overline{x}_j$ dla co najmniej jednego.

Zbiór $\overline{K}$ nazywamy zbiorem optymalnym (w sensie Pareto) jeżeli przypisany mu wektor $\overline{x}$ taki jest.

5.5 Wybrane kryteria destylacji

Współczesne destylatory korpusów dokonują selekcji na podstawie dwóch istotnych kryteriów – rozmiaru pliku oraz pokrycia kodu przez przypadek testowy. Pierwszy z nich wpływa na czas obsługi przypadku przez badany program, co przekłada się na liczbę znalezionych błędów. Korelacja między rozmiarem przypadków a efektywnością wykonanych testów jest zauważalna [145]. Niewielkie pliki posiadają stosunkowo prostą strukturę, przez co fuzzerom trudniej jest na ich podstawie wygenerować bardziej zaawansowane przypadki testowe. Dlatego zamiennie zamiast rozmiaru pliku stosuje się kryterium czasu obsługi przez badany program [15, 19].

Pokrycie kodu jest najważniejszym kryterium instrumentalizacji stosowanym w fuzzingu [146]. Mierzenie stopnia pokrycia kodu daje klarowny obraz dokładności i zasięgu testów.

W artykule [15], prezentującym wyniki działania destylatora *MoonLight*, zwrócono uwagę na względnie wysoką skuteczność *fuzzingu*, w którym zastosowano pusty korpus danych. Po początkowym okresie testów, w którym *fuzzer* nie mógł wygenerować poprawnego pliku, następowała znaczna poprawa wydajności testów pod kątem znajdowania błędów bezpieczeństwa oraz nowych krawędzi w grafie przepływu sterowania. Pusty korpus osiągał wyniki porównywalne lub lepsze od pozostałych korpusów, uzyskanych podczas destylacji pełnego zbioru danych testowych. Interesujące wydawało się, w jakim stopniu pliki wygenerowane przez *fuzzer* różnią się od plików tworzonych przez ludzi.

W artykule [18] zaproponowano entropię uproszczoną jako miarę „losowości" pliku, która posłużyła do porównania plików wygenerowanych przez *fuzzer* z plikami stworzonymi przez użytkowników poszczególnych rodzajów oprogramowania.

Entropia uproszczona [147] jest rozumiana jako średnia ważona prawdopodobieństw wystąpień każdego ze znaków ASCII w określonym ciągu danych. Wykorzystywana jest np. do analizy wstecznej plików wykonywalnych w celu wyszukiwania bloków programu, które posiadają zaszyfrowane informacje.

Wzór na entropię uproszczoną:

$$\epsilon = \sum_{d=1}^{D} p(d) * w_d \qquad (18)$$

gdzie:

$p(d)$ – prawdopodobieństwo wylosowania d–tego bajtu;

D – liczba występujących w pliku unikalnych bajtów (grup znaków);

w_d – liczba wystąpień d–tego bajtu;

U – długość badanego ciągu (pliku) w bajtach;

$\epsilon \in [1, U]$.

W celu ułatwienia procesu analizy wstecznej stosuje się następujące przekształcenie:

$$\epsilon = 1 - \frac{\sum_{d=1}^{D} p(d) * w_d}{U} = 1 - \frac{\sum_{d=1}^{D} \frac{w_d}{U} * w_d}{U} \qquad (19)$$

Zastosowanie opisanego przekształcenia sprawia, że entropia uproszczona przyjmuje wartości z zakresu $\epsilon \in [0,1]$. Dzięki temu można interpretować wyznaczoną wartość jako prawdopodobieństwo wylosowania dwóch różnych bajtów w badanym ciągu. Takie rozwiązanie posiada jedną niedoskonałość – entropia w tej postaci w praktyce nie jest w stanie osiągnąć wartości równej 1 (wspomina o tym sam autor artykułu [147]), nawet dla ciągów całkowicie różnorodnych, posiadających po jednym wystąpieniu każdego znaku z wykorzystywanego alfabetu. Aby to skorygować i móc traktować entropię uproszczoną jako wskaźnik losowości plików (prawdopodobieństwo, że dwa losowo wybrane z ciągu bajty są różne) należy skorzystać z twierdzenia o prawdopodobieństwie całkowitym:

$$P(A) = P(B_1) * P(A|B_1) + P(B_2) * P(A|B_2) + \cdots + P(B_d) * P(A|B_d) \qquad (20)$$

gdzie:

$P(A)$ – prawdopodobieństwo, że dwa wylosowane z ciągu bajty są różne (wartość entropii uproszczonej);

$P(B_d)$ – prawdopodobieństwo wylosowania jako pierwszego d–tego znaku z ciągu;

$P(A|B_d)$ – prawdopodobieństwo że dwa wylosowane z ciągu bajty są różne, pod warunkiem wylosowania jako pierwszego d–tego znaku z ciągu.

W omawianym przypadku:

$$\epsilon = \frac{w_1}{U} * \frac{(U - w_1)}{U - 1} + \frac{w_2}{U} * \frac{(U - w_2)}{U - 1} + \cdots + \frac{w_d}{U} * \frac{(U - w_d)}{U - 1} = \frac{\sum_{d=1}^{D} w_d\,(U - w_d)}{U(U - 1)} \qquad (21)$$

Czyli ostateczny wzór na entropię uproszczoną ma postać:

$$\epsilon = \frac{\sum_{d=1}^{D} w_d\,(U - w_d)}{U(U - 1)} \qquad (22)$$

Zastosowanie wyznaczonego na nowo wskaźnika entropii uproszczonej jako jednego z kryteriów destylacji pozwoliło na zwiększenie wydajności oraz efektywności fuzzingu [18]. Przyrost jest zauważalny, jednak nie tak znaczący (mały przyrost liczby znalezionych

błędów). Nie można go więc traktować jako główne kryterium redukcji zbioru danych testowych. Entropia uproszczona może być wykorzystana jako kryterium pomocnicze, rozszerzając problem ważonego pokrycia zbioru do wielokryterialnego pokrycia zbioru (pokrycia krawędzi) w testach białoskrzynkowych, bądź jako kryterium zamiennie do rozmiaru plików i czasu jego obsługi przez program w testach czarnoskrzynkowych.

Biorąc pod uwagę wszystkie ww. argumenty, proponowana w niniejszej rozprawie autorska metoda destylacji jako kryteria przyjmie:

- pokrycie kodu wyrażone w liczbie wzbudzonych krawędzi w grafie przepływu sterowania;
- czas obsługi przypadków testowych przez badany program;
- rozmiar plików wchodzących w skład korpusu;
- wartość średniej entropii przypadków testowych.

5.6 Destylacja korpusu jako problem optymalizacji wielokryterialnej

W procesie fuzzingu wykorzystuje się zbiór początkowych przypadków testowych - korpus. Jego skład wybiera się z dotychczas zebranych plików, które obsługuje testowany program. Są to pliki zebrane z Internetu, udostępnione przez autora oprogramowania, wygenerowane w sposób losowy lub uzyskane jako wynik poprzednich iteracji fuzzingu [148]. Proces wyboru odpowiednich plików – destylacja – polega na redukcji liczebności zbioru początkowego w taki sposób, aby końcowy, wykorzystywany w testach korpus cechował się maksymalnym pokryciem, mniejszym całkowitym rozmiarem, krótszym czasem obsługi wszystkich wybranych przypadków testowych oraz mniejszą średnią entropią poszczególnych plików. Istotnym jest, aby redukcja zbioru wg ww. kryteriów nie zmniejszyła pokrycia kodu, wyrażanego w liczbie unikalnych wzbudzonych przez korpus krawędzi w grafie przepływu sterowania badanego programu.

OPIS CECH I ICH DOPUSZCZALNE WARTOŚCI:

$L \in N_+$– liczba wszystkich krawędzi przejść w grafie przepływu sterowania badanego oprogramowania;

$V \in 2^N$– zbiór indeksów krawędzi przejść w grafie przepływu sterowania badanego oprogramowania;

$Z \in N_+$ – liczba zebranych przypadków testowych (początkowy korpus);

$O \in 2^N$ – zbiór indeksów zebranych przypadków testowych (początkowy korpus);

$w_u \in N_+$ – rozmiar u–tego przypadku testowego, $u = \overline{1, Z}$;

$\varepsilon_u \in \{0,1\}$ – entropia u–tego przypadku testowego, $u = \overline{1, Z}$;

$c_u \in R_+$ – czas obsługi u–tego przypadku testowego przez badany program, $u = \overline{1, Z}$;

$s_u \subset V$ – zbiór indeksów krawędzi wybudzany przez u–ty przypadek testowy, $u = \overline{1, Z}$;

$W \in N_+$ – rozmiar zredukowanego korpusu;

$\bar{\epsilon} \in [0,1]$ – średnia entropia przypadków testowych w zredukowanym korpusie;

$C \in R_+$ – czas obsługi wszystkich przypadków testowych składających się na zredukowany korpus;

$F \subset V$ – zbiór unikalnych indeksów przypadków testowych wzbudzanych przez zredukowany korpus;

$P \in N_+$ – moc rodziny podzbiorów indeksów krawędzi wzbudzanych przez zredukowany korpus;

b_u – binarna zmienna decyzyjna określająca czy u–ty przypadek testowy wchodzi w skład zredukowanego korpusu, $u = \overline{1, Z}$.

FUNKCJE CELU:

Znaleźć takie wartości zmiennych b_u aby:

$$\begin{cases} W = \min\left(\sum_{u=1}^{Z} w_u * b_u\right) \\ \bar{\epsilon} = \min\left(\sum_{u=1}^{Z} \frac{\varepsilon_u * b_u}{Z}\right) \\ C = \min\left(\sum_{u=1}^{Z} c_u * b_u\right) \\ P = max|F| = max\left|\bigcup_{u}^{Z}(s_u * b_u)\right| \end{cases}$$

ZMIENNE DECYZYJNE, KRYTERIA, DANE

Dane:

$L, V, Z, O, w_u, \varepsilon_u, c_u, s_u$

Zmienne decyzyjne:

b_u

Kryteria:

$W, \bar{\epsilon}, C, P, F$

OGRANICZENIA

$$b_u = \begin{cases} 1 - przypadek\ testowy\ jest\ częścią\ finalnego\ korpusu \\ 0 - przypadek\ testowy\ nie\ jest\ częścią\ finalnego\ korpusu \end{cases}$$

$W \leq \sum_{u=1}^{Z} w_u,\ u = \overline{1,Z}$

$C \leq \sum_{u=1}^{Z} c_u,\ u = \overline{1,Z}$

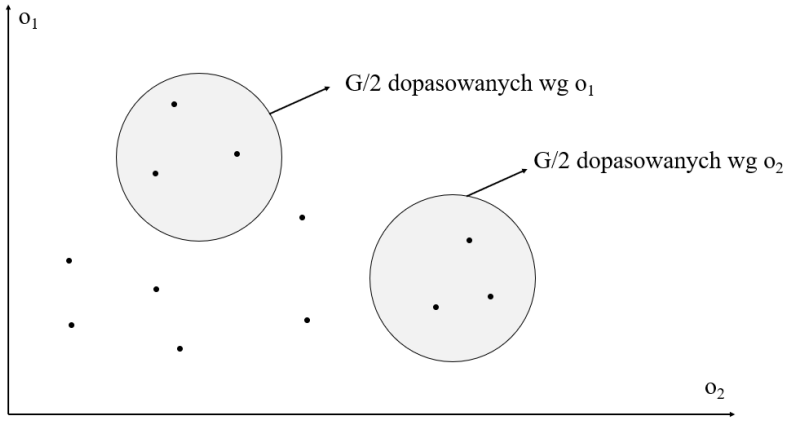
$\bar{\epsilon} \in [0,1]$

$P \leq L$

$F \subset V$

5.7 Algorytm bazowy *VEGA*

J. D. Schaffer [25, 149] zastosował podejście polegające na interesującym rozszerzeniu podstawowego algorytmu genetycznego (*ang. SGA, The Simple Genetic Algorithm*). Nowe rozwiązanie nazwał „algorytmem genetycznym ocenianym wektorowo” (*ang. Vector Evaluated Genetic Algorithm*), które różniło się od SGA sposobem selekcji. Operator ten zmodyfikowano tak, aby w każdym pokoleniu generowano pewną liczbę podpopulacji, dokonując kolejno selekcji proporcjonalnej zgodnie z każdym celem. W przypadku problemu z k celami zostanie wygenerowanych k subpopulacji o wielkości $\frac{G}{k}$ (przy założeniu, że wielkość populacji wynosi G). Podpopulacje te były następnie łączone, aby uzyskać nową populację o rozmiarze G, do której zastosowano operatory krzyżowania i mutacji analogicznie do klasycznego algorytmu genetycznego.



Rys. 26 Schemat przedstawiający tworzenie podpopulacji w algorytmie VEGA rozwiązującym zadanie dwukryterialne.

Schaffer zdał sobie sprawę, że rozwiązania generowane przez jego system są niezdominowane w sensie lokalnym, ponieważ ich niedominacja ogranicza się do obecnej populacji i chociaż rozwiązanie zdominowane lokalnie jest również zdominowane globalnie, nie jest to odwrotne. Osobnik, który jest zdominowany w jednym pokoleniu, może zostać zdominowany przez osobnika, który pojawi się w kolejnym pokoleniu. Zauważył również problem, który w genetyce jest znany jako „specjalizacja” (tj. wystąpić może wykształcenie się „gatunków” w populacji, które wyróżniają się ze względu na wybraną cechę). Zaproponowana selekcja wybiera osobniki, które przodują w jednym kryterium oceny, bez uwzględniania innych. Specjalizacja jest zjawiskiem niepożądanym, ponieważ jest sprzeczna z celem znalezienia kompromisowego rozwiązania. Algorytm *VEGA* w takiej postaci miał również tendencję do poszukiwania skrajnych rozwiązań niezdominowanych tzn. takich, które znajdują się na „krawędziach” zbioru rozwiązań niezdominowanych. Pomijane były

przez to rozwiązania pośrednie. W celu zapobiegnięcia takiemu zjawisku, można zastosować mechanizmy podziału przystosowania. Jednak przy ich wykorzystaniu utrudnione staje się wyznaczanie rozwiązań niezdominowanych. Inną techniką, która mogłaby osłabić specjalizacje jest ograniczenie krzyżowania się osobników.

W dalszych pracach nad algorytmem *VEGA* rozważana była heurystyka wyboru partnera implementowana według następującej procedury:

1. Wybierz losowo osobnika z populacji.
2. Do krzyżowania wybierz partnera, który cechował się największą odległością euklidesową.
3. Dokonaj krzyżowania wybranych osobników.

Nie udało się jednak zapobiec udziałowi gorszych (pod kątem wybranych kryteriów) osobników. Wynika to z losowego wyboru pierwszego osobnika i możliwości dużej odległości euklidesowej między nim, a słabo dopasowanym rozwiązaniem. Schaffer doszedł do wniosku, że klasyczny losowy dobór partnerów jest lepszy od opisanej heurystyki. Dlatego ostatecznie pozostał przy pierwotnej wersji algorytmu, zgodnej z poniższym opisem:

Schemat algorytmu *VEGA*

Dane wejściowe:

G – rozmiar populacji;

T – maksymalna liczba pokoleń;

p_c – prawdopodobieństwo krzyżowania;

p_m– prawdopodobieństwo mutacji.

Zmienne pomocnicze:

P_κ– populacja, zbiór osobników wchodzących w skład κ - tego pokolenia;

$P'_\kappa -$ populacja pomocnicza;

$P''_\kappa -$ populacja pomocnicza;

$P''''_\kappa -$ populacja pomocnicza;

ch_g- chromosom z populacji P_κ;

g – indeks chromosomu (osobnika);

$k -$ indeks kryterium;

κ − indeks pokolenia.

Wyjście algorytmu:

Y – zbiór rozwiązań niezdominowanych.

Krok 1:

Inicjalizacja:

Niech $P_0 = \emptyset$ oraz $\kappa = 0$. Następnie dla $g = 1, \ldots, G$ wykonaj:

a) wygeneruj (wczytaj) osobnika ch_g;

b) dodaj osobnika ch_g do populacji P_0.

Przypisz $P_\kappa = P_0$.

Krok 2:

Przystosowanie i selekcja:

Ustaw $P_\kappa' = \emptyset$ oraz dla każdego z kryteriów wykonaj:

a) dla wszystkich $ch_g \in P_\kappa$ wylicz jego przystosowanie wyznaczając wartość funkcji celu $o_k(ch_g)$

b) dla każdego z kryteriów dokonaj $\frac{G}{k}$ selekcji osobników i skopiuj je do populacji P_κ'

Krok 3:

Rekombinacja:

Ustaw $P_\kappa'' = \emptyset$ dla każdego $g = 1, \ldots, \frac{G}{2}$ wykonaj:

a) wybierz dwa osobniki z P_κ' a następnie usuń je z P_κ',

b) dokonaj operacji krzyżowania, którego wynikiem są dwa osobniki potomne.

c) dodaj osobniki potomne z prawdopodobieństwem p_c do P_κ'', w przeciwnym razie przenieś osobniki rodzicielskie do P_κ''.

Krok 4:

Mutacja:

Ustaw $P_\kappa''' = \emptyset$ dla każdego osobnika z populacji P_κ'' wykonaj:

a) podaj osobnika operacji mutacji z prawdopodobieństwem p_m i dodaj go do P_κ'''.

Krok 5:

Kończenie algorytmu:

Ustaw nowe pokolenie bazowe $P_{\kappa+1} = P_{\kappa}'''$ oraz $\kappa = \kappa + 1$. Sprawdź jeżeli $\kappa \geq T$ (lub dodatkowe kryterium zakończenia algorytmu) jest spełnione, umieść rozwiązania niezdominowane z populacji P_{κ} w zbiorze Y i zakończ działanie algorytmu.

W przeciwnym razie przejdź do **Krok 2**.

5.8 Sposób tworzenia osobników

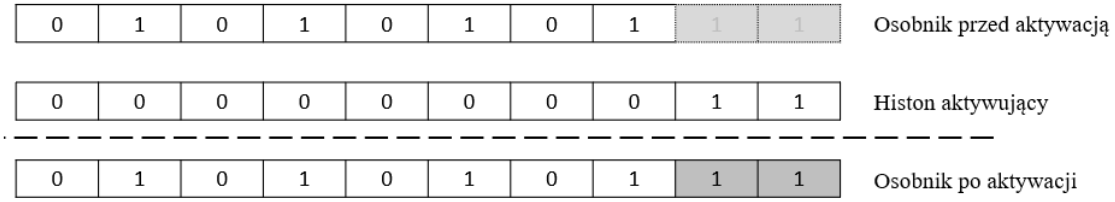
W celu przeprowadzenia destylacji korpusu danych testowych, podjęta zostanie próba rozwiązania wielokryteriowego problemu pokrycia zbioru z wykorzystaniem algorytmu genetycznego. Konieczne jest zatem zastosowanie niestandardowego kodowania, które reprezentuje potencjalne rozwiązanie ww. problemu. Każdy gen w chromosomie reprezentować będzie dokładnie jeden z plików, wchodzących w skład pierwotnego korpusu, który poddany zostanie destylacji. Gen o wartości „1" reprezentować będzie obecność pliku w końcowym przedestylowanym korpusie, „0" zaś jego brak.

5.9 Histon – modyfikator epigenetyczny

5.9.1 Histon aktywujący (acetylujący)

W rozwiązaniu optymalnym znajdować się będą pliki, które wzbudzają unikalne krawędzie (czyli takie, które są wzbudzane tylko przez jeden plik). Pliki takie można wstępnie wykluczyć z chromosomu, dzięki czemu zmniejszy się liczba potencjalnych rozwiązań, rozpatrywanych przez algorytm genetyczny. Przypadki testowe zawierające unikalne krawędzie zostaną dodane dopiero do finalnego rozwiązania, po zakończeniu pracy algorytmu genetycznego.

Proponowany autorski operator epigenetyczny wzorowany na acetylacji histonu, wykorzystuje geny reprezentujące pliki wzbudzające unikalne krawędzie. W przypadku, gdy osobnik poddany zostanie czynnikowi stresującemu (w przypadku algorytmów genetycznych - selekcji) i wpłynie on na niego negatywnie (zostanie odrzucony z przyszłej populacji), zastosowanie aktywacji genów „wyciszonych" może pomóc mu przetrwać, a w efekcie posiadać potomstwo.

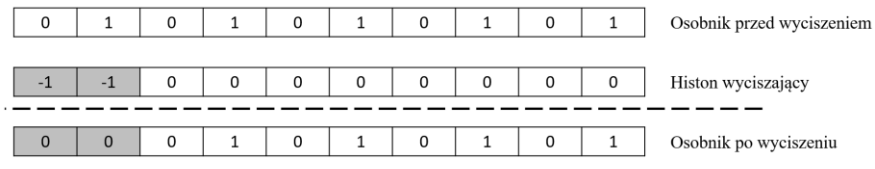


Rys. 27 Schemat działania histonu aktywującego.

5.9.2 Histon wyciszający (deacetylujący)

Destylacja korpusu danych testowych w każdym z jej wariantów promuje osobniki o niewielkim rozmiarze. Związane jest to z ich szybszą obsługą przez badany program. Niewielkie pliki ułatwiają również analizę znalezionego błędu, przyśpieszając proces jego eksploitacji. Zbiór początkowy podlegający redukcji zawierać może pliki o znacznym rozmiarze, przez co szansa na ich wystąpienie w korpusie końcowym ostatecznie i tak jest niska (np. zalecenia autorów *AFL++* określają, że pliki wybrane do koprusu nie powinny być większe niż 1 kB, a 10 kB pliki mają 1% szans na wywołanie błędu w pierwszych 1000 wywołaniach [150]).

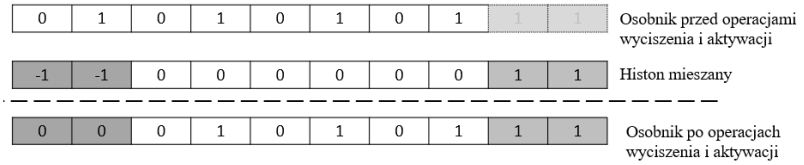
W przypadku gdy osobnik - potencjalny najlepszy korpus - poddany zostanie czynnikowi stresującemu (w przypadku algorytmów genetycznych - selekcji) i wpłynie on na niego negatywnie (zostanie odrzucony), zastosowanie wyciszenia genów reprezentujących największe pliki może pomóc mu przetrwać (potencjalnie posiadać potomstwo). Dalszym rozważaniom podlegać może jaką cześć genów należy uznać jako przeznaczone do potencjalnego wyciszenia. W prezentowanych w dalszej części niniejszej pracy badaniach jako pliki „za duże” uznano 5% największych plików z pierwotnej populacji. Decyzja ta podyktowana była niewielką obecnością plików większych niż 10 kB w przygotowanych, początkowych korpusach.



Rys. 28 Schemat działania histonu wyciszającego.

5.9.3 Histon mieszany

Wyciszanie i aktywacja genów w naturze występuje równolegle, dlatego też kolejny rozważany wariant operatora genetycznego łączy w sobie aktywację i dezaktywację wybranych genów.



Rys. 29 Schemat działania histonu mieszanego

5.10 Sterowanie zbieżnością

Do bazowego algorytmu *VEGA*, w ramach proponowanego rozwiązania, dodatkowo wprowadzony został mechanizm sterowania zbieżnością. Oparty został na dwóch statystykach: rozstępie oraz kwartylu drugiego rzędu głównego kryterium destylacji – liczby pokrytych krawędzi w grafie przepływu sterowania.

Utrzymanie większej bioróżnorodności, która jest utożsamiana z wyższą wartością miary dyspersji, ma za zadanie wpłynąć na dokładniejsze przeszukiwanie przestrzeni potencjalnych rozwiązań. Zostanie to osiągnięte poprzez modyfikowanie prawdopodobieństwa mutacji. Jeżeli rozrzut cechy będzie się zmniejszał przy kolejnych pokoleniach, zwiększone zostanie prawdopodobieństwo jej wystąpienia. Jego redukcja nastąpi zaś wraz z wzrostem rozstępu.

Zbieżność algorytmu genetycznego można osiągnąć z wykorzystaniem samego operatora krzyżowania [151]. Jednak zbyt duże prawdopodobieństwo jego wystąpienia doprowadza do wcześniejszego zakończenia pracy algorytmu. Dlatego zaproponowany mechanizm będzie równocześnie redukował szansę na wystąpienie krzyżowania w sytuacji, gdy mediana liczby pokrytych krawędzi będzie rosnąć w czasie pracy algorytmu. W przypadku spadku wartości centralnej prawdopodobieństwo kojarzenia osobników wzrośnie, aby zwiększyć presje selekcyjną w kierunku wartości maksymalnej.

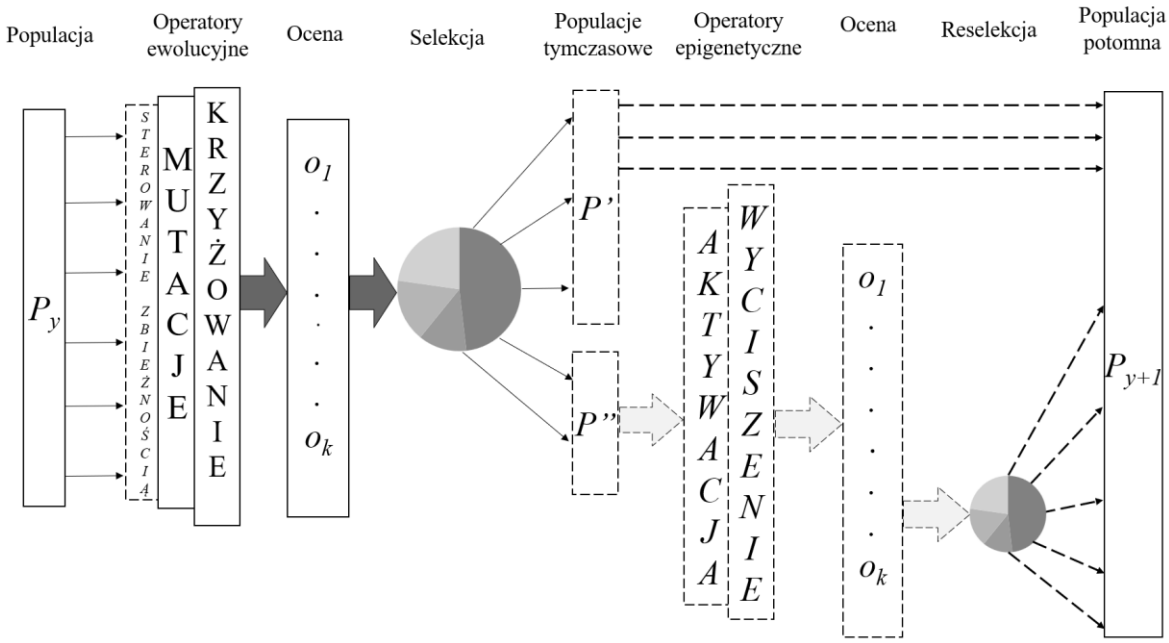
Wyżej opisane rozwiązanie pozwoli na utrzymanie bioróżnorodności i dalsze dążenie algorytmu do zbieżności, przy czym zapobiegnie przedwczesnemu jej wystąpieniu.

Wybór wyżej wymienionych statystyk podyktowany był łatwością ich implementacji oraz faktem, iż ich wyznaczanie nie wymaga realizacji znacznej liczby obliczeń, co mogłoby negatywnie wpłynąć na czas pracy algorytmu genetycznego.

5.11 Algorytm *EpiVEGA*

Proponowana autorska modyfikacja algorytmu *VEGA* opiera się na wprowadzeniu do algorytmu nowego operatora epigenetycznego opartego na wektorze – histonie, zawierającym informacje o możliwym wyciszeniu bądź aktywacji genów. To co wyróżnia prezentowane rozwiązanie na tle wcześniej wymienionych przykładów odwzorowywania zjawisk epigenetycznych, to wprowadzenie dodatkowego etapu – reselekcji. Po etapie selekcji osobniki, które z racji zbyt niskiej wartości funkcji przystosowania zostałyby wyeliminowane z populacji, zostaną poddane działaniu operatów epigenetycznych (wyciszeniu, aktywacji bądź operacji mieszanej) z prawdopodobieństwem p_e (ustalonym eksperymentalnie). Procedura ta ma symulować czynnik stresowy, który w ewolucji biologicznej odpowiedzialny jest za wystąpienie zjawisk epigenetycznych. Tak jak w naturze, zjawisko epigenetyczne ma za zadanie zapobiec śmierci organizmu pomimo jego słabego przystosowania. Całość w założeniu ma wpłynąć pozytywnie na zdefiniowaną w literaturze wadę algorytmu *VEGA* – tendencje do wyszukiwania niezdominowanych rozwiązań ekstremalnych, co skutkuje pomijaniem rozwiązań pośrednich. Operator epigenetyczny pozwoli przetrwać „obiecującym" osobnikom, które zostałyby odrzucone przez pierwotny algorytm.

Dodatkowo w przypadku zadania optymalizacji, w którym jedna z cech jest maksymalizowana, a pozostałe minimalizowane, oprogramowanie oparte na algorytmie *VEGA* może wygenerować populację, która przy odpowiedniej ilości pokoleń zostanie z dużym prawdopodobieństwem zdominowana przez osobniki reprezentujące minimalną wartość redukowanej cechy. W przypadku destylacji korpusu rozwiązanie to będzie tożsame z wyznaczeniem zbioru pustego. Stanie się tak, ponieważ zbiór nieposiadający żadnych elementów cechuje się najniższą wagą, najniższą średnią entropią uproszczoną oraz najniższym czasem obsługi. Aby tego uniknąć, zastosowano mechanizmy sterowania zbieżnością, które poprzez modyfikację prawdopodobieństwa wystąpienia klasycznych operatorów genetycznych pozwolą na uniknięcie uzyskania takiej solucji.



Rys. 30 Schemat algorytmu epiVEGA

<u>Schemat algorytmu epi*VEGA*</u>

Dane wejściowe:

G – rozmiar populacji;

T – maksymalna liczba pokoleń;

T_{max} – maksymalna liczba pokoleń bez poprawy;

p_c – prawdopodobieństwo krzyżowania;

p_m– prawdopodobieństwo mutacji;

p_e – prawdopodobieństwo wystąpienia operacji epigenetycznej;

k – indeks kryteriów.

Zmienne pomocnicze:

κ – numer bieżącego pokolenia;

κ_p– liczba pokoleń bez poprawy;

ϑ – histon;

r_κ – rozstęp głównej cechy w κ-tej populacji;

m_κ– mediana głównej cechy w κ-tej populacji;

ch_g– g–ty chromosom w populacji;

ch_l– l–ty chromosom w populacji pomocniczej;

Y_κ – najlepszy zbiór rozwiązań niezdominowanych wyznaczony przez algorytm do chwili (pokoleniu) κ;

Y' – tymczasowy zbiór rozwiązań niezdominowanych;

P'_κ – populacja pomocnicza;

P''_κ – populacja pomocnicza;

P''''_κ – populacja pomocnicza;

P_l – populacja pomocnicza (osobniki odrzucone);

P'_l – populacja pomocnicza (odrzucone osobniki poddawane operacji epigenetycznej);

Wyjście algorytmu:

Y – zbiór rozwiązań niezdominowanych.

Krok 1:

Generowanie histonu:

Utwórz zerowy wektor binarny ϑ. Wykonaj zgodnie z kodowaniem:

a) Oznacz negatywnie wpływające na wynik problemu allele jako „–1”;
b) Oznacz wartości unikalne dla problemu jako „1”. Jeżeli wartość negatywna jest unikalna pozostaje oznaczona jako „1”.

Krok 2:

Inicjalizacja:

Niech $P_0 = \emptyset$ oraz $\kappa = 0$. Następnie dla $g = 1, \dots, G$ wykonaj:

a) wygeneruj (wczytaj) osobnika ch_g;

b) dodaj osobnika ch_g do populacji P_0;

Ustaw $P_\kappa = P_0$.

Wyznacz rozstęp r_κ oraz medianę m_κ dla P_κ.

Umieść rozwiązania niezdominowane z populacji P_κ w zbiorze Y_κ.

Ustaw $\kappa_p = 0$.

Krok 3:

Przystosowanie i selekcja:

Ustaw $P_\kappa' = \emptyset$, $P_l = \emptyset$ oraz dla każdego z kryteriów wykonaj:

a) dla $ch_g \in P_\kappa$ wyznacz jego przystosowanie dla każdego z kryteriów wyznaczając wartość funkcji celu $o_k(ch_g)$;

b) dla każdego z kryteriów dokonaj $\frac{G}{k+1}$ selekcji osobników i skopiuj je do populacji P_κ'.

Każdy osobnik, który nie znalazł się w tymczasowej P_κ' zostaje skopiowany do populacji P_l.

Krok 4:

Operator epigenetyczny i reselekcja:

Ustaw $P'_l = \emptyset$, $l = 1, \dots, |P_l|$ i wykonaj:

a) Każdego osobnika $ch_l \in P_l$ poddaj wybranej operacji epigenetycznej w oparciu o ϑ z prawdopodobieństwem p_e;

b) dla $ch_l \in P_l$ wyznacz jego przystosowanie dla każdego z kryteriów wyznaczając wartość funkcji celu $o_k(ch_l)$;

Dla każdego z kryteriów dokonaj $\frac{G}{k*(k+1)}$ selekcji osobników i skopiuj je do populacji P'_l.

Wykonaj połączenie zbiorów $P_\kappa' = P_\kappa' + P'_l$ i przejdź do Kroku 5.

Krok 5:

Rekombinacja:

Ustaw $P_\kappa'' = \emptyset$ dla każdego $g = 1, \dots, \frac{G}{2}$ wykonaj:

a) wybierz dwa osobniki z P_κ' a następnie usuń je z P_κ';

b) dokonaj operacji krzyżowania, którego wynikiem są dwa osobniki potomne;

c) dodaj osobniki potomne z prawdopodobieństwem p_c w przeciwnym razie przenieś osobniki rodzicielskie P_κ''.

Krok 6:

Mutacja:

Ustaw $P_\kappa''' = \emptyset$ dla każdego osobnika w populacji P_κ'' wykonaj:

a) podaj osobnika operacji mutacji z prawdopodobieństwem p_m i dodaj go do P_κ'''.

Krok 7:

Kończenie algorytmu:

a) Ustaw nowe pokolenie bazowe $P_{\kappa+1} = P_{\kappa}'''$ oraz $\kappa = \kappa + 1$;

b) Wyznacz rozstęp $r_{\kappa+1}$ oraz medianę $m_{\kappa+1}$ dla P_{κ}''' ;

c) Jeżeli $r_{\kappa+1} > r_{\kappa}$ to zmniejsz p_m ;
W przeciwnym przypadku:
Jeżeli $r_{\kappa+1} < r_{\kappa}$ to zwiększ p_m ;

d) Jeżeli $m_{\kappa+1} > m_{\kappa}$ to zwiększ p_c;
W przeciwnym przypadku:
Jeżeli $m_{\kappa+1} < m_{\kappa}$ to zmniejsz p_c;

e) Umieść rozwiązania niezdominowane z populacji P''' w zbiorze Y';
Sprawdź, czy rozwiązania niezdominowane Y' są lepsze niż w zbiorze Y_{κ}.
Jeżeli tak to $\kappa_p = 0$ i $Y_{\kappa} = Y'$, w przeciwnym przypadku $\kappa_p = \kappa_p + 1$.

f) Jeżeli $\kappa \geq T$ lub $\kappa_p \geq T_{max}$ to $Y = Y_{\kappa}$ i zakończ działanie algorytmu.

W przeciwnym razie przejdź do **Krok 3**.

5.12 Destylator epi–min

Algorytm *epiVEGA* został zaimplementowany jako skrypt *epi–min.py* w języku Python3. Docelowym środowiskiem uruchomieniowym jest system *Linux*. Program jako argumenty wejściowe przyjmuje:

- ścieżkę do testowanego programu;
- ścieżkę do pierwotnego, pełnego korpusu;
- ścieżkę do której zapisywany będzie wynik programu (przedestylowany korpus);
- rozmiar populacji algorytmu epigenetycznego;
- tryb pracy operatora epigenetycznego (wyciszający, aktywujący, mieszany, brak operatora epigenetycznego);
- zastosowanie (bądź nie) mechanizmów sterowania zbieżnością.

Testowany program, napisany w językach C bądź C++ musi zostać skompilowany z wykorzystaniem kompilatora *clang* z poniższymi flagami:

- *fsanitize=address*
- *fsanitize–coverage=trace–pc–guard*

Destylator *epi–min.py* uruchamia tak skompilowany plik binarny z flagą:

– *ASAN_OPTIONS=coverage=1*

Skutkuje to wygenerowaniem pliku **.sancov,* będącego atrybutem wejściowym aplikacji *sancov* [57]. Wynikiem wykonania wymienionego programu jest lista adresów krawędzi w grafie przepływu sterowania badanego programu.

W wersji rozwojowej skrypt *epi–min.py* poza przedestylowanym korpusem generuje dane służące do dalszej analizy statystycznej. Do dodatkowego katalogu zapisywane są wartości cech osobników wchodzących w skład każdej z populacji wygenerowanej w czasie pracy algorytmu genetycznego. Interpretacja tych danych pozwala na prześledzenie zbieżności każdej z rozpatrywanych w procesie redukcji korpusu cech.

Uruchomienie skryptu z parametrami, które wykluczają zastosowanie operatora epigenetycznego oraz sterowania zbieżnością skutkuje uruchomieniem destylatora bazującego na pierwotnym algorytmie *VEGA*.

6. Etap eksperymentalny

Celami etapu eksperymentalnego są:

- ustalenie domyślnej wielkości populacji w algorytmie *epiVEGA*;
- ustalenie domyślnej wartości p_e;
- porównanie parametrów korpusów wyznaczonych z wykorzystaniem algorytmów *afl–cmin*, *VEGA*, *epiVEGA;*
- porównanie skuteczności fuzzingu dla wybranych formatów.

6.1 Zastosowane środowisko

Wszystkie etapy fazy eksperymentalnej zostały zrealizowane na stanowisku komputerowym o następujących parametrach technicznych:

- Procesor: Intel i5–6600K 3.5 GHz;
- Pamięć RAM: 24 GB DDR4 2133 MHz;
- Dysk twardy: SSD 120 GB 560 Mb/s (odczyt) 530 Mb/s (zapis);
- System Operacyjny: Ubuntu 20.04 LTS.

Kompilacja, destylacja korpusu oraz *fuzzing* uruchamiany był z wykorzystaniem jednego rdzenia procesora.

6.2 Wybór formatów plików

W celu oceny uniwersalności zaproponowanego rozwiązania wybrano cztery biblioteki na licencjach *open–source*, służące do obsługi znacznie różnych się od siebie budową formatów plików:

- pliki tekstowe **.xml* : **libxml2 2.9.12**;
- dokumenty tekstowo–graficzne **.pdf* : **pdfio 1.0**;
- pliki graficzne **.gif*: **giflib 5.2.1**;
- pliki skompresowane **.lz4*: **LZ4 1.9.3**.

6.3 Metodyka eksperymentów

Eksperyment został podzielony na dwa główne etapy:

- destylacje korpusu danych testowych;
- *fuzzing*.

W przypadku destylacji, przygotowano cztery zbiory o liczebności 1000 plików. Zbiory zostały zaczerpnięte z Internetu z wykorzystaniem crawlingu. Każdy z nich został poddany destylacji z wykorzystaniem algorytmów *cmin*, *VEGA* i *epiVEGA*.

Warunkami zmiennymi procesu destylacji z wykorzystaniem algorytmów genetycznych były rozmiar populacji (*VEGA*, *epiVEGA*), prawdopodobieństwo wystąpienia operatora epigenetycznego (*epiVEGA*) oraz jego rodzaj (*epiVEGA*).

Po zakończeniu fazy eksperymentalnej procesu destylacji dokonano ewaluacji otrzymanych wyników. Wybrano te parametry pracy każdego z algorytmów, które pozwoliły na uzyskanie wyników najlepszych. W związku z tym, że ocenie podlegały cztery cechy korpusu ustalono ich priorytet według następującej kolejności:

1. Pokrycie krawędzi w grafie przepływu sterowania.
2. Waga korpusu.
3. Czas obsługi całego korpusu.
4. Średnia entropia uproszczona całego korpusu.

Pozwoliło to na ostateczne porównanie zbiorów wynikowych. Zbiory cechujące się najlepszymi wartościami zostały wybrane do etapu drugiego eksperymentu - fuzzingu.

Fuzzing przeprowadzono z wykorzystaniem oprogramowania *AFL++ 3.13c*, uruchomionego w podstawowej konfiguracji, z czasem pracy ustalonym na 100 godzin. Dla każdego z czterech formatów plików jako korpusy wejściowe ustalono:

- zbiór pusty;
- zbiór pełen;
- korpus uzyskany z wykorzystaniem algorytmu *cmin*;
- korpus uzyskany z wykorzystaniem algorytmu *VEGA*;
- korpus uzyskany z wykorzystaniem algorytmu *epiVEGA*.

Efektami pracy fuzzera, które podlegały porównaniu, były:

- liczba znalezionych unikalnych błędów;
- liczba znalezionych unikalnych zawieszeń programu;
- liczba znalezionych krawędzi;
- liczba wszystkich pokrytych krawędzi.

6.4 Destylacja korpusów algorytmem epi*VEGA*

Wyniki pracy destylatora *epi–min* zostały zaprezentowane w tabelach, zawierających wartości rzeczywiste oraz procentowe. Jako punkt odniesienia przyjęto pierwotny,

niezredukowany korpus. Najlepsze uzyskane wyniki dla danego formatu plików zostały wyróżnione ciemniejszym odcieniem wiersza. Dodatkowo zaprezentowano przebieg pracy algorytmu genetycznego dla 1000 pokoleń w konfiguracji, z pomocą której uzyskano najlepszy wynik.

6.4.1 Pliki graficzne

6.4.1.1 Histon wyciszający

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Wyciszający	2164589	0,94	39,73	235
200	0,1	Wyciszający	713488	0,92	20,55	235
500	0,1	Wyciszający	214405	0,88	11,80	235
100	0,2	Wyciszający	2002995	0,93	37,65	235
200	0,2	Wyciszający	411301	0,90	16,38	235
500	0,2	Wyciszający	486935	0,90	19,01	235
100	0,5	Wyciszający	1924221	0,93	22,93	235
200	0,5	Wyciszający	488551	0,90	21,23	235
500	0,5	Wyciszający	301945	0,90	12,50	235

Tab. 1 Wynik destylacji algorytmem epiVEGA korpusu pików graficznych z operatorem wyciszającym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Wyciszający	25,06	99,38	42,42	100,00
200	0,1	Wyciszający	8,26	97,66	21,94	100,00
500	0,1	Wyciszający	2,48	92,93	12,60	100,00
100	0,2	Wyciszający	23,19	98,71	40,20	100,00
200	0,2	Wyciszający	4,76	94,81	17,49	100,00
500	0,2	Wyciszający	5,64	95,21	20,29	100,00
100	0,5	Wyciszający	22,28	98,63	24,48	100,00
200	0,5	Wyciszający	5,66	95,13	22,67	100,00
500	0,5	Wyciszający	3,50	95,83	13,34	100,00

Tab. 2 Wyniki destylacji algorytmem epiVEGA korpusu plików graficznych z operatorem wyciszającym (wartości procentowe).

Wykorzystanie operatora wyciszającego dla plików graficznych pozwoliło przeprowadzić destylację z zachowaniem pierwotnego pokrycia, przy czym zwiększanie prawdopodobieństwa jego wystąpienia wpływało pozytywnie na uruchomienia algorytmu z zadanymi mniejszymi rozmiarami populacji. Zwiększanie rozmiaru przetwarzanej przez algorytm populacji pozytywnie wpłynęło na redukcję wartości wszystkich minimalizowanych

cech, bez konieczności zwiększania prawdopodobieństwa wystąpienia operatora epigenetycznego.

6.4.1.2 Histon aktywujący

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Aktywujący	2676740	0,93	36,32	235
200	0,1	Aktywujący	1500688	0,91	29,92	235
500	0,1	Aktywujący	537653	0,90	17,95	235
100	0,2	Aktywujący	1706969	0,93	25,77	235
200	0,2	Aktywujący	1107356	0,92	22,35	235
500	0,2	Aktywujący	1048781	0,90	26,43	235
100	0,5	Aktywujący	3846641	0,94	41,99	235
200	0,5	Aktywujący	792642	0,90	17,31	235
500	0,5	Aktywujący	3348197	0,94	40,76	235

Tab. 3 Wyniki destylacji algorytmem epiVEGA korpusu plików graficznych z operatorem aktywującym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Aktywujący	30,99	99,03	38,77	100,00
200	0,1	Aktywujący	17,37	96,73	31,95	100,00
500	0,1	Aktywujący	6,22	95,62	19,16	100,00
100	0,2	Aktywujący	19,76	98,14	27,51	100,00
200	0,2	Aktywujący	12,82	97,44	23,86	100,00
500	0,2	Aktywujący	12,14	95,75	28,22	100,00
100	0,5	Aktywujący	44,54	99,33	44,83	100,00
200	0,5	Aktywujący	9,18	95,62	18,48	100,00
500	0,5	Aktywujący	38,76	99,69	43,51	100,00

Tab. 4 Wyniki destylacji algorytmem epiVEGA korpusu plików graficznych z operatorem aktywującym (wartości procentowe).

Wykorzystanie operatora aktywującego dla plików graficznych także pozwoliło przeprowadzić destylację z zachowaniem pierwotnego pokrycia, przy czym zwiększanie prawdopodobieństwa jego wystąpienia wpływało pozytywnie na uruchomienia algorytmu z zadanymi mniejszymi rozmiarami populacji. Wyjątkiem jest populacja o rozmiarze 500 osobników, która pomimo zastosowania większej liczby przetwarzanych osobników, nie pozwoliła na uzyskanie poprawy wyników.

6.4.1.3 Operator mieszany

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Mieszany	2691460	0,94	40,31	235
200	0,1	Mieszany	796187	0,91	22,24	235
500	0,1	Mieszany	641005	0,91	19,22	235
100	0,2	Mieszany	2562433	0,94	38,50	235
200	0,2	Mieszany	1044626	0,92	25,26	235
500	0,2	Mieszany	197819	0,87	11,53	235
100	0,5	Mieszany	536881	0,91	17,77	235
200	0,5	Mieszany	1687261	0,93	32,36	235
500	0,5	Mieszany	1564796	0,93	31,99	235

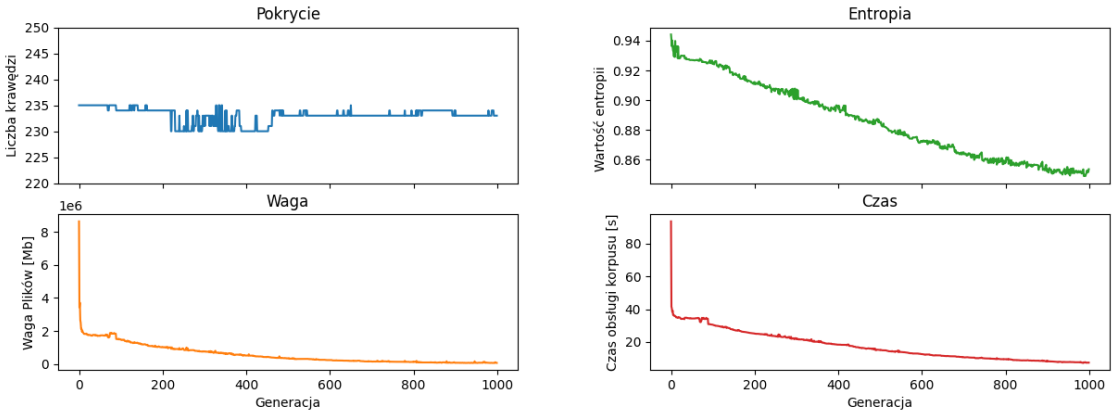
Tab. 5 Wyniki destylacji algorytmem epiVEGA korpusu plików graficznych z operatorem mieszanym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Mieszany	31,16	99,46	43,04	100,00
200	0,1	Mieszany	9,22	96,47	23,75	100,00
500	0,1	Mieszany	7,42	96,06	20,52	100,00
100	0,2	Mieszany	29,67	99,68	41,11	100,00
200	0,2	Mieszany	12,09	97,70	26,97	100,00
500	0,2	Mieszany	2,29	92,04	12,31	100,00
100	0,5	Mieszany	6,22	96,89	18,97	100,00
200	0,5	Mieszany	19,53	98,54	34,55	100,00
500	0,5	Mieszany	18,12	98,30	34,15	100,00

Tab. 6 Wyniki destylacji algorytmem epiVEGA korpusu plików graficznych z operatorem mieszanym (wartości procentowe).

Wykorzystanie operatora mieszanego dla plików graficznych również pozwoliło przeprowadzić destylację z zachowaniem pierwotnego pokrycia. Z jego wykorzystaniem uzyskano najlepszy wynik pracy algorytmu genetycznego (dla populacji liczącej 500 osobników, przy prawdopodobieństwie wykorzystanie operatora epigenetycznego równym $p_e = 0{,}2$).

Przebieg procesu destylacji algorytmem *epiVEGA* skutkującej uzyskaniem najlepszego rozwiązania przedstawiono na rysunku nr 31.



Rys. 31 Przebieg procesu destylacji algorytmem epiVEGA korpusu plików graficznych dla populacji o liczebności 500 osobników oraz p_e=0,2 dla operatora mieszanego.

Algorytm *epiVEGA* skutecznie uniemożliwił pogorszenie się stopnia pokrycia kodu biblioteki *giflib 5.2.1*, przy jednoczesnej redukcji czasu obsługi, rozmiaru korpusu i jego średniej entropii.

6.4.2 Pliki tekstowe

6.4.2.1 Histon wyciszający

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Wyciszający	83246	0,85	22,64	3123
200	0,1	Wyciszający	85805	0,85	23,47	3124
500	0,1	Wyciszający	80393	0,85	13,58	3121
100	0,2	Wyciszający	76046	0,85	24,72	3126
200	0,2	Wyciszający	90403	0,84	24,87	3125
500	0,2	Wyciszający	87892	0,85	14,08	3123
100	0,5	Wyciszający	84053	0,86	23,02	3119
200	0,5	Wyciszający	76331	0,85	24,28	3125
500	0,5	Wyciszający	89862	0,84	21,06	3126

Tab. 7 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowych z operatorem wyciszającym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Wyciszający	60,68	99,29	48,25	99,81
200	0,1	Wyciszający	62,55	99,31	50,01	99,84
500	0,1	Wyciszający	58,60	99,19	28,93	99,74
100	0,2	Wyciszający	55,43	99,06	52,68	99,90
200	0,2	Wyciszający	65,90	98,29	52,99	99,87
500	0,2	Wyciszający	64,07	98,86	30,01	99,81
100	0,5	Wyciszający	61,27	100,51	49,06	99,68
200	0,5	Wyciszający	55,64	99,83	51,74	99,87
500	0,5	Wyciszający	65,50	97,85	44,88	99,90

Tab. 8 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowych z operatorem wyciszającym (wartości procentowe).

Destylacja przeprowadzona z wykorzystaniem operatora histonu wyciszającego pozwoliła na redukcję rozmiaru korpusu o 35–45% i skrócenie o 50–70% czasu obsługi. Entropia pozostała niemal nie zmieniona z różnicami nie większymi niż 2%. Nie udało się także utrzymać pełnego pokrycia, co poskutkowało jego spadkiem o mniej niż 0,4%.

6.4.2.2 Histon aktywujący

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Aktywujący	95205	0,85	23,81	3123
200	0,1	Aktywujący	93697	0,83	21,06	3126
500	0,1	Aktywujący	96342	0,84	23,92	3127
100	0,2	Aktywujący	93898	0,82	20,62	3124
200	0,2	Aktywujący	71530	0,85	21,82	3125
500	0,2	Aktywujący	78428	0,86	15,26	3122
100	0,5	Aktywujący	87239	0,84	23,43	3124
200	0,5	Aktywujący	85746	0,85	23,70	3126
500	0,5	Aktywujący	82223	0,85	16,18	3125

Tab. 9 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowych z operatorem aktywującym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Aktywujący	69,40	99,29	50,74	99,81
200	0,1	Aktywujący	68,30	97,27	44,88	99,90
500	0,1	Aktywujący	70,23	98,46	50,96	99,94
100	0,2	Aktywujący	68,45	96,22	43,93	99,84
200	0,2	Aktywujący	52,14	99,67	46,50	99,87
500	0,2	Aktywujący	57,17	100,88	32,52	99,78
100	0,5	Aktywujący	63,59	98,14	49,92	99,84
200	0,5	Aktywujący	62,50	98,97	50,51	99,90
500	0,5	Aktywujący	59,93	99,05	34,47	99,87

Tab. 10 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowych z operatorem aktywującym (wartości procentowe).

Zastosowanie histonu aktywującego pozwoliło na redukcję rozmiaru korpusu o 30–50% i skrócenie o 50–65% czasu obsługi. Nastąpiło obniżenie średniej entropii maksymalnie o 4%. Ponownie nie udało się utrzymać pełnego pokrycia, co poskutkowało jego spadkiem o mniej niż 0,2%.

6.4.2.3 Histon mieszany

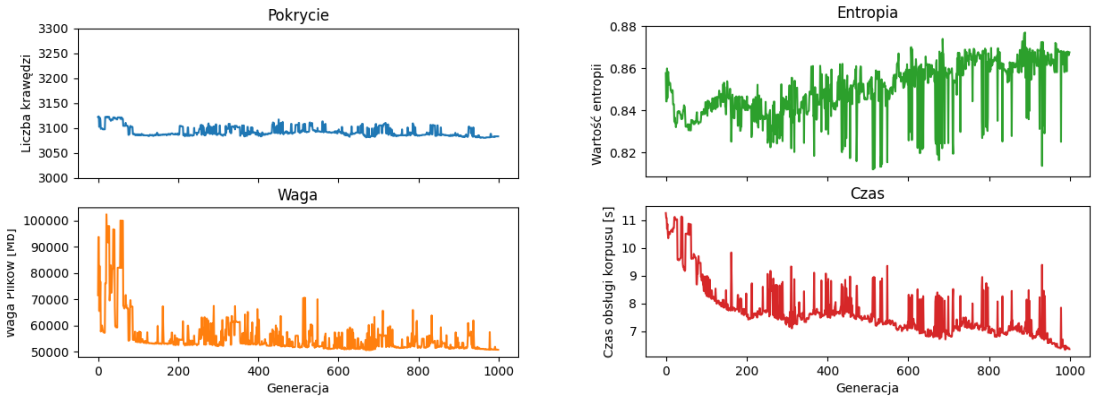
Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Mieszany	91394	0,85	25,27	3125
200	0,1	Mieszany	72939	0,85	22,69	3124
500	0,1	Mieszany	91949	0,86	11,67	3129
100	0,2	Mieszany	76132	0,86	23,57	3122
200	0,2	Mieszany	99425	0,86	27,77	3124
500	0,2	Mieszany	93755	0,84	11,08	3123
100	0,5	Mieszany	82648	0,85	22,71	3125
200	0,5	Mieszany	62485	0,86	22,81	3119
500	0,5	Mieszany	91949	0,86	11,67	3129

Tab. 11 Wyniki destylacji korpusu algorytmem epiVEGA plików tekstowych z operatorem mieszanym.

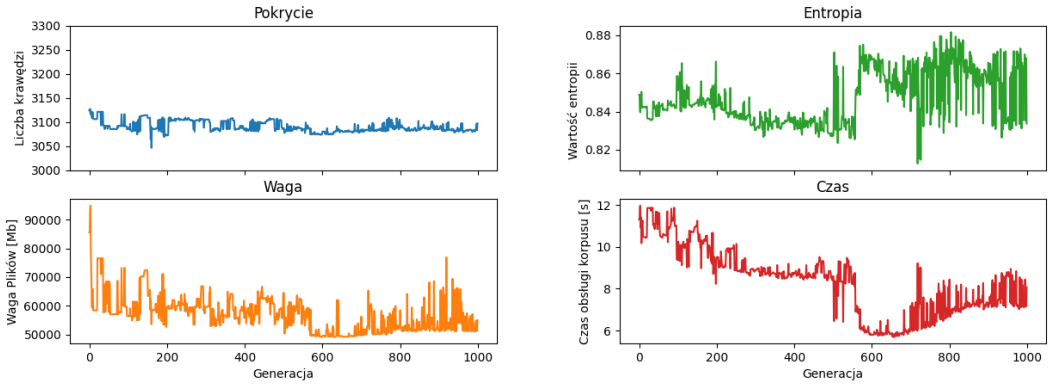
Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Mieszany	66,62	99,03	53,85	99,87
200	0,1	Mieszany	53,17	99,76	48,34	99,84
500	0,1	Mieszany	67,02	100,72	24,87	100,00
100	0,2	Mieszany	55,50	100,50	50,22	99,78
200	0,2	Mieszany	72,47	100,00	59,17	99,84
500	0,2	Mieszany	68,34	98,60	23,61	99,81
100	0,5	Mieszany	60,24	99,22	48,40	99,87
200	0,5	Mieszany	45,55	100,36	48,60	99,68
500	0,5	Mieszany	67,02	100,72	24,87	100,00

Tab. 12 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowych z operatorem mieszanym (wartości procentowe).

Zastosowanie histonu mieszanego pozwoliło na utrzymanie pełnego pokrycia w przypadku populacji liczącej 500 osobników przy $p_e = 0{,}1$ oraz $p_e = 0{,}5$. Otrzymane korpusy cechowały się taką samą redukcją rozmiaru, czasu obsługi oraz średnią entropią. Uzyskany czas obsługi cechuje się redukcją aż o ~75%. Proces destylacji korpusu danych testowych poskutkował również wzrostem średniej entropii w ostatecznym rozwiązaniu o 0,72%. Niski poziom redukcji entropii bądź jej pogorszenie jest spowodowane faktem, że pierwotny korpus cechuje się niską wariancją entropii, która wyniosła $\sigma^2 = 0{,}04$. Fakt ten znacznie utrudniał pracę algorytmu genetycznego w kierunku poprawy tego parametru.



Rys. 32 Przebieg procesu destylacji algorytmem epiVEGA korpusu plików tekstowych dla populacji o liczebności 500 osobników oraz p_e=0,1 dla operatora mieszanego.



Rys. 33 Przebieg procesu destylacji algorytmem epiVEGA korpusu plików tekstowych dla populacji o liczebności 500 osobników oraz p_e=0,5 dla operatora mieszanego.

Algorytm *epiVEGA* skutecznie uniemożliwił pogorszenie się stopnia pokrycia kodu biblioteki *libxml2 2.9.12*, przy jednoczesnej redukcji rozmiaru i czasu obsługi korpusu oraz nieznacznym pogorszeniu się stopnia średniej entropii. Jak widać na rysunkach nr 32 i nr 33, przebieg pracy algorytmu cechował się licznymi odchyleniami od trendu, które świadczą o reakcji mechanizmów sterowania zbieżnością na pogarszającą się jakości proponowanych przez algorytm pośrednich rozwiązań.

6.4.3 Pliki tekstowo–graficzne

6.4.3.1 Operator wyciszenia

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Wyciszający	375005756	0,95	11,62	480
200	0,1	Wyciszający	303855360	0,95	11,95	480
500	0,1	Wyciszający	405235530	0,95	11,96	480
100	0,2	Wyciszający	466291748	0,95	12,68	480
200	0,2	Wyciszający	548169528	0,94	12,31	480
500	0,2	Wyciszający	344680337	0,94	11,02	480
100	0,5	Wyciszający	317107563	0,94	11,78	480
200	0,5	Wyciszający	310782124	0,94	11,45	480
500	0,5	Wyciszający	276360246	0,94	10,89	480

Tab. 13 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowo–graficznych z operatorem wyciszającym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Wyciszający	49,34	99,85	43,29	100,00
200	0,1	Wyciszający	39,98	99,75	44,52	100,00
500	0,1	Wyciszający	53,32	99,96	44,53	100,00
100	0,2	Wyciszający	61,35	99,85	47,24	100,00
200	0,2	Wyciszający	72,12	98,85	45,85	100,00
500	0,2	Wyciszający	45,35	99,62	41,04	100,00
100	0,5	Wyciszający	41,72	99,48	43,88	100,00
200	0,5	Wyciszający	40,89	99,63	42,66	100,00
500	0,5	Wyciszający	36,36	99,33	40,56	100,00

Tab. 14 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowo–graficznych z operatorem wyciszającym (wartości procentowe).

Wykorzystanie operatora wyciszającego podczas destylacji korpusu plików tekstowo–graficznych pozwoliło przeprowadzić destylację z każdorazowym zachowaniem pierwotnego pokrycia. Zwiększanie prawdopodobieństwa wystąpienia modyfikacji epigenetycznej wpływało nieznacznie na poprawę redukcji pozostałych redukowanych parametrów.

6.4.3.2 Histon aktywujący

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Aktywujący	355736537	0,94	12,36	480
200	0,1	Aktywujący	409288704	0,94	12,22	480
500	0,1	Aktywujący	401448749	0,95	12,13	480
100	0,2	Aktywujący	418085730	0,95	12,75	480
200	0,2	Aktywujący	390188125	0,95	12,41	480
500	0,2	Aktywujący	348622566	0,94	12,21	480
100	0,5	Aktywujący	544758508	0,94	12,85	480
200	0,5	Aktywujący	357729536	0,95	11,39	480
500	0,5	Aktywujący	370758154	0,94	12,18	480

Tab. 15 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowo–graficznych z operatorem aktywującym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Aktywujący	46,81	99,40	46,02	100,00
200	0,1	Aktywujący	53,85	99,63	45,51	100,00
500	0,1	Aktywujący	52,82	99,80	45,18	100,00
100	0,2	Aktywujący	55,01	99,68	47,47	100,00
200	0,2	Aktywujący	51,34	99,73	46,22	100,00
500	0,2	Aktywujący	45,87	99,26	45,47	100,00
100	0,5	Aktywujący	71,68	99,42	47,88	100,00
200	0,5	Aktywujący	47,07	99,87	42,41	100,00
500	0,5	Aktywujący	48,78	99,61	45,35	100,00

Tab. 16 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowo–graficznych z operatorem aktywującym (wartości procentowe).

Wykorzystanie histonu aktywującego podczas destylacji korpusu plików tekstowo–graficznych również pozwoliło przeprowadzić destylację z każdorazowym zachowaniem pierwotnego pokrycia. Zwiększanie prawdopodobieństwa wystąpienia modyfikacji epigenetycznej nie wpływało znacząco na poprawę wartości pozostałych redukowanych parametrów.

6.4.3.3 Histon mieszany

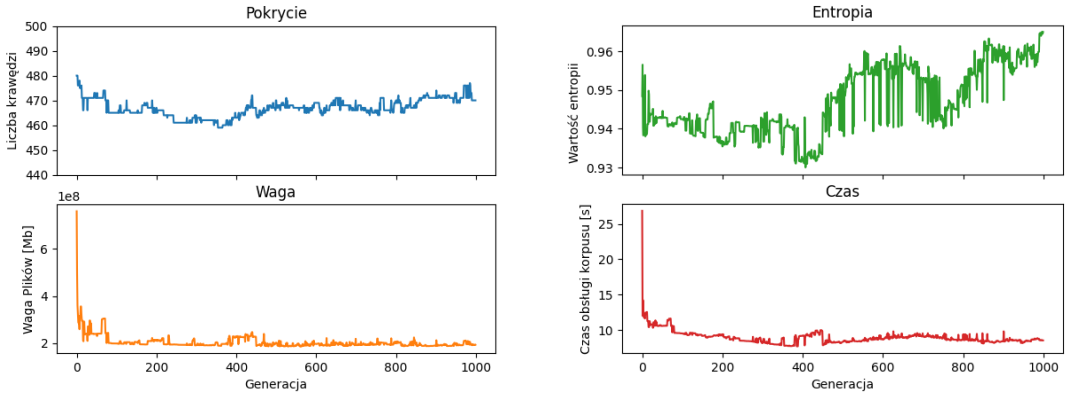
Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Mieszany	498124946	0,96	12,03	480
200	0,1	Mieszany	353287521	0,95	11,78	480
500	0,1	Mieszany	388299290	0,94	12,27	480
100	0,2	Mieszany	267065703	0,94	11,35	480
200	0,2	Mieszany	390188125	0,95	12,41	480
500	0,2	Mieszany	358731295	0,94	12,34	480
100	0,5	Mieszany	410707607	0,94	13,35	480
200	0,5	Mieszany	353944550	0,94	14,31	480
500	0,5	Mieszany	365763114	0,94	12,06	480

Tab. 17 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowo–graficznych z operatorem mieszanym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Mieszany	65,54	100,86	44,81	100,00
200	0,1	Mieszany	46,48	99,70	43,88	100,00
500	0,1	Mieszany	51,09	99,38	45,71	100,00
100	0,2	Mieszany	35,14	99,60	42,29	100,00
200	0,2	Mieszany	51,34	99,73	46,22	100,00
500	0,2	Mieszany	47,20	99,10	45,95	100,00
100	0,5	Mieszany	54,04	99,13	49,72	100,00
200	0,5	Mieszany	46,57	99,41	53,30	100,00
500	0,5	Mieszany	48,12	99,57	44,92	100,00

Tab. 18 Wyniki destylacji algorytmem epiVEGA korpusu plików tekstowo–graficznych z operatorem mieszanym (wartości procentowe).

Wykorzystanie histonu mieszanego podczas destylacji korpusu plików tekstowo–graficznych również pozwoliło przeprowadzić destylację z każdorazowym zachowaniem pierwotnego pokrycia. Zastosowanie tego operatora dla populacji o rozmiarze 100 osobników z $p_e = 0{,}2$ zapewniło uzyskanie najmniejszego korpusu.



Rys. 34 Przebieg procesu destylacji korpusu plików tekstowo–graficznych algorytmem epiVEGA dla populacji o liczebności 500 osobników oraz p_e =0,2 dla operatora mieszanego.

Algorytm *epiVEGA* skutecznie pozwolił na wyznaczenie korpusu, który zachował pierwotne pokrycie przy jednoczesnej redukcji rozmiaru, czasu obsługi oraz średniej entropii zbioru przypadków testowych. Rysunek nr 34 przedstawia przebieg pracy algorytmu genetycznego. Znacząca redukcja rozmiaru oraz średniego czasu obsługi programu nastąpiła już w pierwszych przetwarzanych przez algorytmach pokoleniach. Wskazywać to może na dużą liczbę redundantnych przypadków testowych w pierwotnym korpusie.

6.4.4 Pliki skompresowane (LZ4)

6.4.4.1 Histon wyciszający

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Wyciszający	12144623	0,89	12,73	390
200	0,1	Wyciszający	14250975	0,89	12,19	390
500	0,1	Wyciszający	13852012	0,90	12,44	390
100	0,2	Wyciszający	13748615	0,89	13,13	390
200	0,2	Wyciszający	15002714	0,89	12,18	390
500	0,2	Wyciszający	13995330	0,89	12,53	390
100	0,5	Wyciszający	14540676	0,89	13,24	390
200	0,5	Wyciszający	17105918	0,89	12,87	390
500	0,5	Wyciszający	13244556	0,89	13,23	390

Tab. 19 Wyniki destylacji korpusu plików skompresowanych algorytmem epiVEGA z operatorem wyciszającym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Wyciszający	58,61	99,35	46,53	100,00
200	0,1	Wyciszający	68,77	99,52	44,52	100,00
500	0,1	Wyciszający	66,85	101,01	45,43	100,00
100	0,2	Wyciszający	66,35	99,29	47,98	100,00
200	0,2	Wyciszający	72,40	99,22	44,49	100,00
500	0,2	Wyciszający	67,54	99,26	45,78	100,00
100	0,5	Wyciszający	70,17	99,84	48,38	100,00
200	0,5	Wyciszający	82,55	99,27	47,04	100,00
500	0,5	Wyciszający	63,92	99,41	48,32	100,00

Tab. 20 Wyniki destylacji korpusu plików skompresowanych algorytmem epiVEGA z operatorem wyciszającym (wartości procentowe).

Wykorzystanie operatora wyciszającego podczas destylacji korpusu plików skompresowanych pozwoliło przeprowadzić destylację z każdorazowym zachowaniem pierwotnego pokrycia. Zwiększanie prawdopodobieństwa wystąpienia modyfikacji epigenetycznej wpływało znacząco na poprawę redukcji pozostałych redukowanych parametrów.

6.4.4.2 Histon aktywujący

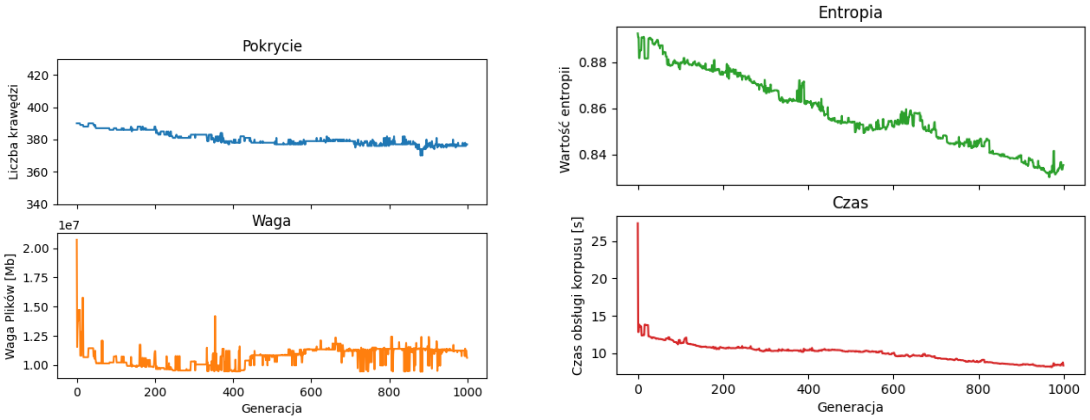
Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Aktywujący	13921721	0,88	13,73	390
200	0,1	Aktywujący	15195958	0,86	9,30	390
500	0,1	Aktywujący	14994359	0,85	8,95	390
100	0,2	Aktywujący	15710027	0,88	12,34	390
200	0,2	Aktywujący	13157422	0,88	12,44	390
500	0,2	Aktywujący	15222350	0,85	8,73	390
100	0,5	Aktywujący	13732189	0,88	11,93	390
200	0,5	Aktywujący	15361873	0,88	9,63	390
500	0,5	Aktywujący	11439838	0,89	11,84	390

Tab. 21 Wyniki destylacji algorytmem epiVEGA korpusu plików skompresowanych z operatorem aktywującym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Aktywujący	67,19	99,00	50,17	100,00
200	0,1	Aktywujący	73,33	96,55	34,00	100,00
500	0,1	Aktywujący	72,36	95,33	32,70	100,00
100	0,2	Aktywujący	75,82	98,38	45,10	100,00
200	0,2	Aktywujący	63,50	98,46	45,45	100,00
500	0,2	Aktywujący	73,46	95,06	31,88	100,00
100	0,5	Aktywujący	66,27	98,51	43,60	100,00
200	0,5	Aktywujący	74,14	98,81	35,20	100,00
500	0,5	Aktywujący	55,21	100,21	43,26	100,00

Tab. 22 Wyniki destylacji algorytmem epiVEGA korpusu plików skompresowanych z operatorem aktywującym (wartości procentowe).

Wykorzystanie histonu aktywującego w algorytmie *epiVEGA* pozwoliło na zachowanie pierwotnego pokrycia, przy jednoczesnej redukcji rozmiaru i czasu obsługi korpusu plików skompresowanych dla każdej z konfiguracji. Zwiększyła się także redukcja średniej entropi, która dla $p_e = 0{,}1$ i $p_e = 0{,}2$ wyniosła nawet ~5%. Oceniając korpusy według ustalonej kolejności rozpatrywania kryteriów opisywana konfiguracja pozwoliła uzyskać najlepszy zbiór, który cechował się 45% redukcją wagi i 56% redukcją czasu. Zostało to okupione wzrostem o 0,21% średniej entropi.



Rys. 35 Przebieg procesu destylacji algorytmem epiVEGA korpusu plików skompresowanych dla populacji o liczebności 500 osobników oraz p_e = 0,5 dla operatora aktywującego.

6.4.4.3 Operator mieszany

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	0,1	Mieszany	14119535	0,89	12,98	390
200	0,1	Mieszany	13860672	0,89	12,74	390
500	0,1	Mieszany	11749213	0,87	11,98	390
100	0,2	Mieszany	12483842	0,90	13,23	390
200	0,2	Mieszany	14104314	0,89	12,80	390
500	0,2	Mieszany	13008655	0,89	12,93	390
100	0,5	Mieszany	16455836	0,89	13,56	390
200	0,5	Mieszany	12794248	0,88	10,04	390
500	0,5	Mieszany	14095235	0,88	12,22	390

Tab. 23 Wyniki destylacji algorytmem epiVEGA korpusu plików skompresowanych z operatorem mieszanym.

Wielkość populacji	Prawdopodobieństwo wystąpienia operatora epigenetycznego	Rodzaj operatora	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	0,1	Mieszany	68,14	99,68	47,42	100,00
200	0,1	Mieszany	66,89	99,40	46,54	100,00
500	0,1	Mieszany	56,70	98,00	43,78	100,00
100	0,2	Mieszany	60,25	100,52	48,35	100,00
200	0,2	Mieszany	68,06	99,74	46,75	100,00
500	0,2	Mieszany	62,78	99,20	47,25	100,00
100	0,5	Mieszany	79,41	99,26	49,54	100,00
200	0,5	Mieszany	61,74	98,72	36,68	100,00
500	0,5	Mieszany	68,02	98,78	44,64	100,00

Tab. 24 Wyniki destylacji algorytmem epiVEGA korpusu plików skompresowanych z operatorem mieszanym (wartości procentowe).

Algorytm *epiVEGA* z zastosowanym histonem mieszanym skutecznie uniemożliwił pogorszenie się stopnia pokrycia kodu biblioteki *lz4 1.0.0*, przy jednoczesnej redukcji rozmiaru, czasu obsługi i średniej entropii (poza konfiguracją w której populacja liczyła 100 osobników a p_e = 0,2).

6.5 Wyniki destylatorów cmin, *VEGA*

W czasie eksperymentów algorytm *VEGA* uruchamiano z prawdopodobieństwem krzyżowania i mutacji takim samym jak w przypadku algorytmu *epiVEGA*, które wyniosły odpowiednio $p_c = 0{,}5$ i $p_m = 0{,}2$. Jedynym zmiennym parametrem pracy algorytmu był rozmiar populacji (100, 200 i 500 osobników).

Algorytm *cmin* uruchomiono w konfiguracji:

afl–cmin –i pierwotny_koprus/ –o katalog_wyjsciowy/ –m none — ./a.out @@

6.5.1 VEGA

6.5.1.1 Pliki graficzne

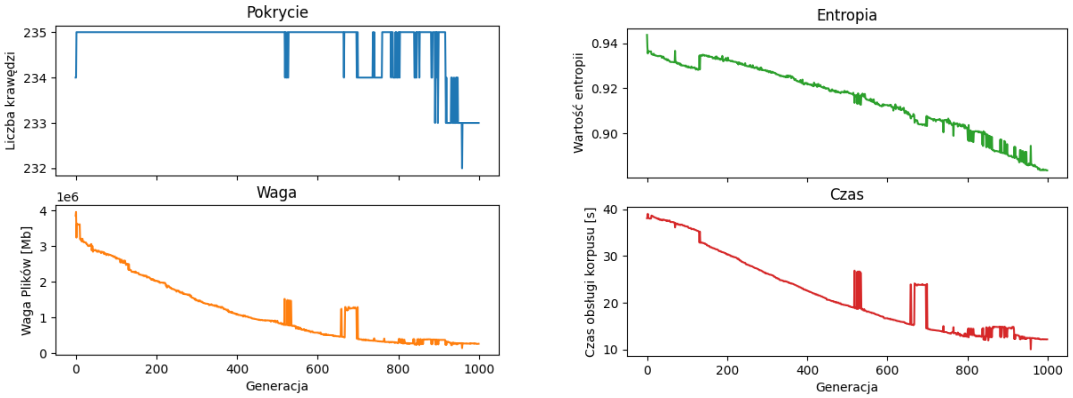
Wielkość populacji	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	1374696	0,90	21,91	234
200	1914361	0,93	30,00	235
500	256101	0,90	12,64	235

Tab. 25 Wyniki destylacji algorytmem VEGA korpusu plików graficznych.

Wielkość populacji	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	15,92	95,32	23,39	99,57
200	22,16	98,50	32,03	100,00
500	2,97	95,32	13,49	100,00

Tab. 26 Wyniki destylacji algorytmem VEGA korpusu plików graficznych (wartości procentowe).

Destylacja algorytmem *VEGA* korpusu plików graficznych pozwoliła na utrzymanie pierwotnego pokrycia, przy redukcji pozostałych rozpatrywanych kryteriów dla populacji liczących 200 i 500 osobników.



Rys. 36 Przebieg destylacji algorytmem VEGA korpusu plików graficznych.

Jak widać na rysunku nr 36, algorytm *VEGA* utrzymał pełne pokrycie biblioteki *giflib 2.5.2* przez niemal 900 pokoleń, przy ciągłej redukcji minimalizowanych kryteriów.

6.5.1.2 Pliki tekstowe

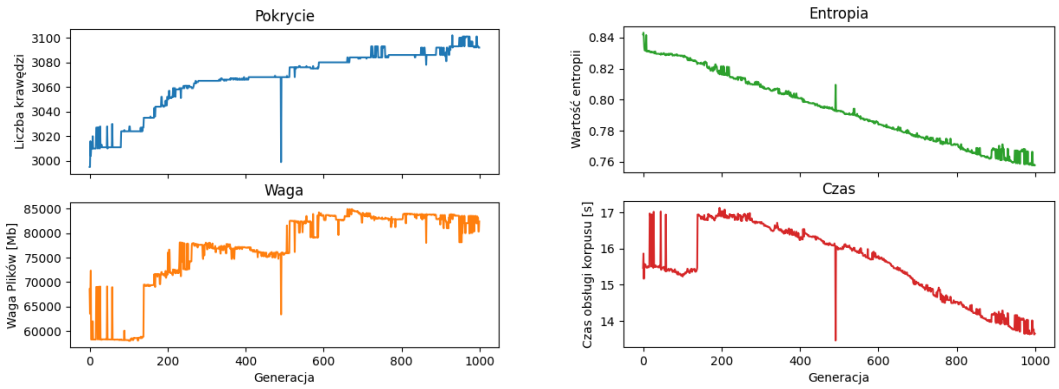
Wielkość populacji	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	81972	0,79	14,92	3057
200	71883	0,84	15,08	3064
500	82983	0,77	14,21	3102

Tab. 27 Wyniki destylacji algorytmem VEGA korpusu plików tekstowych.

Wielkość populacji	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	59,75	92,27	31,79	97,70
200	52,40	98,11	32,13	97,92
500	60,49	89,93	30,28	99,14

Tab. 28 Wyniki destylacji algorytmem VEGA korpusu plików tekstowych (wartości procentowe).

Destylacja algorytmem *VEGA* korpusu plików tekstowych nie pozwoliła na utrzymanie pierwotnego pokrycia dla wszystkich zadanych rozmiarów populacji.



Rys. 37 Przebieg destylacji algorytmem VEGA korpusu plików tekstowych.

Na rysunku nr 37 widać zauważalną korelację między zwiększającym się wraz z liczbą przetworzonych pokoleń pokryciem, a rosnącym rozmiarem destylowanego korpusu. Pomimo tej tendencji – średnia entropia oraz czas obsługi korpusu były skutecznie redukowane w trakcie pracy algorytmu genetycznego.

6.5.1.3 Pliki tekstowo–graficzne

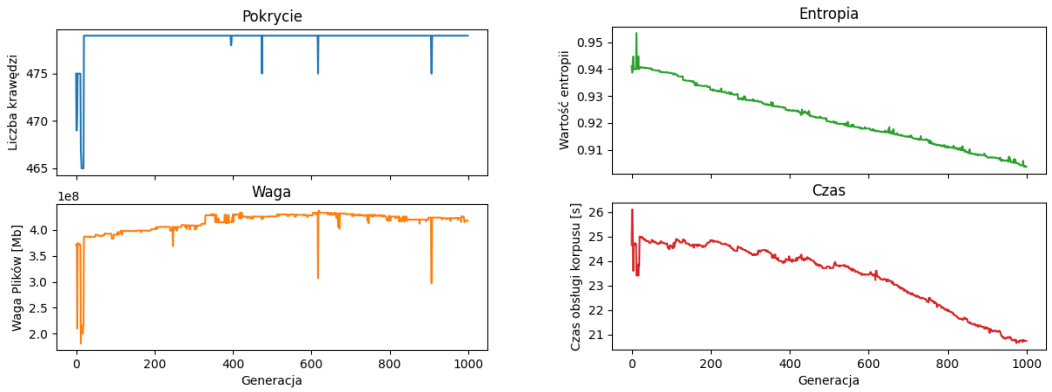
Wielkość populacji	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	319646816	0,95	23,33	475
200	419633573	0,91	20,65	479
500	453095905	0,94	24,08	477

Tab. 29 Wyniki destylacji algorytmem VEGA korpusu plików tekstowo–graficznych.

Wielkość populacji	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	42,06	100,17	86,89	98,96
200	55,21	95,95	76,91	99,79
500	59,62	99,11	89,69	99,38

Tab. 30 Wyniki destylacji algorytmem VEGA korpusu plików tekstowo–graficznych (wartości procentowe).

Destylacja algorytmem *VEGA* korpusu plików tekstowo–graficznych nie pozwoliła na utrzymanie pierwotnego pokrycia dla wszystkich zadanych rozmiarów populacji, tracąc mniej niż 1% krawędzi w grafie przepływu sterowania.



Rys. 38 Przebieg destylacji algorytmem VEGA korpusu plików tekstowo–graficznych.

Na rysunku nr 38 widać zauważalną korelację między zwiększającym się wraz z liczbą przetworzonych pokoleń pokryciem, a rosnącym rozmiarem destylowanego korpusu plików tekstowo–graficznych. Pomimo tego trendu – średnia entropia oraz czas obsługi korpusu były skutecznie redukowane w trakcie pracy algorytmu *VEGA*.

6.5.1.4 Pliki skompresowane

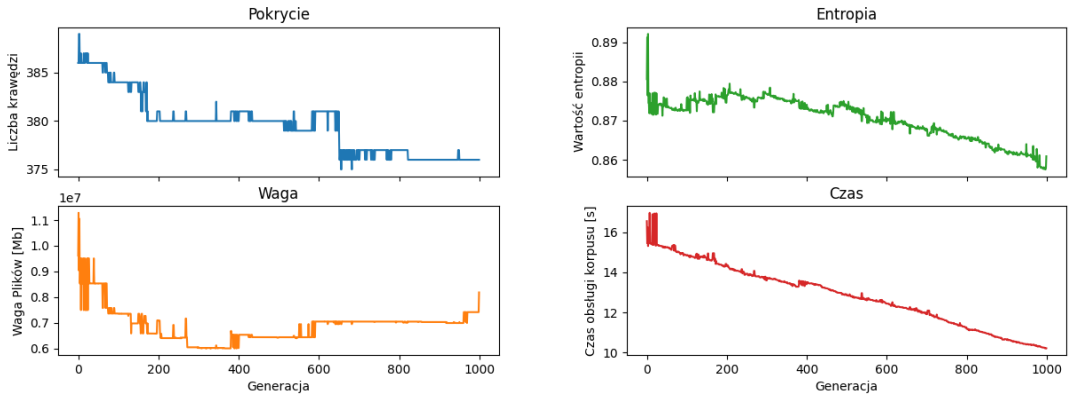
Wielkość populacji	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
100	10770437	0,88	10,95	385
200	9482770	0,88	15,41	387
500	11052969	0,89	15,30	389

Tab. 31 Wyniki destylacji algorytmem VEGA korpusu plików skompresowanych.

Wielkość populacji	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
100	51,98	98,59	40,01	98,72
200	45,76	98,59	56,30	99,23
500	53,34	99,71	55,90	99,74

Tab. 32 Wyniki destylacji algorytmem VEGA korpusu plików skompresowanych (wartości procentowe).

Destylacja algorytmem *VEGA* korpusu plików skompresowanych nie pozwoliła na utrzymanie pierwotnego pokrycia dla wszystkich zadanych rozmiarów populacji, jednak straty w pokryciu kodu wynosiły mniej niż 1% wszystkich krawędzi w grafie przepływu sterowania.



Rys. 39 Przebieg destylacji algorytmem VEGA korpusu plików skompresowanych.

Na rysunku nr 39 widać, że algorytm nie potrafił utrzymać wstępnie uzyskanego pokrycia. Redukcja wszystkich rozpatrywanych wag pogorszała liczbę aktywowanych krawędzi w grafie przepływu sterowania oprogramowania LZ4.

6.5.2 Destylator cmin

Format plików	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
*.gif	842788	0,93	11,03	234
*.xml	104323	0,85	9,59	3112
*.pdf	13736702	0,96	4,58	457
*.lz4	16412707	0,86	2,18	366

Tab. 33 Wyniki destylacji algorytmem cmin.

Wielkość populacji	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
*.gif	9,76	98,83	11,77	99,57
*.xml	76,04	98,92	20,44	99,46
*.pdf	1,81	101,13	17,07	95,21
*.lz4	79,00	96,35	7,96	94,00

Tab. 34 Wyniki destylacji algorytmem cmin (wartości procentowe).

Algorytm *cmin* uwzględnia wyłącznie rozmiar plików i pokrycie kodu jako kryteria w procesie destylacji korpusu. Żaden z uzyskanych korpusów nie cechował się zachowaniem pełnego pokrycia, chociaż w przypadku plików tekstowych i graficznych utrata pokrycia wyniosła tylko ~0,5%. Pomimo braku rozpatrywania czasu obsługi jako kryterium destylacji, zauważalna jest dość wysoka jego redukcja wynosząca ~80–90%. Ma to prawdopodobnie

związek z wstępnym wykluczeniem większych plików, które ze względu na swój rozmiar są usuwane z kolejki pierwotnych plików jeszcze przed uruchomieniem procesu destylacji.

6.6 Porównanie korpusów wynikowych

W celu dokonania porównania (wg. ustalonej w pkt. 5.5 kolejności kryteriów) korpusów uzyskanych algorytmami *VEGA* i *epiVEGA* wybrano najlepsze zredukowane zbiory spośród wszystkich uzyskanych w pierwszej fazie eksperymentów. Wyniki te zestawiono w tabelach porównujących zbiory testowe poszczególnych formatów plików, razem z korpusami pełnymi i uzyskanymi algorytmem *cmin*.

6.6.1 Pliki graficzne

Algorytm	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
Pełen Korpus	8637268	0,94	93,67	235
cmin	842788	0,93	11,03	234
VEGA	256101	0,90	12,64	235
epi*VEGA*	197819	0,87	11,53	235

Tab. 35 Wyniki destylacji plików graficznych.

Wielkość populacji	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
Pełen Korpus	100,00	100,00	100,00	100,00
cmin	9,76	98,83	11,77	99,57
VEGA	2,97	95,15	13,49	100,00
epi*VEGA*	2,29	92,04	12,31	100,00

Tab. 36 Wyniki destylacji plików graficznych (wartości procentowe).

Algorytm *epiVEGA* wyznaczył korpus plików graficznych o pełnym pokryciu krawędzi, najmniejszej wadze oraz najniższej średniej entropii. Zaproponowane rozwiązanie ustąpiło algorytmowi *cmin* tylko pod kątem czasu obsługi wszystkich przypadków testowych, uzyskując wynik lepszy od algorytmu *VEGA*.

6.6.2 Pliki tekstowe

Algorytm	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
Pełen Korpus	137187	0,86	46,93	3129
cmin	104323	0,85	9,59	3112
VEGA	82983	0,77	14,22	3102
epi*VEGA*	91949	0,86	11,67	3129

Tab. 37 Wyniki destylacji plików tekstowych.

Wielkość populacji	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
Pełen Korpus	100,00	100,00	100,00	100,00
cmin	76,04	98,92	20,44	99,46
VEGA	60,49	89,76	30,30	99,14
epi*VEGA*	67,02	100,72	24,87	100,00

Tab. 38 Wyniki destylacji plików tekstowych (wartości procentowe).

Zaproponowany algorytm *epiVEGA* jako jedyny wyznaczył korpus danych testowych, który zapewnił pełne pokrycie krawędzi w grafie przepływu sterowania biblioteki obsługującej pliki tekstowe. Wynik ten został osiągnięty kosztem pozostałych kryteriów – uzyskano korpus o pogorszonej średniej entropii oraz czasie obsługi, który jest dłuższy niż w przypadku algorytmu *cmin*. Biorąc pod uwagę fakt, iż celem problemu *MCSCP* jest uzyskanie pełnego pokrycia, tylko algorytm *epiVEGA* spełnił ten warunek.

6.6.3 Pliki tekstowo–graficzne

Algorytm	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
Pełen Korpus	760027749	0,95	26,85	480
Cmin	13736702	0,96	4,58	457
VEGA	419633573	0,91	20,65	479
epi*VEGA*	267065703	0,94	11,35	480

Tab. 39 Wyniki destylacji plików tekstowo–graficznych.

Wielkość populacji	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
Pełen Korpus	100,00	100,00	100,00	100,00
Cmin	1,81	101,13	17,07	95,21
VEGA	55,21	95,44	76,92	99,79
epi*VEGA*	35,14	99,60	42,29	100,00

Tab. 40 Wyniki destylacji plików tekstowo–graficznych (wartości procentowe).

Algorytm *epiVEGA* jako jedyny wyznaczył korpus plików tekstowo–graficznych o pełnym pokryciu krawędzi. Ustąpił algorytmowi *VEGA* tylko pod kątem średniej entropii zbioru przypadku testowych, natomiast algorytm *cmin* zaproponował zbiór o mniejszej wadze oraz krótszym czasie obsługi. Ponownie, po uwzględnieniu faktu, iż celem problemu *MCSCP* jest uzyskanie pełnego pokrycia, ponownie tylko algorytm *epiVEGA* spełnił ten warunek.

6.6.4 Pliki skompresowane

Algorytm	Waga korpusu [B]	Entropia uproszczona	Czas obsługi korpusu [s]	Liczba pokrytych krawędzi
Pełen Korpus	20721467	0,89	27,37	390
cmin	16412707	0,86	2,18	366
VEGA	11052969	0,89	15,30	389
epi*VEGA*	11439838	0,89	11,84	390

Tab. 41 Wyniki destylacji plików skompresowanych.

Wielkość populacji	Waga korpusu [%]	Entropia uproszczona [%]	Czas obsługi korpusu [%]	Liczba pokrytych krawędzi [%]
Pełen Korpus	100,00	100,00	100,00	100,00
Cmin	79,20	96,35	7,96	93,85
VEGA	53,34	99,71	55,90	99,74
epi*VEGA*	55,2	100,2	43,26	100

Tab. 42 Wyniki destylacji plików skompresowanych (wartości procentowe).

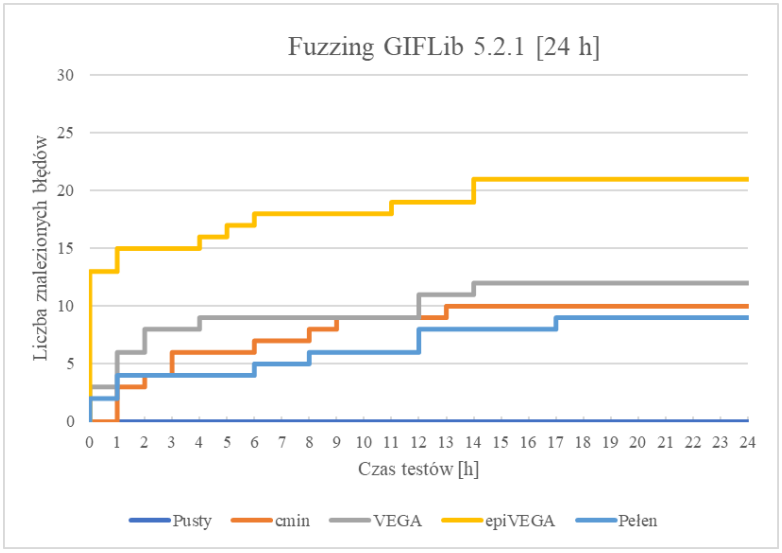
Algorytm *epiVEGA* po raz kolejny jako jedyny wyznaczył korpus plików tekstowo–graficznych o pełnym pokryciu krawędzi, spełniając główne wymaganie dla rozwiązania problemu *MCSCP*.

6.7 Fuzzing – etap eksperymentalny

W celu dalszego porównania jakości uzyskanych w procesie destylacji zredukowanych zbiorów, w drugiej części eksperymentów przeprowadzono *fuzzing* z wykorzystaniem korpusu pełnego (który nie został poddany destylacji), korpusu pustego (nie zawierającego przypadków testowych) oraz tych wyznaczonych algorytmami *cmin, VEGA* i *epiVEGA*. Zgodnie z przyjętymi standardami [66], w czasie fuzzingu jako główne kryterium porównania przyjęto liczbę znalezionych błędów. Kryteriami pomocniczymi były unikalne zawieszenia programu, liczba nowych znalezionych ścieżek oraz liczba wszystkich znalezionych ścieżek w grafie przepływu sterowania. Dobór kryteriów pomocniczych podyktowany był danymi, które można uzyskać podczas fuzzingu z wykorzystaniem oprogramowania *AFL++* (wynikowe pliki *full_fuzzer_stats* i *plot_data*). Warto nadmienić, iż generowanie przypadków testowych powodujących zawieszenie pracy programu nie jest celem fuzzingu. Pliki takie mogą zostać wykorzystane do potencjalnej złośliwej aktywności (zawieszenie pracy aplikacji), więc są uwzględniane w statystykach testów. Nie stanowią jednak podstawy do ewaluacji skuteczności fuzzerów i jakości korpusów danych testowych. Liczba znalezionych ścieżek oraz liczba wszystkich znalezionych ścieżek w grafie przepływu sterowania są zaś pomocne głównie w procesie zarządzania korpusem, co pozwala na zwiększenie efektywności testów w przyszłości.

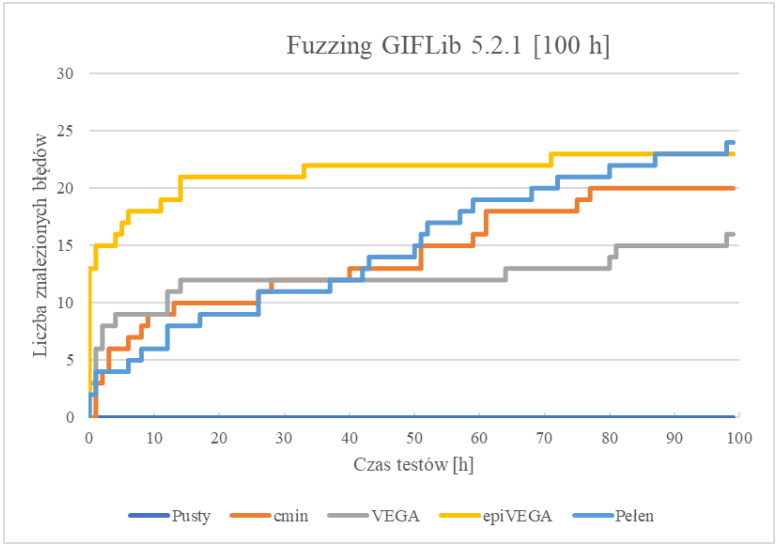
Fuzzing przeprowadzono dla każdego z korpusów przez 100 godzin (łączny czas testów to 2000 godzin), z wyszczególnieniem pierwszych 24 godzin. Czas trwania testów równy jednej dobie uznaje się za minimalny czas przeprowadzania fuzzingu [66]. Ponad czterokrotne wydłużenie trwania testów miało na celu uwypuklenie faktu, że destylacja zbioru danych testowych usprawnia *fuzzing* poprzez redukcje czasu niezbędnego do znalezienia potencjalnych podatności. Testy z wykorzystaniem pierwotnego korpusu powinny ostatecznie zwracać analogiczne wyniki, jednak ich uzyskanie zajmować może znacznie więcej czasu.

6.7.1 Fuzzing biblioteki przetwarzającej pliki graficzne



Rys. 40 Liczba znalezionych unikalnych błędów w bibliotece GIFLib 5.2.1 w ciągu pierwszych 24 godzin fuzzingu.

W ciągu pierwszych 24 godzin fuzzingu biblioteki *GIFLib 5.2.1* nie wykryto błędów wyłącznie z wykorzystaniem zbioru pustego. Korpus wyznaczony algorytmem *epiVEGA* pozwolił na wykrycie 21 błędów, co było wynikiem znacznie wyższym niż w przypadku zastosowania algorytmów *VEGA* i *cmin* (odpowiednio 12 i 10 znalezionych błędów).

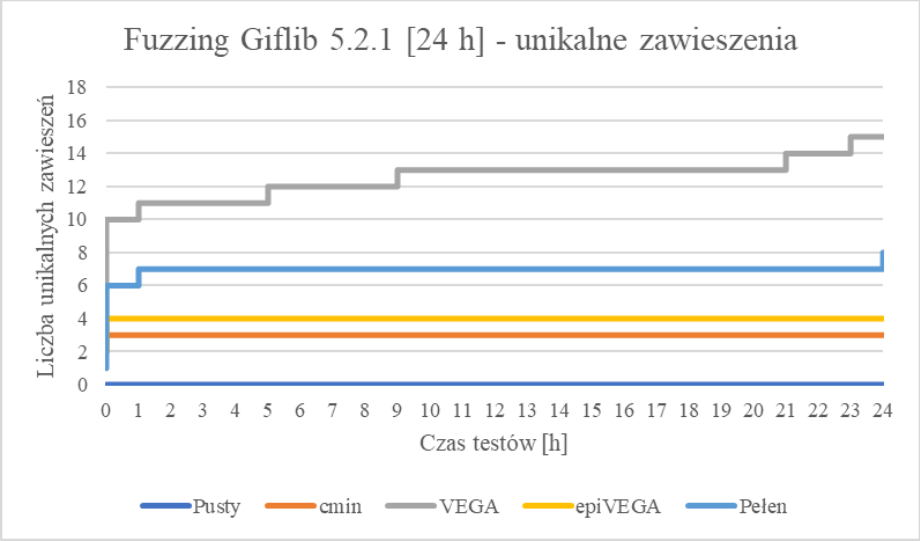


Rys. 41 Liczba znalezionych unikalnych błędów w bibliotece GIFLib 5.2.1 w ciągu 100 godzin fuzzingu.

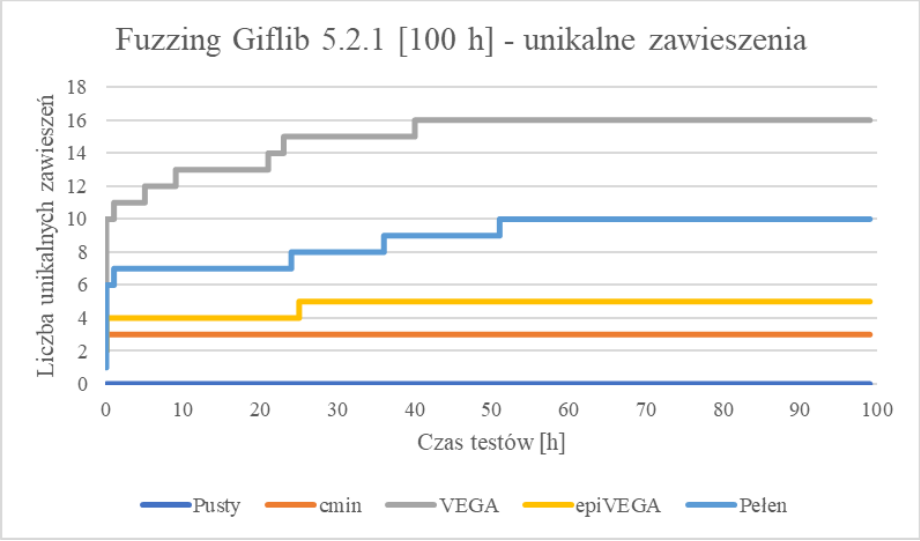
Kolejne 76 godzin testów nie zmieniły wyżej opisanej tendencji. Główna zmiana dotknęła wyników uzyskanych z wykorzystaniem zbioru pełnego. W pierwszej dobie testów

z jego wykorzystaniem znaleziono 9 błędów, jednak liczba ta systematycznie rosła w czasie aż do osiągnięcia 24 potencjalnych podatności, co okazało się najlepszym wynikiem.

Zbiory uzyskane w procesie destylacji pozwoliły na szybsze wykrywanie znacznej ilości błędów, co jest zgodne z celem redukcji zbioru danych testowych. Jej głównym zadaniem jest znajdowanie błędów szybciej niż w przypadku fuzzingu wykorzystującego pełen korpus. Korpus wyznaczony algorytmem *epiVEGA* pozwolił na znalezienie tylko jednego błędu mniej, co okazało się wynikiem znacznie lepszym niż w przypadku pozostałych algorytmów.



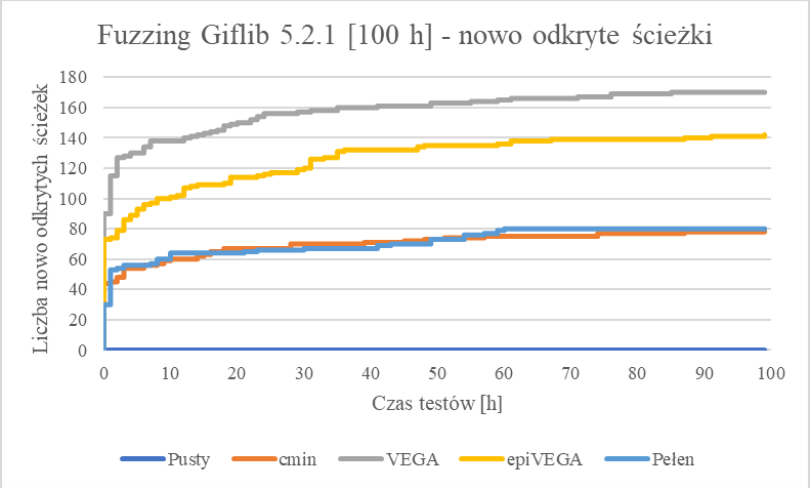
Rys. 42 Liczba znalezionych unikalnych zawieszeń w bibliotece GIFLib 5.2.1 w ciągu pierwszych 24 godzin fuzzingu.



Rys. 43 Liczba znalezionych unikalnych zawieszeń w bibliotece GIFLib 5.2.1 w ciągu 100 godzin fuzzingu.

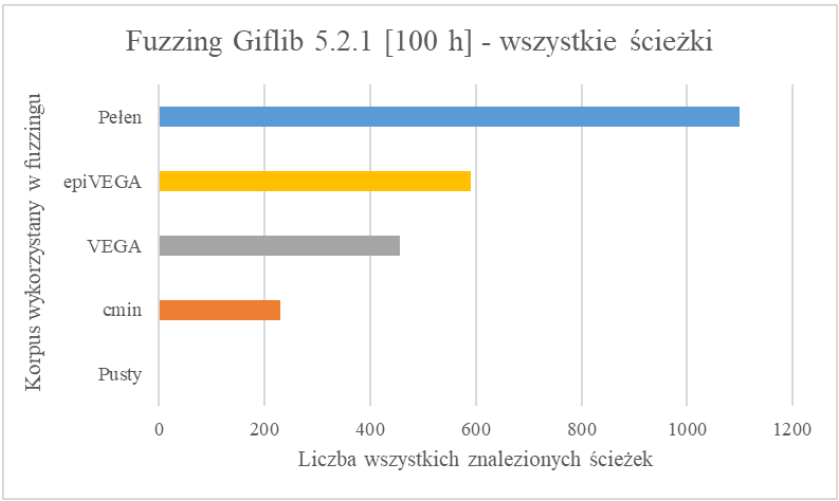
Zarówno w pierwszej dobie testów, jak i w dalszym ich etapie, najwięcej unikalnych zawieszeń uzyskano podczas fuzzingu z wykorzystaniem algorytmu *VEGA*. Zaproponowane

algorytm *epiVEGA* uplasował się na trzecim miejscu, między zbiorem pełnym, a destylatorem *cmin*.



Rys. 44 Liczba znalezionych nowych ścieżek w bibliotece GIFLib 5.2.1 w ciągu 100 godzin fuzzingu.

Fuzzer *AFL++* określa efektywność i stopień wydajności testów poprzez liczbę aktywowanych ścieżek w grafie przepływu sterowania, a nie liczby aktywowanych krawędzi. W tym przypadku najwięcej nowych krawędzi uzyskano z wykorzystaniem korpusu wyznaczonym *VEGA*, a algorytm *epiVEGA* uplasował się na drugiej pozycji. Korpus pełny oraz ten uzyskany za pomocą algorytmu *cmin* uzyskały zbliżone do siebie wyniki. Warto zwrócić uwagę na fakt, iż fuzzer wykorzystując pusty był niezdolny do wygenerowania przypadków testowych, które mogły pokryć jakąkolwiek ścieżkę, co sugeruje, że nie były one w stanie przejść procesu walidacji plików wejściowych.

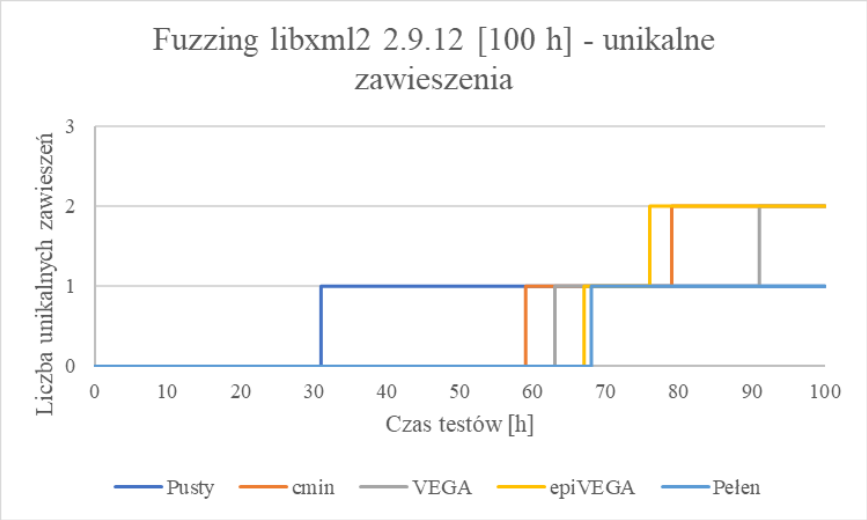


Rys. 45 Liczba znalezionych wszystkich ścieżek w bibliotece GIFLib 5.2.1 w ciągu 100 godzin fuzzingu.

Na końcu testów zsumowaniu podlegają nowo odkryte ścieżki wraz z tymi, które są wzbudzane przez wykorzystywany korpus. Algorytm *epiVEGA* wyznaczył zbiór, za pomocą którego aktywowano najwięcej ścieżek wśród korpusów uzyskanych w procesie destylacji.

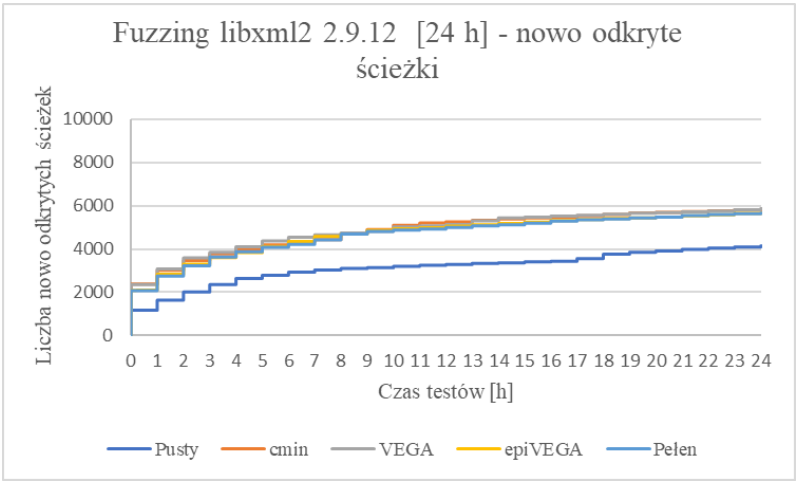
6.7.2 Fuzzing biblioteki przetwarzającej pliki tekstowe

Fuzzing biblioteki *libxml2 2.9.1* z wykorzystaniem wszystkich wybranych korpusów nie pozwolił na znalezienie żadnych błędów.



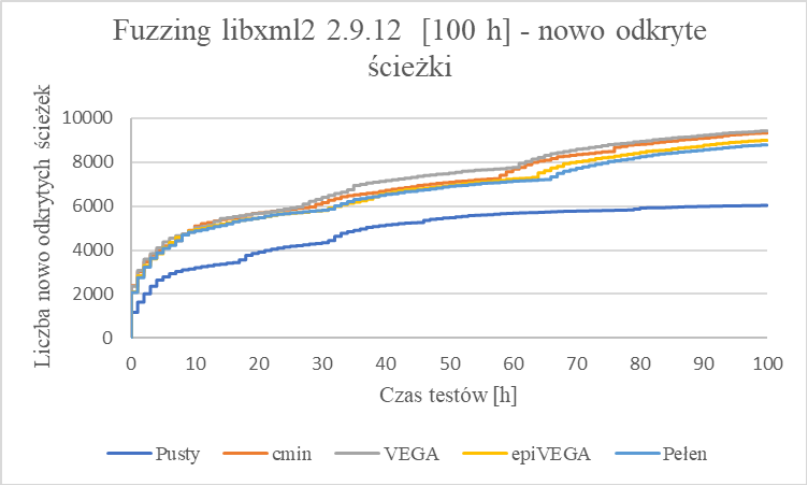
Rys. 46 Liczba znalezionych unikalnych zawieszeń w bibliotece libxml2 2.9.12 w ciągu 100 godzin fuzzingu.

Pierwsze unikalne zawieszenia zostało wykryte się dopiero w 31 godzinie testów. Wszystkie korpusy wyznaczone w procesie destylacji pozwoliły na znalezienie dwóch unikalnych zawieszeń. Wykorzystanie zbiorów pustego i pełnego poskutkowało znalezieniem jednego unikalnego zawieszenia.



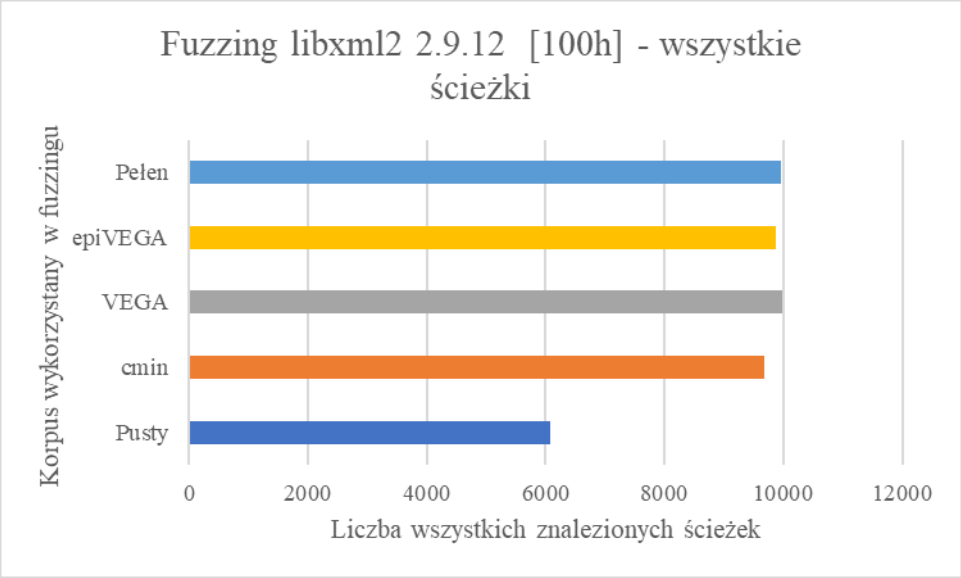
Rys. 47 Liczba znalezionych nowych ścieżek w bibliotece libxml2 2.9.12 w ciągu pierwszych 24 godzin fuzzingu.

Pierwsze 24 godziny testu poskutkowały znalezieniem ok. 6000 ścieżek przez korpusy przedestylowane i zbiór pełny oraz ok. 4000 ścieżek z pomocą zbioru pustego. Biblioteka *libxml2* obsługuje pliki posiadające mniej skomplikowaną strukturę, co pozwoliło na wygenerowanie przez pusty korpus stosunkowo poprawnych przypadków testowych.



Rys. 48 Liczba znalezionych nowych ścieżek w bibliotece libxml2 2.9.12 w ciągu 100 godzin fuzzingu.

Zakończenie testów poskutkowało wyznaczeniem ok 9000 ścieżek przez *fuzzer* wykorzystujący korpusy przedestylowane oraz korpus pełen. *Fuzzing* bazujący na zbiorze pustym pozwolił na aktywację ok. 6000 ścieżek.



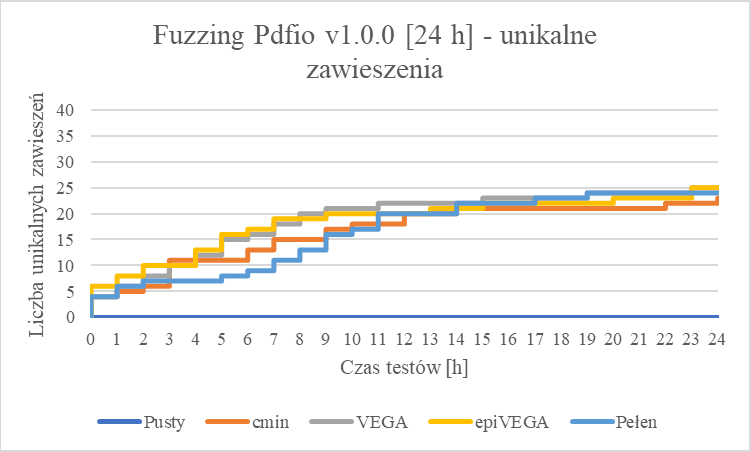
Rys. 49 Liczba znalezionych wszystkich ścieżek w bibliotece libxml2 5.2.1 w ciągu 100 godzin fuzzingu.

Korpusy otrzymane w procesie destylacji algorytmami *epiVEGA* i *VEGA* oraz korpus pierwotny pozwoliły na pokrycie ok 10 000 ścieżek. Nieznacznie ustąpił im zbiór testowy

wyznaczony przez algorytm *cmin*. Korpus pusty wygenerował przypadki testowe które pozwoliły pokryć ponad 6000 ścieżek.

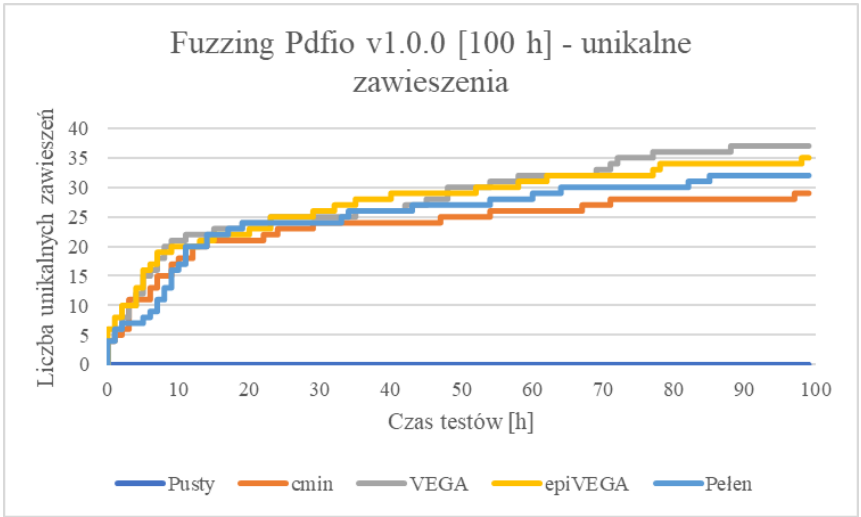
6.7.3 Fuzzing biblioteki przetwarzającej pliki tekstowo–graficzne

Fuzzing biblioteki *pdfdio 1.0.0* z wykorzystaniem wszystkich wybranych korpusów nie pozwolił na znalezienie żadnych błędów.



Rys. 50 Liczba znalezionych nowych ścieżek w bibliotece Pdfio v1.0.0 w ciągu pierwszych 24 godzin fuzzingu.

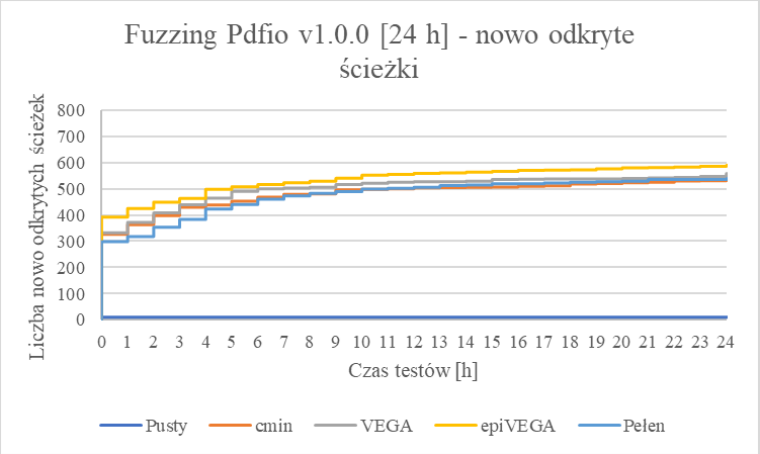
Niepuste korpusy (zbiór pełen oraz te uzyskane algorytmami *cmin*, *VEGA*, *epiVEGA*) pozwoliły na wyznaczenie od 23 do 25 unikalnych zawieszeń. *Fuzzing* bazujący na zbiorze pustym nie wywołał żadnych niezamierzonych reakcji testowanego programu.



Rys. 51 Liczba znalezionych nowych ścieżek w bibliotece Pdfio v1.0.0 w ciągu 100 godzin fuzzingu.

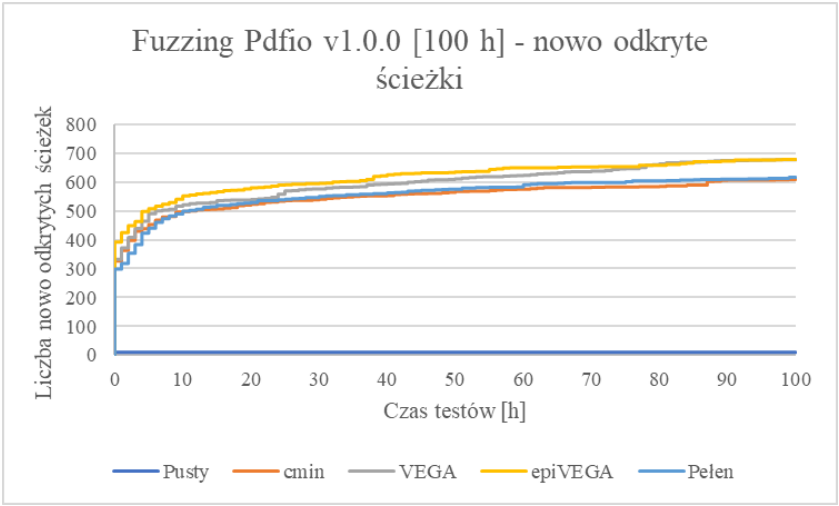
Kolejne 76 godzin testów doprowadziły do znalezienia 37 przypadków testowych powodujących zawieszenie biblioteki *pdfio* z wykorzystaniem korpusu wyznaczonego algorytmem *VEGA* i 35 w przypadku algorytmu *epiVEGA*. 32 unikalnych zawieszeń uzyskano z wykorzystaniem zbioru pełnego, a 29 podczas fuzzingu bazującego na korpusie

wyznaczonym algorytmem *cmin*. Zbiór pusty, pomimo zwiększenia czasu trwania testów, nie pozwolił na wykrycie żadnych niepożądanych anomalii w pracy testowanego oprogramowania.



Rys. 52 Liczba znalezionych nowych ścieżek w bibliotece Pdfio v.1.0.0 w ciągu pierwszych 24 godzin fuzzingu.

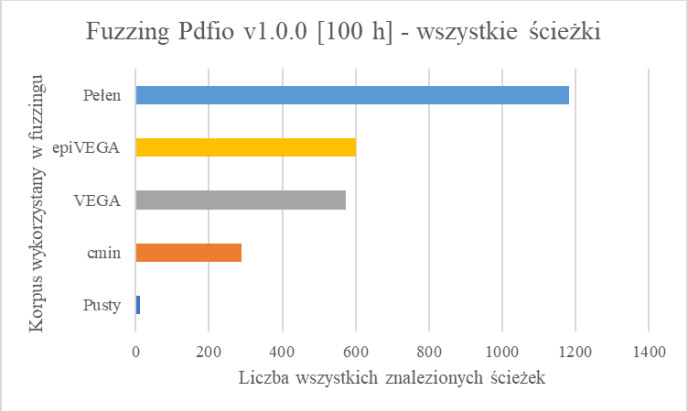
Pierwsze 24 godziny testu poskutkowały znalezieniem ok. 600 ścieżek przez korpus przedestylowany algorytmem *epiVEGA* oraz ok. 550 ścieżek z pomocą pozostałych zbiorów niepustych. *Fuzz*er wykorzystując pusty zbiór był w stanie wygenerować przypadki testowe, które pozwoliły na pokrycie tylko 10 ścieżek. Tak jak w przypadku biblioteki *giflib*, fakt ten jest najprawdopodobniej związany z procesem walidacji plików wejściowych, której nie mogły ukończyć wygenerowane przez fuzzer przypadki testowe.



Rys. 53 Liczba znalezionych nowych ścieżek w bibliotece Pdfio v.1.0.0 w ciągu 100 godzin fuzzingu.

Wydłużenie czasu pracy fuzzera doprowadziło niemal do zrównania się wyników uzyskanych z wykorzystaniem korpusów redukowanych algorytmami *VEGA* i *epiVEGA* oraz wyników uzyskanych z wykorzystaniem zbioru pełnego i korpusu zredukowanego

algorytmem *cmin*. *Fuzzing* bazujący na pustym korpusie nie pozwolił na zwiększenie liczby pokrytych ścieżek w grafie przepływu sterowania.

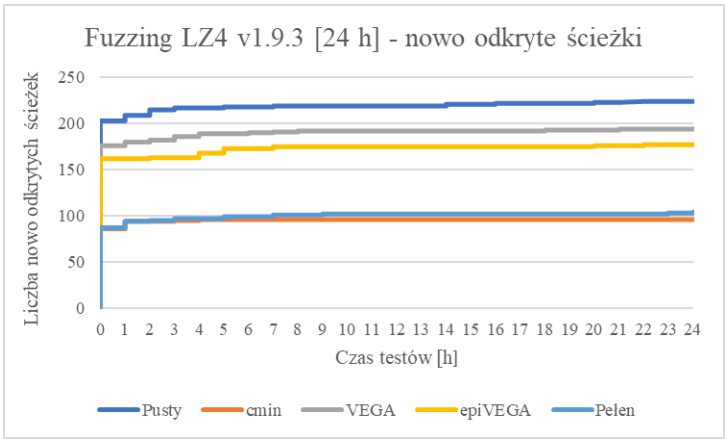


Rys. 54 Liczba wszystkich znalezionych ścieżek w bibliotece Pdfio v.1.0.0 w ciągu 100 godzin fuzzingu.

Korpus pełen aktywował w bibliotece *pdfio* 1809 ścieżek. Algorytmy genetyczne *VEGA* i *epiVEGA* odpowiednio 1252 i 1280. Zbiór wyznaczony algorytmem *cmin* wyznaczył w grafie przepływu sterowania 899 ścieżek. Zbiór pusty pozwolił na aktywację wyłącznie 23 ścieżek.

6.7.4 Fuzzing biblioteki przetwarzającej pliki skompresowane

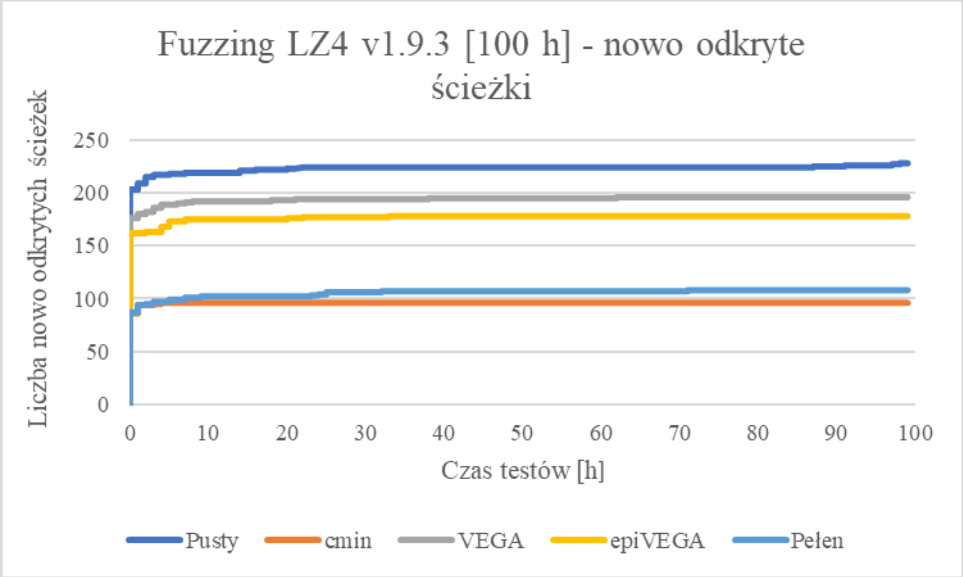
Fuzzing biblioteki LZ4 v1.9.3 z wykorzystaniem wszystkich wybranych korpusów nie pozwolił na znalezienie żadnych błędów ani unikalnych zawieszeń programu.



Rys. 55 Liczba znalezionych nowych ścieżek w bibliotece LZ4 v.1.9.3 w ciągu pierwszych 24 godzin fuzzingu.

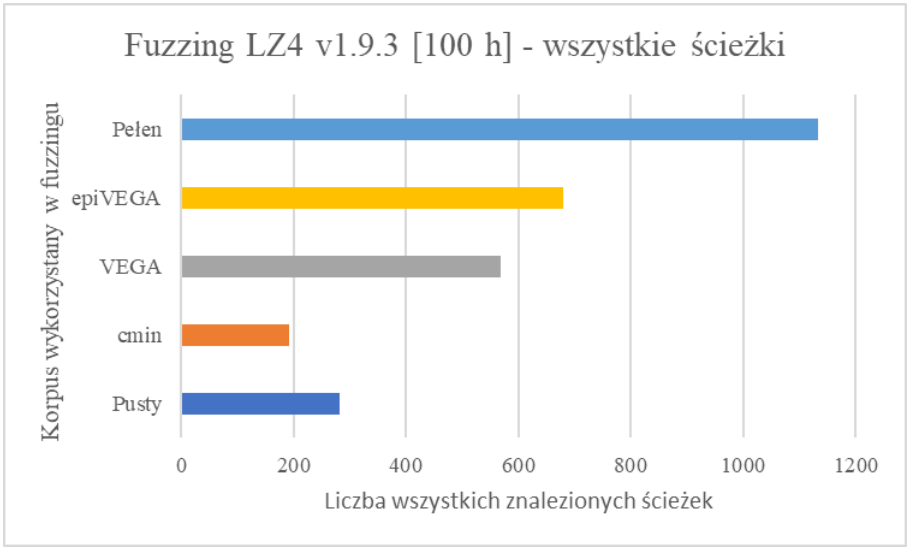
W pierwszej dobie testów najwięcej nowych ścieżek zostało wygenerowanych za pomocą zbioru pustego – ok 250. Na kolejnych miejscach uplasowały się korpusy wygenerowane algorytmami genetycznymi *VEGA* oraz *epiVEGA*, które aktywowały odpowiednio 190 i 160

ścieżek. Zbliżone wyniki osiągnął *fuzzing* z wykorzystaniem zbioru redukowanego algorytmem *cmin* i korpusu pełnego (ok. 100 ścieżek).



Rys. 56 Liczba znalezionych nowych ścieżek w bibliotece LZ4 v.1.9.3 w ciągu 100 godzin fuzzingu.

Kolejne 76 godzin testów nie wprowadziły znaczących zmian w wynikach wyznaczonych w pierwszej dobie testów. Przyrost wahał się między 1–5 nowych, odkrytych ścieżek.



Rys. 57 Liczba wszystkich znalezionych ścieżek w bibliotece LZ4 v.1.9.3 w ciągu 100 godzin fuzzingu.

Sumując wszystkie odkryte ścieżki wraz z tymi, które były pierwotnie wzbudzane przez poszczególne korpusy, ponownie największą liczbą aktywowanych ścieżek wykazał się zbiór niezredukowany. Następny w kolejności wynik uzyskano z wykorzystaniem korpusu wyznaczonego algorytmem *epiVEGA*. Algorytm *cmin* wyznaczył korpus, który pomimo tego,

że cechował się bardzo wysoką redukcją rozmiaru i czasu obsługi, w przypadku biblioteki LZ4 v1.9.3 aktywował najmniej ścieżek – tylko 199.

7. Podsumowanie i wnioski

Etap eksperymentalny potwierdził dotychczas wykazaną skuteczność fuzzingu w znajdywaniu błędów w oprogramowaniu (przytoczone wcześniej w pracy badania przeprowadzane na uniwersytecie *Wisconsin–Madison* pod przewodnictwem P. B. Millera). Jedna na cztery testowane biblioteki posiadała błędy, które można było znaleźć z zastosowaniem automatycznych, pseudolosowych testów. *Fuzzing* oparty o zestaw wyznaczony algorytm *epiVEGA* wykazał się skutecznością zbliżoną do testów wykorzystujących pełen zbiór testowy, przy czym znaczną liczbę błędów wykazano w pierwszych 24 godzinach testów. Pozwala to stwierdzić, że destylacja z jego pomocą spełniła swoje zadanie. Redukcja zbioru przypadków testowych z wykorzystaniem zmodyfikowanego autorsko algorytmu genetycznego pozwoliła przeprowadzić *fuzzing*, który cechował się wyższą efektywnością w krótszym czasie w porównaniu do korpusów, które uzyskano z zastosowaniem innych metod destylacji.

Etap eksperymentów, którego głównym celem była destylacja pierwotnego korpusu, rozpatrywana jako problem wielokryterialnego pokrycia zbioru, potwierdził skuteczność algorytmu *epiVEGA* w wyznaczaniu zredukowanych zbiorów przypadków testowych. Dla każdego rozpatrywanego formatu plików, zbiór wyznaczony z wykorzystaniem zaproponowanego zmodyfikowanego algorytmu genetycznego, cechował się pokryciem krawędzi w grafie przepływu sterowania równym korpusowi pierwotnemu, przy redukcji cech korpusu rozumianych jako wagi problemu *MCSCP*. Wyższą skutecznością cechowały się konfiguracje algorytmu *epiVEGA*, w których prawdopodobieństwo wystąpienia zjawiska epigenetycznego wynosiło $p_e \geq 0{,}2$. Również większy rozmiar populacji ($G = 500$) w większości przypadków (poza jednym) zapewniał rozwiązania, które przy redukcji poszczególnych wag cechowały się pierwotnym pokryciem kodu badanego oprogramowania. Spośród wariantów zaproponowanego operatora epigenetycznego – histonu, najlepsze wyniki osiągnięto podczas destylacji z wykorzystaniem operatora mieszanego (ponownie poza jednym), który zarówno aktywuje pożądane geny, jak i dezaktywuje geny potencjalnie szkodliwe.

Zaproponowane kodowanie, w którym obecność plików w końcowym korpusie utożsamiano z genem o wartości „*1*” a ich brak symbolizowała wartość „*0*”, okazał się skuteczny zarówno w pierwotnym algorytmie *VEGA*, jak i w jego zmodyfikowanej wersji.

Sterowanie zbieżnością algorytmu *epiVEGA*, bazujące na medianie oraz rozstępie, nie pozwoliło na redukcję pokrycia wraz z kolejnymi pokoleniami algorytmu genetycznego, przy czym wyraźnie wpłynęło na osłabienie trendu opadającego wartości przyjętych wag (całkowitego rozmiaru korpusu, średniej entropii oraz czasu obsługi).

W oparciu o zawarte w pracy rozważania teoretyczne oraz wyniki uzyskane w etapie eksperymentalnym, można potwierdzić postawioną tezę, że:

wykorzystanie w procesie destylacji wielokryterialnego algorytmu genetycznego wzbogaconego o operator epigenetyczny oraz mechanizm sterowania zbieżnością pozwala na efektywną redukcję korpusu danych testowych wykorzystywanych w procesie fuzzingu.

Do głównych elementów oryginalnych niniejszej rozprawy należy zaliczyć:

- rozszerzenie destylacji korpusu danych testowych wykorzystywanych w procesie fuzzingu z problemu minimalnego ważonego pokrycia zbioru do problemu wielokryterialnego pokrycia zbioru;
- wykorzystanie entropii uproszczonej jako nowego dodatkowego kryterium destylacji korpusu danych testowych;
- zaproponowanie operatora epigenetycznego, który uwzględnia eliminację osobnika w procesie selekcji jako czynnik stresowy niezbędny do jego wystąpienia (poprawne odwzorowanie zjawiska z ewolucji biologicznej);
- zaproponowanie kodowania, które pozwala na implementację operatora epigenetycznego tak, aby był zgodny z rozwiązywanym problemem i twierdzeniem o schematach;
- rozszerzenie algorytmu *VEGA* o operator epigenetyczny;
- rozszerzenie algorytmu *VEGA* o sterowanie zbieżnością;
- eksperymentalne ustalenie minimalnego skutecznego prawdopodobieństwa p_e wystąpienia operatora epigenetycznego oraz jego wariantu dla poszczególnych formatów plików podczas destylacji korpusu.

Bazując na dokonanym przeglądzie literatury oraz zrealizowanych w ramach rozprawy eksperymentach, możliwe jest wyznaczenie kolejnych etapów badań nad algorytmami ewolucyjnymi oraz ich zastosowaniem w procesie fuzzingu.

Niezwykle interesującym wydaje się pytanie, czy odpowiednie sortowanie chromosomu lub permutowanie genów w przypadku zastosowania operatorów epigenetycznych pozwoliłoby na stworzenie schematów, które pozytywnie wpłynęłyby na rozwiązywanie zadań z wykorzystaniem algorytmów ewolucyjnych.

Wartym zgłębienia zagadnieniem wydaje się również wykorzystanie algorytmu ewolucyjnego w roli samego *fuzzera*. Implementacja heurystyki tylko zbliżonej do zasad funkcjonowania algorytmów genetycznych poskutkowała powstaniem najbardziej znanego ze wszystkich fuzzerów – *AFL*. Znaczne ilości modyfikacji dokonywanych na przypadkach testowych podczas fuzzingu od razu nasuwają na myśl genetyczny operator mutacji. Wzbogacenie testów o element krzyżowania zmutowanych przypadków testowych oraz ich selekcji, mógłby poskutkować uzyskiwaniem jeszcze lepiej dopasowanych korpusów, a co za tym idzie – skutkować większą liczbą znalezionych w badanym oprogramowaniu potencjalnych podatności.

Bibliografia

[1] A. Takanen, „Fuzzing : the Past, the Present and the Future” w *Symposium sur la sécurité des technologies de l'information et des communications*, Rennes, 2009.

[2] J. W. Duran i S. Ntafos, „A report on random testing” w *ICSE '81: Proceedings of the 5th international conference on Software engineering*, San Diego, 1981.

[3] T. Klooster, F. Turkmen, G. Broenink, R. ten Hove i M. Böhme, „Effectiveness and Scalability of Fuzzing Techniques in CI/CD Pipelines” *arXiv:2205.14964 [cs.SE]*, 30 Maj 2022.

[4] K. Zmitrowicz, Jakość projektów informatycznych. Rozwój i testowanie oprogramowania, Gliwice: Helion, 2015.

[5] „AFLplusplus” [Online]. Available: https://aflplus.plus/.

[6] M. Kocielski, „LogicalTrust/minevra_lib” 2019. [Online]. Available: https://github.com/LogicalTrust/minerva_lib.

[7] „honggfuzz” [Online]. Available: https://honggfuzz.dev/.

[8] „libFuzzer – a library for coverage-guided fuzz testing” [Online]. Available: https://llvm.org/docs/LibFuzzer.html.

[9] Mozilla Foundation, „Firefox Source Docs” [Online]. Available: https://firefox-source-docs.mozilla.org/tools/fuzzing/fuzzing_interface.html.

[10] G. Vranken, „Differential fuzzing of cryptographic libraries” 14 Maj 2019. [Online]. Available: https://guidovranken.com/2019/05/14/differential-fuzzing-of-cryptographic-libraries/.

[11] Y. Zhang, W. Huo, K. Jian, J. Shi, L. Longquan, Z. Yanyan, Z. Cgai i L. Baoxu, „ESRFuzzer: an enhanced fuzzing framework for physical SOHO router devices to discover multi-Type vulnerabilities” *Cybersecur*, nr 24, 2021.

[12] Z4ziggy, „ZIGFRID – A PASSIVE RFID FUZZER.” 21 lipiec 2017. [Online]. Available: https://z4ziggy.wordpress.com/2017/07/21/zigfrid-a-passive-rfid-fuzzer/.

[13] M. Pachnik, „Methods of generating test data for carrying out the fuzzing process” *COMPUTER SCIENCE AND MATHEMATICAL MODELLING*, pp. 27-32, 2019.

[14] M. Böhme, V. Pham i A. Roychoudhury, „Coverage-based Greybox Fuzzing as Markov Chain” w *CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Wiedeń, 2016.

[15] L. Hayes, H. Gunadi, A. Herrera, J. Milford, S. Margath, M. Sebastian, M. Norrish i A.

L. Hosking, „MoonLight: Effective Fuzzing with Near-Optimal Corpus Distillation” *arXiv:1905.13055 [cs.CR],* 2019.

[16] Y. Wang, P. Jia, L. Liu, C. Huang i Z. Liu, „A systematic review of fuzzing based on machine learning techniques” *PLOS ONE,* pp. 1-37, 18 Sierpień 2020.

[17] M. Zalewski , „Technical “whitepaper” for afl-fuzz” [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt.

[18] M. Pachnik, R. Kasprzyk i K. Worwa, „Simplified Entropy as A Criterion for Corpus Distillation in Fuzzing Process” w *Conference: Proceedings of the 37th International Business Information Management,* Cordoba, 2021.

[19] L. Hayes, H. Gunadi, A. Herrera, J. Milford, S. Margath, M. Sebastian, M. Norrish i A. L. Hosking, „Corpus Distillation for Effective Fuzzing: A Comparative Evaluation” *arXiv:1905.13055v2 [cs.CR],* 2020.

[20] „AFLplusplus/afl-cmin” [Online]. Available: https://github.com/AFLplusplus/AFLplusplus/blob/stable/afl-cmin.

[21] C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou i J. Chen, „SmartSeed: Smart Seed Generation for Efficient Fuzzing” *arXiv:1807.02606,* Lipiec 2018.

[22] T. H. Cormen, C. E. Leiserson i R. L. Rivest, w *Wprowadzenie do algorytmów*, Warszawa, Wydawnictwa Naukowo-TEchniczne, 1997, pp. 346-374.

[23] K. Chromiński, Zastosowanie procesów epigenetycznych w algorytmach genetycznych, Katowice: Uniwersytet Śląski, 2019.

[24] A. Enrique i D. H. Stolfi, „EpiGenetic Algorithms: A New Way of Building GAs Based on Epigenetics” *Information Sciences,* Październik 2017.

[25] J. D. Schaffer, „Multiple Objective Optimization with Vector Evaluated Genetic Algorithms” w *Proceedings of the 1st International Conference on Genetic Algorithms,* Pittsburgh, 1985.

[26] A. Takanen, J. D. Demott i C. Miller, Fuzzing for Software Security Testing and Quality Assurance, Norwood: Artech House, 2008.

[27] B. So, L. Fredriksen i B. P. Miller , „An Empirical Study of the Reliability of UNIX Utilities” *Communications of the ACM Vol. 33,* pp. 32-44, 1990.

[28] S. Schumilo , C. Aschermann , R. Galwik, S. Schinzel i H. Thorsten , „kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels” w *26th USENIX Security Symposium,* Vancouver, 2017.

[29] J. Nelißen, Buffer Overflows for Dummies, Swansea : SANS Institute, 2002.

[30] A. Hertzfeld, „Monkey Lives” Październik 1983. [Online]. Available: https://www.folklore.org/StoryView.py?story=Monkey_Lives.txt.

[31] P. Agrawal i V. Agrawal, „Probabilistic Analysis of Random Test Generation Method for Irredundant Combinational Logic Networks.” *IEEE Transactions on Computers C-24,* pp. 691-695, 1975.

[32] B. P. Miller, L. Fredriksen i B. So, „An Empirical Study of the Reliability of UNIX Utilities” *Communications of the ACM,* Grudzień 1990.

[33] B. P. Miller, D. Koski, C. P. Lee, V. Magnaty, R. Murthy, A. Natarajan i J. Steidl , „Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services” *Computer Sciences Technical Report,* Kwiecień 1995.

[34] J. Forrester i B. P. Miller, „An Empirical Study of the Robustness of Windows NT Applications Using Random Testing” w *4th USENIX Windows Systems Symposium*, Seattle, 2000.

[35] B. P. Miller, G. Cooksey i F. Moore, „An Empirical Study of the Robustness of MacOS Applications Using Random Testing” w *First International Workshop on Random Testing*, Nowy Jork, 2006.

[36] B. P. Miller, Z. Mengxiao i E. Heymann, „The Relevance of Classic Fuzz Testing:Have We Solved This One?” *IEEE Transactions on Software Engineering.,* Luty 2021.

[37] C. Double, „A Quick Look at the Rust Programming Language” 2011. [Online]. Available: https://bluishcoder.co.nz/2011/03/31/a-quick-look-at-the-rust-programming-language.html.

[38] „Chromium Blog” Kwiecień 2012. [Online]. Available: https://blog.chromium.org/2012/04/fuzzing-for-security.html.

[39] „CVE-2014-6271 Detail” National Vulnerability Database, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2014-6271.

[40] iblanda, „Stagefright fuzzing” [Online]. Available: https://github.com/fuzzing/MFFA.

[41] „Python” [Online]. Available: https://www.python.org/.

[42] MITRE, 30 Wrzesień 2015. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1538.

[43] „OSS-fuzz” [Online]. Available: https://github.com/google/oss-fuzz.

[44] „Go” [Online]. Available: https://golang.org.

[45] Java. [Online]. Available: https://www.java.com/pl/about/whatis_java.jsp.

[46] „Onefuzz” [Online]. Available: https://github.com/microsoft/onefuzz.

[47] M. Jurczyk, „Issue 2002: Samsung Android multiple interactionless RCEs and other remote access issues in Qmage image codec built into Skia” Google, 28 Styczeń 2020. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=2002.

[48] M. E. Khan, „Different Forms of Software Testing Techniques for Finding Errors” *International Journal of Computer Science Issues,* tom 7, nr 3, pp. 11-16, 2010.

[49] F. Khan i M. Ehmer, „A Comparative Study of White Box, Black Box and Grey Box Testing Techniques” *International Journal of Advanced Computer Science and Applications,* tom 3, nr 6, pp. 12-15, 2012.

[50] S. Nidhra, „Black Box and White Box Testing Techniques - A Literature Review” *International Journal of Embedded Systems and Applications* , tom 2, nr 2, pp. 29-50, 2002.

[51] A. Roman, Testowanie i jakość oprogramowania. Modele techniki, narzędzia, Warszawa: Wydawnictwo Naukowe PWN SA, 2015.

[52] J. P. Aumasson i Y. Romailler, „Automated Testing of Crypto Software Using Differential Fuzzing” w *BlackHat,* 2017.

[53] S. Nilizadeh, . Y. Noller i C. Pasareanu, „DifFuzz: Differential Fuzzing for Side-Channel Analysis” w *IEEE/ACM 41st International Conference on Software Engineering (ICSE),* 2019.

[54] „ptrace(2) — Linux manual page” 21 3 2021. [Online]. Available: https://man7.org/linux/man-pages/man2/ptrace.2.html.

[55] „OpenSSL Security Advisory” 26 8 2016. [Online]. Available: https://www.openssl.org/news/secadv/20160926.txt.

[56] K. Frankowicz, „LLVM LibFuzzer” *Programista,* pp. 64-66, 10 2017.

[57] „Clang 16.0.0git documentation” [Online]. Available: https://clang.llvm.org/docs/SanitizerCoverage.html.

[58] „Google/Fuzzer test suite” [Online]. Available: https://github.com/google/fuzzer-test-suite.

[59] M. Zalewski, „afl-generated, minimized image test sets (partial)” [Online]. Available: https://lcamtuf.coredump.cx/afl/demo/.

[60] „afl-tmin - Man Page” [Online]. Available: https://www.mankier.com/8/afl-tmin.

[61] M. Składnikiewicz i M. Jurczyk, „Fuzzing” *Programista,* pp. 58-66, Lipiec 2016.

[62] M. Arjovsky, S. Chintala i L. Bottou, „Wasserstein GAN” *arXiv:1701.07875,* Styczeń

2017.

[63] J. Jinho , H. Hu, D. Solodukhin, D. Pagan, K. H. Lee i T. Kim, „Fuzzification: Anti-Fuzzing Techniques” w *USENIX Security*, Thuwal, 2019.

[64] „Obfuscated Files or Information: Software Packing” MITRE, [Online]. Available: https://attack.mitre.org/techniques/T1027/002/.

[65] C. Collberg, C. Thomborson i D. Low, „A Taxonomy of Obfuscating Transformations” Department of Computer Science, University of Auckland, New Zeland, 1997.

[66] T. Brunson, „Trail of Bits Blog” 2018. [Online]. Available: http://blog.trailofbits.com/2018/10/05/how-to-spot-good-fuzzing-research.

[67] J. H. Bremermann, „Optimization through evolution and recombination.” w *Yovits C. M., Jacobi T. G., Goldstein D. G. (ed.): Self – organizing systems*, Spartan Books, 1962, pp. 93-106.

[68] A. S. Fraser, „Simulation of genetic systems by automatic digital computers” *Australian Journal of Biological Science*, pp. 484-491, 1957.

[69] J. G. Friedman, „Digital simulation of an evolutionary process” *General Systems Yearbook*, p. 171 – 184, 1959.

[70] L. J. Fogel, J. A. Owens i M. J. Walash, Artificial intelligence through simulated evolution, John Wiley & Sons Publishing, 1966.

[71] R. Ingo, „Evolution Strategy: Nature’s Way of Optimization” *Optimization: Methods and Applications, Possibilities and Limitations. Lecture Notes in Engineering, Springer, Berlin, Heidelberg*, tom 47, pp. 106-126, 1989.

[72] J. H. Holland, Adaptation in natural and artificial systems, Cambridge : The MIT Press. 2nd edition, 1992.

[73] K. S. W. De Jong, „A formal analysis of the role of multi-point crossover in genetic algorithms” *Ann Math Artif Intell 5*, pp. 1-26, 1992.

[74] W. Spears i K. De Jong, „On the Virtues of Parametrized Uniform Crossover” w *4th International Conference on Genetic Algorithms*, San Diego, 1991.

[75] D. J. Schaffer i L. J. Eshelman, „“On Crossover as an Evolutionarily Viable Strategy.” *ICGA*, 1991.

[76] D. B. Fogel i J. W. Atmar, „Comparing genetic operators with gaussian mutations in simulated evolutionary processes using linear systems” *Biological Cybernetics*, pp. 111-114, 1990.

[77] T. . P. Dinh, B. H. Thanh, T. . T. Ba i . L. N. Binh , „Multifactorial evolutionary

algorithm for solving clustered tree problems: competition among Cayley codes" *Memetic Computing 12,* pp. 185-217, 2020.

[78] H. Dang Quoc, L. Nguyen The, C. Nguyen Doan i N. Xiong, „Effective Evolutionary Algorithm for Solving the Real-Resource-Constrained Scheduling Problem." *Journal of Advanced Transportation ,* pp. 1-11.

[79] M. Chudy, „Wybrane algorytmy optymalizacji" Warszawa, EXIT, 2014.

[80] B. Batut, D. P. Parsons i S. Fischer, „In silico experimental evolution: a tool to test evolutionary scenarios" *BMC Bioinformatics,* 2013.

[81] J. H. Holland, „Outline for a logical theory of adaptive systems" *Journal of the Association for Computing Machinery,* p. 297 – 314, 1962..

[82] K. A. De Jong, *PhD. Dissertation - An analysis of the behavior of a class of genetic adaptive systems.,* Ann Arbor: University of Michigan, 1975.

[83] D. E. Goldberg, Genetic Algorithms in Search Optimization & Machnie Learning, Alabama: Addison-Wesley Publishing Company, Inc., 1985.

[84] L. J. Fogel, J. A. Owens i M. J. Walsh, Artificial intelligence through simulated evolution., Hoboken: John Wiley & Sons Publishing,, 1996.

[85] J. Heitkotter i D. Beasley, „The Hich-Hiker's Guide to Evolutionary Computation" 1 Kwiecień 1991. [Online]. Available: http://aiinfinance.com/gafaq.pdf.

[86] D. B. Fogel, „An analysis of evolutionary programming" w *Evolutionary Programming Society*, 1992.

[87] A. Király i J. Abonyi , „A Novel Approach to Solve Multiple Traveling Salesmen Problem by Genetic Algorithm" *Computational Intelligence in Engineering ,* p. 141–151, Październik 2010.

[88] D. Rutkowska, M. Piliński i L. Rutkowski, Sieci neuronowe, algorytmy genetyczne i systemy rozmyte, Warszawa: PWN, 1991.

[89] R. Patel i M. M. Raghuwanshi, „Review on Real Coded Genetic Algorithms Used in Multiobjective Optimization" w *3rd International Conference on Emerging Trends in Engineering and Technology*, Goa, 2010.

[90] K. Murawski, „Obliczenia ewolucyjne - geneza i zastosowanie" *Biuletyn Instytutu Automatyki i Robotyki WAT,* nr 15, pp. 55-104, 2001.

[91] J. H. Holland, „Adaptation in natural and artificial systems" *University of Michigan Press,* 1975.

[92] Z. Michalewicz, Algorytmy genetyczne + struktury danych = programy ewolucyjne,

Warszawa: Wydawnictwo Naukowo-Techniczne, 1999.

[93] A. Bethke, Genetic algorithms as function optimizers, University of Michigan, 1980.

[94] C. A. Villee, Biologia, Warszawa: Państwowe Wydawnicto Rolnicze, 1990.

[95] E. Lazaro, „Mutagen" w *Encyclopedia of Astrobiology 2nd ed.*, Berlin, Springer, 2015, p. 1649–1650.

[96] R. N. Greenwell, J. E. Angus i M. Finck, „Optimal Mutation Probability for Genetic Algorithms" *Mathl. Comput. Modelling,* tom 21, nr 8, pp. 1-11, 1995.

[97] N. Soni i T. Kumar, „Study of Various Mutation Operators in Genetic Algorithms" *International Journal of Computer Science and Information Technologies,* tom 5, nr 3, pp. 4519-4521 , 2014.

[98] S. M. Kantamneni, PhD. Genetic Algorithm as a Computational Approach for Phase Improvement and Solving Protein Crystal Structures, Hamburg: University of Hamburg, 2020.

[99] S. N. Sivanandam i S. N. Deepa, Introduction to Genetic Algorithms, Berlin: Springer, 2008.

[100] S. Alves, S. Oliviera i A. R. Rocha Neto, „A novel educational timetabling solution through recursive genetic algorithms" w *Latin America Congress on Computational Intelligence (LA-CCI),* Curitiba, 2015 .

[101] S. Wang, D. Zickler, N. Kleckner i L. Zhang, „Meiotic crossover patterns: Obligatory crossover, interference and homeostasis in a single process" *Cell Cycle,* tom 14, nr 3, pp. 305-314, 2015.

[102] M. M. Navarro i B. B. Navarro, „Evaluations of Crossover and Mutation Probability of Genetic Algorithm in an Optimal Facility Layout Problem" w *Proceedings of the 2016 International Conference on Industrial Engineering and Operations Management,* Kuala Lumpur, 2016.

[103] V. P. Patil i D. D. Pawar, „The optimal crossover or mutation rates in genetic algorithm: a review" *International Journal of Applied Engineering and Technology,* tom 5, nr 3, pp. 38-41, 2015.

[104] A. J. Umbakar i P. D. Sheth, „Crossover Operators in Genetic Algoritms: A Review" *Ictac Jornal on Soft Computing,* tom 6, nr 1, pp. 1083-1092, 2015.

[105] K. A. De Jong i W. M. Spears, „A formal analysis of the role of multi – point crossover in genetic algorithms" *Annals of Mathematics and Artificial Intelligence,* nr 5, pp. 1-12, 1992.

[106] S. Narmadha, V. Selladurai i G. Sathish, „Multi-Product Inventory Optimization using Uniform Crossover Genetic Algorithm” *International Journal of Computer Science and Information Security,* tom 7, nr 1, pp. 170-179, 2010.

[107] A. E. Eiben, C. H. van Kemenade i J. N. Kok, „Orgy in the Computer: Multi-Parent Reproduction in Genetic Algorithms” w *Proceedings of Advances in Artificial Life, Third European Conference on Artificial Life*, Granada, 1995.

[108] M. J. Varnamkhasti, L. S. Lee, M. R. Abu Bakar i W. J. Leong, „A Genetic Algorithm with Fuzzy Crossover Operator and Probability” *Advances in Operations Research,* Luty 2012.

[109] B. Hasan, M. A. Khamees i A. Mahmoud, „A Heuristic Genetic Algorithm for the Single Source Shortest Path Problem” w *2007 IEEE/ACS International Conference on Computer Systems and Applications,* Amman, 2007.

[110] H. Pohlheim, *GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with Matlab,* 2006.

[111] D. E. Goldberg i K. Deb, „A Comparative Analysis of Selection Schemes Used in Genetic Algorithms” *Foundations of Genetic Algorithms,* tom 1, pp. 66-93, 1991.

[112] A. Lipowski i D. Lipowska, „Roulette-wheel selection via stochastic acceptance” *Physica A: Statistical Mechanics and its Applications,* tom 391, nr 6, pp. 2193-2196, 2011.

[113] L. Costa i P. Oliviera, „An Evolution Strategy for Multiobjective Optimization” w *Proceedings of the 2002 Congress on Evolutionary Computation.*, Honolulu, 2005.

[114] S. H. Nguyen, „Algorytmy ewolucyjne - teoria, techniki dodatkowe.” Polsko-Japońska Akademia Technik Komputerowych, 2009. [Online]. Available: https://edu.pjwstk.edu.pl/wyklady/nai/scb/wyklad11/w11.htm.

[115] F. Sadjadi, „Comparison of fitness scaling functions in genetic algorithms with applications to optical processing” w *Proceedings of SPIE - The International Society for Optical Engineering*, Orlando, 2004.

[116] W. C. Jackson i J. D. Norgrad, „A Hybrid Genetic Algorithm with Boltzmann Convergence Properties” *Journal of Optimization Theory and Applications,* pp. 431-443, 2008.

[117] D. Bhandari, C. D. Murthy i S. K. Pal, „Variance as a stopping criterion for genetic algorithms with elitist model.” *Fundamenta Informaticae*, tom 120, nr 2, pp. 145-164, 2012.

[118] A. E. Eiben i S. K. Smit, „Parameter tuning for configuring and analyzing evolutionary” *Swarm and Evolutionary Computation,* tom 1, nr 1, pp. 19-31, 2011.

[119] A. E. Eiben i S. K. Smit, „Evolutionary algorithm parameters and methods to tune” w *Autonomous Search,* Springer, 2012, pp. 25-38.

[120] M. Mosayebi i M. S. Sodhi, „Tuning genetic algorithm parameters using design of experiments” w *GECCO '20: Genetic and Evolutionary Computation Conference*, Cancun, 2020.

[121] M. F. Abdelatti i M. S. Sodhi, „Using Reinforcement Learning for Tuning Genetic Algorithms” w *GECCO 2021 - The Genetic and Evolutionary Computation Conference,* Lille, 2021.

[122] E. S. Nicoara, „Mechanisms to Avoid the Premature Convergence of Genetic Algorithms” *UniversităŃii Petrol – Gaze din Ploieşti,* tom LXI, nr 1, pp. 87-96, 2009.

[123] E. K. Gbashi, *Proposed Secret Encoding Method Based Genetic Algorithm For Elliptic Curve Cryptography Method,* Irackie Stowarzyszenie Technologii Informacyjnych, 2018.

[124] Y. Jiang, L. Tang, L. H i A. Zeng, „A variable-length encoding genetic algorithm for incremental service composition in uncertain environments for cloud manufacturing” *Applied Soft Computing,* tom 123, nr 1, p. 108902 , Kwiecień 2022.

[125] N. Kubota, K. Shimojima i T. Fukda, „The Role of virus infection in a virus-evolutionary genetic algorithm” *App. Math. and Com. Sci.,,* tom 6, nr 4, pp. 415-429, 1996.

[126] H. Xingshi i L. Han, „A novel binary differential evolution algorithm based on artifical immune system” w *Evolutionary Computation 2007*, Singapur, 2007.

[127] C. H. Waddington, „The epigenotype” *Endeavor,* tom I, pp. 18-20, 1942.

[128] A. Pal, „Epigenetics and DNA Methylation” w *Protocols in Advanced Genomics and Allied Techniques,* Springer, 2021, pp. 245-278.

[129] T. Chen, B. Long, G. Ren, T. Xiang, L. Li, Z. Wang, Q. Z, Y. He, Q. Zeng, S. Hong i G. Hu, „Protocadherin20 Acts as a Tumor Suppressor Gene: Epigenetic Inactivation in Nasopharyngeal Carcinoma” *Journal of Cellular Biochemistry,* tom 115, nr 8, pp. 1766-1775, 2015.

[130] A. Prunell, „Biophysical Journal” *Biophysical Journal,* tom 72, nr 3, pp. 983-984, 1997.

[131] M. B. Eslaminejad, N. Fani i M. Shahhoseini, „Epigenetic Regulation of Osteogenic and Chondrogenic” *Cell Journal*, Maj 2013.

[132] A. Klosin, E. Casas, C. Hidalgo-Carcedo, T. Vavouri i B. Lehner, „Transgenerational transmission of environmental information in C. elegans” *Science,* pp. 320-323, 2017.

[133] R. Paro, U. Grossniklaus, R. Santoro i A. Wutz, „Genomic Imprinting” w *Introduction to Epigenetics*, tom 12, Springer Nature Switzerland AG, 2021, pp. 91-115.

[134] P. Soloway, „Genetics: Paramutable possibilities” *Nature,* tom 441, nr 7092, pp. 413-414, 2006.

[135] M. Capovilla, A. Robichon i M. Rassouzadegan, „A new paramutation-like example at the Delta gene of Drosophila” *PLoS ONE,* tom 12, nr 3, pp. 1-16, 2017.

[136] A. J. Wijnen, G. Stein i J. Lian, „Bookmarking the Genome: Maintenance of Epigenetic Information” *Journal of Biological Chemistry,* tom 286, nr 21, pp. 18355-18361, 2011.

[137] P. P. Liberski, Choroba Creutzfeldta-Jakoba i inne choroby wywołane przez priony – pasażowalne encefalopatie gąbczaste człowieka, Lublin: Czelej, 2003.

[138] M. Gryzińska, K. Andraszek, A. Strachecka i G. Jocek, „Metylacja DNA w regulacji” *Przegląd hodowlany,* nr 2, pp. 2-5, 2012.

[139] B. Wolf, „Allelic Exclusion or Inclusion” w *Paradoxes in Immunology*, Boca Raton, CRC Press, 2019, pp. 307-324.

[140] J. Manjrekar, „Epigenetic inheritance, prions and evolution” *Journal of Genetics,* tom 96, nr 3, 2017.

[141] B. M. Javierre, H. Hernando i E. Ballesta, „Environmental triggers and epigenetic deregulation in autoimmune disease” *Discovery Medicine,* tom 12, nr 67, pp. 535-545, 2011.

[142] A. Capara, P. Toth i M. Fischetti, „Algorithms for the Set Covering Problem” *Annals of Operations Research,* tom 98, nr (1-4), pp. 353-371, 2000.

[143] W. Zang, X. Liu i X. Li, „A DNA Solution on Surface for Minimal Set Covering Problem” w *Pervasive Computing and the Networked World*, Phnom Penh, 2014.

[144] Y. Liu, „A heuristic algorithm for the multi-criteria set-covering problems” *A heuristic algorithm for the multi-criteria set-covering problems,* tom 6, nr 5, pp. 21-23, 1993.

[145] „Efficient Fuzzing Guide” Google, [Online]. Available: https://chromium.googlesource.com/chromium/src/+/main/testing/libfuzzer/efficient_fuzzing.md#Seed-corpus.

[146] K. Frankowicz, „Kamil Frankowicz” 13 Kwiecień 2017. [Online]. Available: https://frankowicz.me/llvm-libfuzzer-fast-track-2/.

[147] M. Składnikiewicz , „Entropia – pomiar i zastosowanie” *hakin9,* pp. 58-61, 3/2008.

[148] J. Kääp, „Corpus distillation & fuzzing” [Online]. Available: https://nordictestingdays.eu/files/files/jaanus_kaap_fuzzing.pdf.

[149] J. D. Schaffer, „Multiple Objective Optimization with Vector Evaluated Genetic Algorithms” w *Proceedings of the 1st International Conference on Genetic Algorithms*, Pittsburgh, 1985.

[150] „Performance Tips” Google, [Online]. Available: https://afl-1.readthedocs.io/en/latest/tips.html#performance-tips.

[151] W. M. Spears, „Crossover or mutation?” w *Conference: Foundations of Genetic Algorithms*, Colorado, 1992.

Spis Tabel

Spis Rysunków