Resumo da dissertação Redes neurais de programas do mundo real e suas aplicação à evolução automatizada de software de Eric Schulte

Fábio Moreira Duarte

Resumo

Os softwares de desenvolvimento de ecosistemas são produtos de força evolutiva, e problemas do mundo real podem ser aprimoradas com uso de automatização. Esta dissertação apresenta evidências empiricas que software são inerentement robustos para pequenas transformações aleátorias no programa, ou mutações. Operações de mutação podem ser aplicadas no codigo fonte, código assembler compilado, ou binário executável. Gerando invariantes dos programas trabalhados diferindo significativamento do original, porem funcional. Aplicando sucessivas mutações a mesmo programa, cobre uma grande rede neural de variantes funcionais de projetos do mundo real.

Propriedades de robustez mutacional e redes neurais correspondentes é estudo em biologia e acredita estar relacionado a capacidade de evolução e adaptação não supervisionada. Como em sistemas biologicos, robustez mutacional e redes neurais em sistemas de software permitem evolução automatizada.

A dissertação apresenta aplicações que software alavancam redes neurais para automatizar desenvolvimento comum de software e tarefas de manutenção. Redes neurais são exploradas para gerar diversas implementações do software para aprimorar a segurança na execução e reparar erros latentes.

1 Introdução

Apresenta estudos impiricos dos efeitos de pequenas transformações aleátorias, ou mutações, em softwares do mundo real. Funcionalidade de software é robustez inerente para mutação. Aplicando sucessivas mutações, explora-se grandes redes neurais de variantes funcionais de projetos de software existentes. Utiliza-se robustez mutacional de software e redes neurais para aprimoração de software atraves de processos evolucionários de modificações estocasticas e avaliação fitness imitando seleção natural.

Propoem-se que softwares de desenvolvimento de ecosistemas é produto de força evolucionária. Apresenta evidências impiricas de robustez mutacional e redes neurais em software.

2 Plano de fundo e revisão literária

O estudo de robustez e evolutividade em conceitos biologicamente proveem o conceito e terminologias, onde investigou-se os mesmo principios em sistemas computacionais.

2.1 robustez e evolutividade em biologia

A habilidade de sistemas vivos de manter a funcionalidade atraves de diferentes ambientes e adaptar-se a novos ambientes é importante para sistemas feitos pelo homem.

Sistemas vivos consistem de genótipos de fenótipos. O genótipo são as informações hereditárias que especifica, ou da origem ao organismo. O organismo resultante, ou iterações com o mundo são fenótipos.

Genótipos e fenótios possuem tipos de robustez. Robustez do genótipo é a capacidade de produzir o mesmo fenótipo, embora aja diversificações no ambiente.

Robustez ambiental é a habilidade do fenótipo de manter a funcionalidade através de perturbações no ambiente.

RObustez mutacional aparenta ser uma característica evolutiva, tendendo a aumentar a robustez mutacional de componentes biologicos. Robustez mutacional é favorecida por seleção natural como proteção contra mutações.

O fitness em biologia evolocionária é a habilidade para sobreviver e reproduzir.

2.2 Computação evolutiva

Descreve-se os campos de evolução digital e algoritmos evolutivos. Em evolução digital em sistemas computacionais utiliza-se experimentos não factiveis ainda em sistemas biológicos. Algoritmos evolutivos em sistemas de engenharia são otimizadas utilizando algoritmos que imitam processos biológicos de seleção general.

2.3 Evolução digital

Quadros de tempo evolutivos e o grau de controle ambiental requer experimentos não alcançaveis em experimentos utilizando sistemas biologicos. Evolução digital, permite mais controle via modelos computacionais de população evolutiva. Tais modelos representam genótipos utilizando linguagens assembly especializadas em ambiente onse sua execução determina o sucesso da reprodutividade.

Modelos de evolução biologica, geroram hipóteses sobre propriedades de linguagens de programação que podem encoragar evolutividade.

2.4 Algoritmos evolutivos

Algoritmos evolutivos, incluem algoritmos genéticos e programação genética de sub campo.

Algoritmos genéticos aplicam visões Darwinistas sobre seleção natural para problemas de otimização em engenharia. Algoritmo genético requer uma função fitness onde o algoritmos busca maximização.

Programação genética é uma especialização de algoritmos genéticos com soluções candidatas a programas, geralmente representadas por uma arvores sintática abstrata.

Algoritmos genéticos e programação genética diferente quanto a forma que os candidatos a solução são representadas. O trabalho apresenta dissertações aplicadas a programas existentes como programação genética.

Algoritmos evolutivos são dependentes de propriedades fitness definida pela função fitness. Estudos NK foram aplicadas a efetividade das tecnicas sobre ajustamento.

2.5 Engenharia de software

Computação aproximada engloba tecnicas que buscam explorar troca de precisao reduzida em computação para aumento de eficiência. A motivação principal é que sistemas computacionais provee mais precisão e confiabilidade requirida no nivel do hardware atraves de resultados visiveis.

Beal e Sussman propoem um sistema para aumentar a robustez de software, pre processando as entradas do programa. Supondo que a maior parte dos softwares operam sobre subconjunto de possiveis entradas.

Sistemas anteriores aprendem e impoêm variántes extraidas de binários executáveis utilizando Daikon. Quando as variantes são violadas por exploração de vulnerabilidade no programa origina aplicando preservação de invariantes, garando execução continua.

Tais tecnicas encluem realização de laços e botões dinâmicos. Realização de laços compila o software para uma simples IR (interpretação representativa), construtores de laços são encontrados na IR e modificados para executar laços, ignorando algumas execuções do laço.

A equivocos quanto a fragilidade do software e que pequenas mudanças no codigo levariam a mudanças catastróficos. A fragilidade é codificada em sistemas de teste de mutação. Tais sistemas medem conjuntos de teste de programas pela percentual de mudanças aleátorias no programa que leva o conjunto a falhar, supondo que mudanças no resultado do programa trabalhado resulta em ruptura. Presume que programas mutantes são falhas e equivalentes ao programa original.

A detecção de programas equivalentes é problema aberto na comunidade de teste mutacional.

2.6 Genprog: Programa evolutivo de reparo

Genprog é uma ferramenta para reparação automatizada defeitos utilizando algoritmos evolutivos. Requer que o programa seja escrito em C e acompanhado por um conjunto de teste.

Como entrada Genprog requer que o codigo fonte em C do programa com erro, o conjunto de teste no qual a versão atual do software é habilitada a passar, e um teste de falha possa indicar o erro.

Genprog busca por variantes do programa original que é capaz de passar no caso de teste originalmente falhado, enquanto passa no conjunto de teste de regreção. A versão é retornada por porões de busca evolucionária de tecnicas Genprog. Como passo final pos processando a diferença entre o programa original e o reparo é minimizado para o menor conjunto requiridos para alcançar o reparo. É realizada utilizando debugador delta.

3 Representação de software, mutação e redes neurais

Em sistemas biotológicos, robustez mutacional e redes neurais alcançadas por mutação de preservação de fitness, necessária para aprimoramento evolucionário.

Robustez mutacional e redes neurais surgem em sistemas com uma estrutura genética que suporta mutação e cruzamento dando surgimento a um fenótipo que suporta a avaliação fitness.

3.1 Representação e transformação

3.1.1 Representação

Observou-se representações de programas com informações genéticas que podem ser modificadas com operações de mutação e cruzamento. Representação de programas geralmente são três vetores lineares hierárquicos.

Investigara-se duas arvores de representação de alto nivel e três vetores de representação de baixo nivel. Os três programas de representação são: baseada em CIL, frontend da família de linguagem C para LLVM (Clang) ASTs. As três programas de representação de baixo nivel incluem codigo ASM, maquina virtual de baixo nivel (LLVM) IR e uma representação aplicada diretamente ao arquivo binário ELF.

3.1.2 Clang-AST

Representações de alto nivel baseada em analise ASTs de familia de linguagem C, utilizando frontend da família de linguagem C para LLVM (Clang). O nivel de representação é proximo a codigo fonte escrito por desenvolvedores de software humano.

3.1.3 Cil-AST

O nivel mais alto de presentação é baseada em analise AST de codigo fonte em C utilizando linguagem intermediaria C (CIL).

CIL simplifica construtores de codigo fonte em C para facilita a manipulação.

3.1.4 LLVM

A maquina virtual de baixo nivel (LLVM) representa operações sobre compiladores LLVM, tarefa única estatística (SSA). LLVM suporta multiplas linguagens frontend tornando aplicável a vários projetos de software.

Ferramentas de conjuntos de risco emergem en torno de infraestruturas LLVM.

3.1.5 ASM

Linguagens de compilação são passiveis de modificação ao nivel assembly (ASM). Representa programas como vetores de instruções de assembly. Compiladores suportam tradução direta, a flag -s no compilador do compilador GCC

cria gera uma representação de instruções ASM. Representações de programas ASM, analisam a sequência de instruções em vetores de instruções assembly de argumentos. Equivalente a saída emitida pelo GCC -S em linhas de caracteres. Podendo ser manipulado e serializado devolta a text em string de instruções assembly.

3.1.6 ELF

O formato de arquivo executável e formato vincúlado (ELF) é formato de compilação e bibliotecas vinculadas e arquivos executaveis. O codigo em arquivos ELF é executado, carregado para a memoria e traduzido em instruções assembler de argumentos serializados na GPU. Utilizando ferramentas em combinações com disassimiladores, é possivel para modificar a sequência de instruções assemble em arquivos ELF.

3.1.7 Transformações

Cada programas de representação utilizado suporta conjuntos de três multiplas transformações e uma transformação de cruzamento.

Os quatro transformaçoes do programa são simples e gerais, não codificam o dominio de conhecimento especificado para o programa material manipulado. Aplicado para representação de multiplos programas, e multiplas linguagens sem modificação. As transformações são análogas a transformações geneticas biológicas e realiza por desenvolvedores humanos.

Nenhuma transformação cria novo codigo. Removem duplicata e re ordenam elementos presentes no programa original. O design é baseado em muitos programas existêntes que contem o codigo requerido para implementar comportamento desejado dado a especificação. Com benefício de limitar a transformação do programa, limitando o tamanho do espaço potêncial do programa.

3.1.8 Mutação

As transformações de mutação são copiadas, deletadas e trocadas. Copia duplicatas em sub arvores AST, instruções em vetores e insere e posições aleátorias no AST, imediatamente ápos uma localização escolhida no vetor. Deletar remove de maneira aleátoria uma sub arvore ou vetor de elementos do AST. Troca, troca duas sub arvores ou vetores de elemento da ASt.

3.1.9 Cruzamento

cruzamento recombina duas presentações do programa e produz duas novas representações com elementos de cada pai em processo análogo para processo biológico.

3.1.10 Implementação de requisitos

Cada nivel de diferentes locais de representação requerem como o software sera programaticamente manipulado e qual parte requer ser avaliada.

3.2 Avaliação do fitness

Avaliação fitness é um processo de dois passos: O genótipo deve ser expressado como executável, e após rodar contra um conjunto de testes para o avaliar o fenótipo.

3.2.1 Expressão

Fitness do programa é a propriedade do fenótipo ou comportamento no mundo. Para avaliar o fitness do programa, ele deve estar expresso em um fenótipo.

AST: As representações CIL e Clang AST são serializadas devolta para codigo fonte da familia C. Ferramentas Clang produzem codigo fonte formatado idêntico ao codigo de entrada. Então é compilado e vinculado em executáveis utilizando cadeia de ferramentas do programa original.

Expressões podem falha se a compilaçao ou vinculamento falha.

ASM: As intruções assemble argumentadas constituem genomas ASM e LLVM serializados em arquivos de texto. Vinculado em executavel utilizando cadeia de ferramentas do programa original.

Expressões podem falhar se o vinculador falha.

ELF: A representação do genoma ELF é composto de seções de arquivos ELF, carregado na memoria durante execução. Serializados diretamente em disco executável. Requer ferramentas de construção externa.

Eliminado a compilação e vinculamento para representação nos niveis ASM e ELF aumentam a eficiência da avaliação quando comparada a outros niveis.

3.2.2 Avaliação

Deve ser avaliado por funções de correção.

Durante avaliações funcionais e não funcionais o programa que falha recebe o pior valor de fitness possivel.

Avaliação funcional: O objetivo é determinar se o comportamento do programa é correto ou aceitável. Determinado pelas habilidades para passar em todos os testes no conjunto de testes. Não determina se o programa é semanticamente equivalente ao original.

Avaliação não funcional: Avalia a desejabilidade das propriedas não funcionais da execução do programa. Ferramentas de perfil padrão são utilizados para avaliação.

3.3 Robustez mutacional de software

Robustez é aspecto de pesquisa em engenharia de software, com respeito a confiabilidade e disponibilidade de sistemas de software. Refere-se a funcionalidade de variantes do programa, ou instâncias do software com genomas aleátoriamente mutados.

Investiga-se a adequação dos programas de caso de teste para avaliar a funcionalidade do programa e encontrar a maioria dos casos do conjunto de teste utilizado como proxy para especificações formais. Descobriu-se que os resultados detém varios conjuntos de teste de qualidade, medida utilizando estados e intruções cobridas pelo ASM. Variantes neutras satisfazer o objetivo requisitado pelo programa, definado no conjunto de testes. Podendo diferir em propriedas espelho da funcionalidades como ordem de operações ou comportamentos não especificados, consumo de tempo de execução ou memória.

3.3.1 Design experimental

Define e robustez mutacional de software e descreve as tecnicas usadas para medição em programas de referência.

3.3.2 Robustez mutacional de software

É uma propriedade composta de programas de software P, um conjunto de operações de mutação M e conjuntos de test T: P-¿true,false. Escrita MutRB(P,T,M) a fração da variante P' = m(p), existe m pertence M para t(P) verdadeiro existe t pertence T.

3.3.3 Medição de robustez mutacional de software

Avalia robustez mutacional utilizando CIL, AST e ASM. Os conjunto de teste utilizados são distribuidos com o programa P, e descreve em detalhes sobre a descrição de programas de referência.

Avaliar todas as possibilidades das variantes de primeira ordem é custosa. As tecnicas utilizadas para estimular a robustez mutacional de cada programa de referência.

O programa original executa em conjuntos de teste e cada nó AST ou instrução ASM executada pelo conjunto de teste é identificada. Informações identificadas AST coletada por cada nó instrumental AST para desenhar e identificar durante a execução.

Operações de mutação são aplicadas uniformementes ao longo dos traçados coletados no passo 1.

Cada variante unica compilada com sucesso é avaliada utilizando o conjunto de teste do programa.

A fração de variantes unicas que são compiladas com sucesso e passam em todos os teste no conjunto de teste dado recursos limitados.

3.4 Programas de referência e conjuntos de teste

Os programas do sistemas foram escolhidos para representa conjuntos de teste de baixa a media qualidade. É avaliado a robustez mutacional dos programas mantido pelos desenvolvedores de software e distribuidos com o programa.

Os progrmas de ordenação foram retirados de Codigo Rosetta e selecionados por serem facilmente testados. O conjunto de teste cobrem cada ramo no AST e exercitado cada intrução assemble executável em compiladores assemble utilizando gcc.

O programa Siemens foi selecionado para representa o conjunto de teste atingidos.

3.4.1 Resultados

Relata-se que a taxa de robustez mutacional, analise de robustez mutacional pela qualidade do conjunto deteste, taxonomia de variantes neutras, avaliação de robustez mutacional sobre multiplas linguagens, e avaliação de robustez mutacional sobre os programas de representação.

3.4.2 Taxas de robustez mutacional

Sobre todos os programas de representação descobriu-se que robustez mutacional de 36.8% e minimo de robustez mutacional de software de 21.2%. Valores maiores que preditos. Sugere que programas do mundo real são maiores que implementações alternativas que são descobertas por aplicações de programas de mutação aleátoria.

Há pequena variação na robustez mutacional entre os projetos de software embora uma grande variança em conjuntos de teste de qualidade. Um extremo conjuntos de teste de alta qualidade e declaração completa, ramo e instruções assembly cobre cerca de 20% da robustez mutacional. Os outros extremos, conjunto de teste minimo projetado para ordenação bolha, não checa programas de saida, requer compilação e execução com sucesso, sem falhas, tendo 84.8% de robustez mutacional. Sugere que propriedade inerente do software e não é medida direta da qualidade do conjunto de teste.

3.4.3 Robustez mutacional por conjuntos de teste de qualidade

A diferença em conjuntos de teste são qualidade da fonte do conjunto de teste e quantidade coberta.

Ordenação: O conjunto de teste alavanca a simplicidade de especificos programas de ordenação e implementação cobrindo com dez testes.

Siemens: O conjunto de teste siemens foram retirados da comunidade de teste, desenvolvido por multiplas partes atraves de publicações até atingir estado executável.

Sistemas: O conjunto de teste foram retirando de projetos com fonte aberta do mundo real. Utilizados por desenvolvedores de software para controlar seu desenvolvimento e reflete no alcance do conjunto de teste utilizados.

Quantitativamente a conjunto de teste de classificação fornece 100% do codigo coberto pode nó AST e intruções de nivel ASM.

3.4.4 Taxonomia de variantes neutras

Gerou-se 35 variantes neutras aleátorias de ordenação bolha. Os trações fenotípicos da variantes foram comparadas ao programa original. Agrupados em taxonomia de sete categorias, listadas decrescentemente quanto a frequência. Categoria 1 e algumas variantes da categoria 5 afetam a saida do programa, mudando a impressão do STDOUT ou mudando o valor de retorno de ERRNO final.

Os demais afetaram a saida do programa, categoria 6 e membros da categoria 4 afetaram o comportamento do programa. Categoria 2 incluem a remoção de variantes não necessárias, reoordenando instruções não interativas e mudanças de estado sobrescrita ou não lidas novamente.

Estados das categorias 2,3 e 4 produzem programas com maior robustez mutacional que o original. Incluem mudanças com inserção redundante e guarda de controle de fluxo, e mudanças introduzidas por variantes redundantes.

A maioria das variantes neutras são semanticamentes distintas do programa original. Apenas variantes das categorias 4,5 e 6 não impactam o comportamento da execução podendo ser considerado semanticamente equivalênte.

3.4.5 Robustez mutacional através de multiplas linguagens

Para endereçar a questão do resultado depender de particularidades de certos paradigmas, avaliou-se a robustez mutacional do programa compilado a nivel ASM abrangendo paradigmas de arvore de programação.

3.4.6 Robustez mutacional através de multiplas representações

Os teste e implementações dos programas utilizados estão disponíveis online, como código utilizado no experimento e análise.

Cada algoritmos de ordenação são representados utilizando Clang, CIL, LLVM, ASM e ELF. resultando em 20 representações de cada e mutado aleatoriamente e avaliado 1000 vezes.

O nivel mais alto é CLang possuindo a menor robustez mutacional de software. A baixa robustez mutacional é um efeito da imaturidade da representação e transformações do CLang.

Baixo nivel da representação do programa para o mais alta robustez mutacional de software com exceção da representação ELF é frágil devido a necessidade de manter comprimento geral de genoma. e a incapacidade de atualizar o deslocamento do programa literal.

O fitness é igual ao numero de entradas ordenadas corretamente de conjunto de teste, designadas para cobrir todos os ramos em cada implementação de ordenação.

3.5 Redes neurais de software

3.5.1 Vão de redes neurais

Experimentos mediram que parte de mutações de primeira orem são neutras. Explora-se os efeitos de sucessivas mutações neutras em programas assembly pequeno. Implementa um codigo assembly de inserção ordenada. Aplica mutações aleatórias com representação ASM e operações de mutação. Cada variante é retida se neutra e descartada caso contrário. O processo continua até coletar 100 variantes neutras de primeira ordem. Das variantes é gerada 100 variantes de segunda ordem neutras, através de laços na população de mutantes de primeira ordem, mutações aleatórias individuais contem o resultado se é neutral ou descarta-a. Após as 100 variantes de segunda ordem ser acumuladas, o processo é iterativo para produzir variantes neutras de alta ordem. O processo produz população neutra separa do programa original por sucessivas mutações neutras.

O resultado mostra que o processo de robustez mutacional de software aumenta a distância mutacional do programa original. O resultado corresponde ao desvio da população do perímetro do espaço neutro do programa. o tamanho do programa aumenta com a distância do programa original, sugerem que o programa possa estar conseguindo robustez adicionando instruções inúteis.

3.5.2 Mutantes neutros de alta ordem

Para endereçar questões da densidade de variantes neutras no espaço de possíveis programas, aplica-se multiplas combinações aleatórias de operações de mutação para indivíduos sem verificar neutralidade. Tais variantes de alta ordem, caminham aleatoriamente longe do programa original no espaço de todos os programas possíveis, avalia-se a porcentagem das variantes de alta ordem na distância do programa original, observando a taxa de variantes neutras como mudanças como distância mutacional.

Compara a taxa de variantes neutras de caminhando para a taxa de variantes neutras com uso de funções fitness para garantir que o a caminhada mantenha como programas de rede neural. Confirma empiricamente que a ferramentas de funções fitness de busca por variantes neutras, opôs-se para sugestão alternativa como busca aleatória.

3.5.3 Design experimental

Representação ASM, como população inicial de 2 elevado a 10 é gerado por variantes de ordem neutra de implementações do quicksort em C. A população pode derivar iterativamente selecionando indivíduos da população, mutando-os e inserindo resultados neutros na população. O resultado é uma exploração neutra que mante o tamanho da população de 2 elevado a 10 até as despesas

fitness total de elevação de 2 elevado a 18 foi trocada. O fitness é o caminho da mutação do programa original é salvo para cada teste de variante incluindo variantes não neurais.

A distribuição do numero de indivíduos testados para cada numero da mutação do original durante a busca neutral é salva. Uma população de comparação de variantes aleatórias de alta ordem 2 elevado a 18 é gerada elaborações repetidas da distribuição da distância mutacional de resultando para busca neutral, e para cada distância elaborada gerando variantes aleatórias de alta ordem com mesma ordem, ou distância mutacional do original.

3.5.4 Neutralidade de variantes neutras de alta ordem aleatórias

Modelos decaem exponencialmente o numero de variantes neutras de alta ordem, podendo ser encontrada através de caminhos aleatórios neutros em cada passo intermediário. A alta correlação entre decaída exponencial e a taxa de decaimento da neutralidade ao longo de caminhos aleatórios indicam que muitas variantes neutras de alta ordem são produto de ancestrais neutras de baixa ordem. Pode ser o efeito de alta dimensionalidade em espaço do programa, ou diferença em efetiva dimensionalidade entre espaço neutro e espaço do programa.

3.5.5 Neutralidade comparativa ao longo de aleatoriedade e caminhos guiados

A neutralidade comparativa por distancia mutacional sobre 100 mutações do programa original para ambas aleatoriedades e caminhos guiados por espaço do programa. Por contraste a neutralidade aumenta ao longo do caminho guiado. O maior aumento é viável em variantes neutras de alta ordem encontradas por busca neutra comparada em busca aleatória.

3.5.6 Analise de variantes neutras aleatórias de alta ordem

As variantes de interesse podem ser separadas em redes neurais em reversão de mutações anteriores, e retornam para uma diferente posição em redes neurais do local da saída. Descobriu-se que metade aproximadamente não resultado de reversão ou passo anterior. Historias fitness onde o fitness de seus ancestrais dos valores abaixam a zero ou perto de zero e aumentam lentamente ou salta de volta para o valor neutro.

Tais casos de caminhos aleatórios saem e entram novamente em redes neurais em novos lugares encontra-se porções de rede neural não alcançáveis ou removidas pelo programa original quando o caminho neutro é restrito.

Os resultados indicam que é possível encontrar novas porções de neres neurais utilizando caminho aleatório.

3.6 Espaço do programa e analise de redes neurais

A dissertação desenvolve aplicações para softwares de engenharia automatizada descobrindo versões alternativas de programas existentes através de aplicações

de transformações de programa. Cada representação é define as transformações associadas ao espaço do programa.

Os elementos do espaço do programa são instancias do programa, e distancia entre qualquer dois elementos como distancia calculada utilizando operações de mutação espacial. Cada espaço possui um tamanho, ou numero de programas diferentes que a representação é capaz de especificar.

3.6.1 Tamanho do espaço do programa

Dado um espaço do programa P, um limite de comprimento L e total de U elementos únicos, o tamanho total de P é P (U + 1) elevado a L. Um é adicionada para U contas para remover estados, e o resultado cresce para o comprimento do programa dado cada encaixe no comprimento do programa, pode receber U + 1 valores possíveis.

Pode-se calcular o tamanho do espaço do ASM do programa escrito em MIPS assembler de tamanho L = 10. Cada instrução MIPS tem 32 bits de comprimento, há 2 elevado a 32 possíveis argumentos de instrução MIPS.

3.6.2 Numero de vizinhos

O numero de vizinhos é igual ao numero de transformações que geram programas únicos. É possível que duas operações de mutações diferentes produzem o mesmo programa.

3.6.3 Acessibilidade

Fração do espaço do programa é acessível dado qualquer programa inicial, e o comprimento do menor caminho é acessível da posição. Dado um programa inicial p um espaço do programa P, onde p tem u elementos únicos e P tem e possíveis elementos e comprimento do programa l, o tamanho total de P é modulo P = (e + 1) elevado a l, e o numero de programas acessível de p ou R(p) para ser R(p) = (u+1) elevado a l.

3.6.4 Densidade da rede neural

O tamanho e densidade da rede neural no espaço do programa requer modelo analisável de espaço de programa, utilizado para calcular e estimar.

Define a rigidez do espaço do programa baseado em representações de vetores com comprimento fixo, utilizando operadores únicos de mutação. Calcula-se o numero de novas representações do programa encontrado em sucessivos passo. Prove mapeamento entre espaço rígido e espaço do programa ASM permitindo calculo de projeto de rigidez espacial para espaço de programa gls:asm.

3.6.5 Espaço rígido

O rígido espaço do programa usa o vetor de representação do programa. Cada programa é representado por um vetor com comprimento d. A dimensionabili-

dade sera igual ao comprimento do programa original. O espaço contém apenas programas de comprimento igual ou menor.

Os elementos apareceram em vetores de programa que recebem conjunto E de possíveis elementos com conjunto contendo conjuntos vazios 0. Colocando 0 como índice do programa é equivalente a deletar o conteúdo do índice. O tamanho total de E é e.

Uma operação de mutação é definida no espaço, denominada "replacement". Dado uma variante p, um índice i menor igual a d, e um elemento e pertencente a E, substitui o conjunto p[i] ¡-e.

3.6.6 Expansão gradual no espaço rígido

O numero de vizinhos únicos acessíveis atraves de aplicações de replacement do programa original podem ser calculadas utilizando relação de recorrência. Cada variante é alcansável em d aplicações de replacement.

ni + 1 = ni*(d-i)(e-1)/(i+1)

d é a dimensionabilidade do espaço do programa e sendo e é o numero de elementos.

3.6.7 Mapeando entre ASM e espaço rígido

Há mapeamento muito para um me programas ASM únicos, com multiplos membros do espaço rígido de mapeamento.

O elemento 0 pode ser colocado em qualquer lugar no espaço rígido do programa.

Dado um elemento ASM de tamanho k, onde k menor igual a d e k + j = d, o elemento ASM terá combinação de i j projeções no espaço rígido.

Para cada expansão i, nz,i é o numero de programas com z elementos 0 no passo i. O primeiro programa tem 0 elementos, n0,0 = 1.

3.6.8 Tamanho total das redes neurais

Assume um valor único de robustez mutacional de software mantem cada passo da expansão no espaço rígido do programa. r é a taxa de robustez mutacional do software. Calcula-se o numero total de variantes neutras em ASM. Para cada i menos igual a d, calcula-se o numero de variantes neutras em ni, no espaço rígido.

Trabalhos anteriores de Martinez e Monperrus analisam espaço do programa concluídas e programas de reparo requer mais de maior igual a 5 ou 10 passo para o espaço do programa ser impossível ser encontrado com busca automatizada.

3.7 Discussão

Os resultados contradizem a ideia que mecanismo de engenharia intencional são precisos e frágeis a pequenas perturbações.

3.7.1 Proveniência evolucionária de software

Desenvolvedores de software tem selecionado, reutilizado e eficientemente modifica e robustez de software de ferramentas de desenvolvimentos e padrões de design.

O processo de espelhamento da seleção natural. tem produzido tem produzido características biológicas de software.

3.7.2 Re-interpretação dos teste de mutação

As tecnicas apresentadas imitam as utilizadas em teste de mutação. No entanto realiza interpretações diferentes dos valores e estatura semântica de mutantes neutros.

O espaço envolta do programa é similar ao panorama do fitness, os pontos representam programas distintos sintaticamente, cada programa é associado com a interpretação semântica. Mutação aleatória da representação sintática do programa pode conter efeitos semânticos.

Determina interpretações alternativas dos resultados, para cada especificação existe multiplas implementações corretas não equivalentes. Enfatiza diferentes visualizações do software das tecnicas de teste de mutação, nomeadamente, cada especificação do programa, todas as implementações corretas são semanticamente equivalentes.

Assume-se que existe programa a e b, a não é equivalente a b e ambos satisfazem a especificação S. Sem perda de generalidade, sendo a o programa original e b o mutante de a. T é o conjunto de teste de S. Segundo Offut, há duas possibilidades quando T é aplicado a b. A mutação é assassinada, mas o caso de teste é insuficiente ou a mutação é funcionalmente equivalente. O ex caso é impossível pois b é assumido por ser uma implementação correta de S e não deve ser assassinada pelo conjunto de teste S. O caso é impossível pois assumiu-se que a é diferente de b. Contradição, existe a e b satisfazendo a especificação S, a igual a b ou existe especificação S.

Mutação neutra não equivalente requer um atenção dos desenvolvedores. Cada mudança requerida ao conjunto de teste do programa e possíveis mudanças a especificação do programa e o programa original. O problema de diferenciação de problemas de mutantes equivalentes e não equivalentes é denominado a problema de mutante equivalente.

Operações de mutação Mothra são especificadas para a linguagem de programação Fortran mais especificas. Incluem operações para constantes, variáveis ou troca de listas podendo ser implementadas utilizando combinação da copia e inserção, inclui operações análogas para deletar e copiar.

3.7.3 Legibilidade das transformações

O processo de comunicação é importante para revisão manual de adaptações do programa evoluídos. A legibilidade das transformações do programa diferem para cada nivel de representação.

Clang, mutações AST a nivel Clang prove a melhor legibilidade. Mudanças ao Clang AST podem ser automaticamente convergido para diferentes níveis de origem, propriamente indentado e pronto para desenvolver outra revisão manual.

Cil, mutação ASM nesses níveis são aplicada a LLVM e IR ou codigo assembly compilado. Mudanças nas variantes do programa podem ser representada como assembly e diferentes IR. Permitindo revisão manual.

Compiladores prove opções para mapiar especificas instruções em assembly para linhas especificas do código no programa original.

Mutação ELF em representações ELF tomam lugar na porção executável de arquivos ELF traduzidos em instruções assembly utilizando desmontamento.

3.7.4 Avaliação funcionalidade contra não funcionalidade

Avaliações não funcionais levam a suavizadores de paisagem fitness acessível por algoritmos de busca evolutiva. Leva a uma paisagem fitness escalonada composta de planaltos planos que não provem guia para tecnicas de computação evolutiva.

Propriedades funcionais e não funcionais de software análoga para discreta e continua, ou quantitativa, respectivos traços de organismos biológicos. Traços discreto de organismos biológicos expressam fenótipos em infinitos números de classes discretas e controlada por genes.

4 Aplicações: diversidade do programa

Tecnicas automatizadas de programas de geração de variantes com tempo de execução ou diversidade fenotípica são coletadas artificialmente. Torna sistemas computacionais mais seguros contra ataque dificultando encontra, reproduzir, e transferir exploração entre maquinas.

Demonstra-se que populações de diversas variantes neutras podem prover diversidade suficientemente para reparar defeitos latentes no programa.

Introduz aplicações de redes neurais na geração automatizada da diversidade implementada.

4.1 Metodologia

Acessa o grau no qual a população de variantes neutras do programa pode reparar falhas latentes no programa original. Procegue enviando falhas latentes em softwares do mundo real, gerando população de variantes do software com conhecimento das falhas semeadas, e avaliando o grau de reparo do software.

4.2 Resultados

Descobriu-se que muitos dos programas com cinco falhas semeadas e 5000 variantes neutras e ao menos uma variante neutra proativamente reparado uma das falhas.

Os tipos mais comuns de falhas reparadas mais comuns assemelham-se as operações de mutação. Não foi capaz de identificar características com falhas reparáveis e não reparáveis.

Os experimentos incluem cinco falhas latentes por programa. Programas implantados tem mais que cinco defeitos pendentes.

4.3 Discussão

O uso de robustez mutacional é análoga a teste de mutação de software com diferenças críticas, mutantes neutros são retidos em vez de examinados manualmente. O conjunto de teste é não aumentado para assassinar todos mutantes, o conjunto de operações de mutação considerados diferentes. A prática comercial de teste mutacional foi limitada por esforço significante requirido para analise de mutantes que passam no conjunto de teste. Os mutantes devem ser classificados manualmente e classificado como falha ou superior ao programa original.

A metodologia proposta pode prover alternativas aos teste de mutação tradicional, mantendo população de variantes neutras. Quando uma falha é encontrada no programa original, será detectada na execução por todas variantes contra o erro e verificação se os membros da população são anormais com respeito a população. Variantes não falhas precisam ser analisadas apenas para sugerir reparos.

5 Aplicação: Assembly e reparo do programa a nivel binário

Reparo automatizado do programa evolutivo a nivel CIL, AST demonstra um alcance da aplicabilidade. Representação do programa em baixo nivel, oferece características desejáveis incluindo generalidade para linguagens alem de C, reduz os requisitos do codigo fonte, tempo de expressão mais e cobertura do espaço possíveis do programa.

5.1 Localização de falhas

É o processo de determinar a localização das falhas do programa. Representações do programa AST confiou na localização de falhas pra operações com foco em mutação, relacionadas ao defeito a ser reparado. O processo utiliza tracos sintetizado da execução do programa de entradas boas e ruins para estimar a probabilidade da porção do programa estar relacionada com a falha.

A tecnica requer estados AST de nivel de instrumentação do programa. Soluções de instrumentações análogas são problemáticas para representações do programa de baixo nivel, não tendo acesso a fonte do programa.

Desenvolveu-se dois métodos de criação de perfil, aplicados para assembler arbitrários em programas ELF, um arsenal de tempo determinista pesado e uma técnica de amostragem estocástica de peso leve. Compara-se os resultados das tecnicas, descobrindo os que são comparáveis e concluiu-se que a amostragem técnica é preferível.

A técnica de amostragem determinística utiliza uma habilidade de tempo de execução baseada em rastreamento, coletando os valores obtidos pelo contador do programa durante a execução. Preferível para programas pequenos.

Estocagem amostral usa perfil de todo perfil do sistema Linux para provar o valor do contador do programa durante a execução. Retorna um contador do numero total de instruções na prova do programa. A amostragem controla o fluxo e é vulnerável a lacunas e sobre amostragem de instruções. A granulidade fina estima o numero total de instruções executadas.

Para compensar, aplicou-se uma convolução gausseana para endereçar com raio de 3 intruções assembler. O resultado suavizado endereça cas instruções x e a soma ponderada G(x) a contagem de amostras brutas e 3 vizinho (x + i) de cada lado.

5.2 Benchmarks

Para avaliar a efetividade do reparo nos niveis ASM e ELF, programas benchmark foram usados. A taxa de sucesso e busca foram coletadas e comparadas com trabalhos anteriores.

5.3 Eficácia

Comparou-se a habilidade do nivel de representação ASM e ELF para reparar defeitos. Descobriu-se sucesso geral comparável a taxas entre níveis de reparo

ASM e ELF.

Reduziu-se o numero de avaliações fitness requeridas para encontra um reparo.

5.4 Eficiência

Tendo encontrado eficiência entre os níveis, considou-se a eficiência do processo de reparo por nivel de representação. Em geral representações ASM e ELF da performance do programa de reparo automático, mais eficiente que o maior nivel do CIL AST.

Tempo de execução: O tempo de execução do processo de reparo é dominado pelo tempo gasto para avaliar o fitness. O tempo gasto incluem avaliação fitness do tempo requirido para executar o conjunto de teste, e o tempo requirido para expressar o programa executável, o baixo nivel de representação são mais eficientes por não ser necessária compilação.

Disco utilizado: O disco utilizado durante o reparo difere entre niveis da representação. A pegado do disco reduzido dos níveis ASM e ELF, devido a não necessidade de compilação e vinculação.

Memória: A memoria do processo de reparo é o espaço requirido para manter a população de candidatos a variantes de reparo. O é o tamanho da representação do programa na memória.

5.5 Discussão

Descreve experimentos da habilidade do nivel ASM e ELF da representação do programa para reparo de falhas de programas anteriores de reparo a nivel CIL AST.

Descobriu-se ser efetivo para programas de repara a nivel CIL AST, mais eficientes em termo de consumo de memoria.

6 Aplicações: corrigindo o executável de origem próxima

A habilidade para manipular arquivos ELF diretamente evita o acesso a recursos do desenvolvedor de software como codigo fonte ou cadeias de ferramentas. Demonstrou-se a aplicação reparando multiplas vulnerabilidades de segurando em roteadores sem fio.

Descreve-se como o método de reparo Genprog, pode ser aplicado a novos casos de uso.

Demonstra-se do método reparando exploits desconsiderados em versões 4 de NETGEAR's WNDR3700 roteador sem fio. Sem uso de testes de regressão, descobriu-se que 80% geraram reparo automáticos.

6.1 Descrição dos exploits

Descreve-se duas versões do NETGEAR WNDR3700. A popularidade de roteadores torna vulnerabilidade do sistema generalizada.

O binário implantado é inseguro por causa:

1. Qualquer URL começando com "BRS"burla a autentificação.
2. Qualquer URL incluindo a substring "unauth.cgi"ou "securityquestions.cgi"burla a autentificação.

Muitas páginas começam com "BRS", provendo ataques com acesso a informações pessoais.

6.2 Método de reparo automático

A técnica de reparo utilizada consiste de 3 estágios:

1. Extrair o binário de forma executável do firmware e reproduzir a exploração.
2. Utilizar tecnicas evolucionárias para buscar por reparos aplicando aleatoriedade para o binário.
3. Construir os casos de teste, para aprimorar a qualidade de reparo de candidatos não satisfatórios.

6.2.1 Extração firmware e virtualização

NETGEAR distribui firmware com imagem completa do sistema para roteadores WNDR3700, incluindo arquivos de roteadores que contém vulnerabilidade net-cgi executável. Extraiu-se os arquivos do sistema com ferramenta de extração binwalk firmware, escaneando os dados binários em arquivos firmware, procurando assinaturas identificando seção de dados embutidos, incluindo squashfs armazenando os arquivos de sistema.

Roteadores executam em arquitetura MIPS big-endian, requerendo emulação para reproduzir a exploração e avaliar o reparo dos candidatos. Utilizou-se sistema emulador QEMU para emular arquitetura leve com Linux Debian. Os arquivos extraídos do roteador é copiado no emulador MIPS do Linux. Os diretorios são montados dentro dos arquivos extraídos e vinculados aos diretórios na máquina virtual.

6.2.2 Programa de reparo automatizado e arquivos ELF

O algoritmos de reparo construiu uma população com 512 variantes do programa. A população é evoluída por processo iterativos de avaliação, seleção, mutação e cruzamento em uma versão do programa original é encontrado para reparar a falha. Traços da execução foram coletadas durante a execução e utilizada como localização de falhas para bias de mutações aleatórias em direção ao programa que contém a falha.

6.2.3 Desafio: Mutação de binários despojados

Programas executáveis para Unix e sistemas embutidos são distribuídos como arquivos ELF. Contendo cabeçalhos e tabelas contendo dados administrativos e seções do código e dados do programa. Três elementos do arquivo ELF são cabeçalho de ELF, a tabela do programa. Cabeçalhos indicam as tabela da seção e do programa, as tabelas de seção armazenam informações do layout da seção nos arquivos ELF do disco, a tabelas do programa armazenam informações de como copiar seções do disco na memória para execução.

Operações de mutação devem tomar arquivos do programa sem corromper a estrutura dos arquivos ou quebrando os endereço do codigo nos dados do programa.

6.2.4 Testando regressão em demanda

O programa de reparo depende da habilidade de avaliar a validade das variantes do programa. As mutações são aleatórias não considerando preservar a semântica do programa. São para criar novas falhas ou vulnerabilidades.

Mutando programas sem segurança na rede do conjunto de teste, os reparos evoluídos introduzem significado de regressão. Aplicando processo rigoroso após o reparo primários ser identificado, as regressões são removidas. Reduz a diferença entre o reparo evoluído e o original como possíveis debugação delta.

6.3 Reparando a exploração do NETGEAR

Descreveu-se a configuração experimental utilizada para testar as tecnicas de reparo de vulnerabilidade do NETGEAR WNDR3700.

6.3.1 Metodologia

Os reparos foram realizados em uma maquina de classe servidor. Utilizou-se equipamentos de teste para avaliar o fitness de cada variante do programa.

6.3.2 Avaliação fitness

O algoritmos de reparo usou 32 threads para evaliação fitness paralela. Pareada com simples QEMU VM com testes de fitness.

A estrutura de teste incluir hospedeiro e script de teste de convidados. O hospedeiro executa no servidor realizando reparos e o visitante executa a maquina virtual MIPS. O hospedeiro copia a variante do net-cgi executável para a maquina virtual do visitante onde o visitante testa o script executando os comandos net-cgi e reportando os resultados do passar, falha ou erro dos teste.

6.3.3 Parâmetros de reparo

O reparo utiliza os parâmetros, a população maxima é 2 elavado a 9 indivíduos, seleção é realizada utilizado torneio de tamanho de dois. A população supera a população maxima, o individuo é escolhido para desejo utilizando torneio onde o menos fitness é selecionado para evitar desejo.

O requisito de memoria incremental pela população é compensada pelo uso de estado estável do algoritmo evolucionário.

6.3.4 Resultados experimentais

Relatou-se resultados do tempo tomado para gerar reparos, a eliminação de localizações falsas, e o impacto do processo de minimização, respeita o tamanho do reparo por diferença de byte.

6.3.5 Execução dos reparos

Em 8 de 10 execuções do algoritmo, os testes de exploração foram suficientes para gerar reparo satisfatório.

6.3.6 Reparo sem localização de falhas

Em cenários NETGEAR não foi utilizado formas de localização de falhas, AS limitações, como no Genprog, das transformações para porções do programa exercitado por entradas do programas falhas, prevenindo erros serem encontrados para suas modificações necessárias fora das entradas falhas.

6.3.7 O impacto da minimização

Em alguns casos o reparo sugerido inicialmente não é satisfatório. O reparo evoluído inicial difere do programa em cima de 200 localizações na media dos dados do programa ELF. A discrepância é dada pela acumulação de candidatos editados em porções não testadas dos dados do programa.

6.4 Discussão

Os resultados apresentados abrem a possibilidade de usuários repararem a vulnerabilidade do software em fonte fechadas do software sem informações especiais para auxiliar o fornecedor do software.

Ha ressalvas associadas com trabalho inicial. Demonstrou-se reparo a executáveis únicos. Os resultados não aparentam serem baseados em propriedades únicas da exploração do NETGEAR. O sucesso da função de reparação é dada pelo impacto da minimização e robustez mutacional de software.

7 Aplicação: Otimizando propriedades do programa não funcional

A aplicação apresentada considerou apenas propriedades funcionais de software. Descreve uma aplicação de exploração evolucionária de redes neurais parar otimização de propriedades não funcionais, e busca o panorama fitness definido por propriedade não funcionais para conjunto de guias de busca evolucionária.

Tempo de execução requerido para software são dominadas por propriedades não funcionais complexas.

Propriedades do tempo de execução como consumo de energia são resultado de iterações complexas com detalhes do hardware e ambiente de execução do software, limitando a efetividade das tecnicas gerais.

Descreve GOA, uma técnica de otimização de pos compilação alavancando busca evolucionária para automaticamente encontrar maquinas, ambientes e carga específica de trabalho no espaço do codigo do programa assembler. A efetividade é medida reduzindo o consumo de energia para aplicações benchmark PARSEC em plataformas de hardware diferentes. GOA encontrou hardware e optimizou a carga específica de trabalho e reduziu o consumo de energia em 20%.

7.1 Algoritmo de otimização genético

GOA é uma técnica pos compilação de otimização dirigida para carga específica de trabalho. A entrada é analisada em uma representação de nivel ASM. GOA utiliza carga específica de trabalho provido pelo desenvolvedor de software para exercitar otimizações candidatas para avaliar candidatos a otimização para funcionalidade e medida como propriedade de execução.

O nivel de representação ASM é extraida do processo de construção, designado um fitness e usado para semear um população de variantes do programa. Busca por candidatos a programas de otimização. Cada iteração da busca: seleciona a otimização candidata para a população, transforma a, vincula os resultados em um executável, executa os resultados executáveis contra carga específica de trabalho, coleta informações pessoais do programa, para programas corretamente processados, combina as informações em avaliações fitness utilizando funções fitness, reinsere a otimização e pontuação fitness na população, seleciona os melhores indivíduos encontrado na busca e minimiza com respeito ao programa original. GOA retorna um assembler que pode ser revisado manualmente or aplicado ao programa original para produzir versões otimizadas.

7.2 Entrada

GOA requer três entradas, o codigo assembler do programa a ser otimizado, um conjunto de teste de regressão que capture funcionalidade, e alvos de otimização mensurável.

O programa a ser otimizado apresenta um arquivo assembly nico.

O desenvolvedor deve fornecer uma função fitness, onde GOA tentara otimiza-la.

7.2.1 O algoritmo GOA

GOA usa um algoritmo de evolucionário de estado estável. Difere de aplicações de computação evolucionária para problemas do mundo real. Em vez de trocar a população por passo discretos, algoritmos evolutivos de estado estável operam um indivíduo candidato a otimização por seleção, transformação, avaliação e inserção.

A população é inicializada com n copias do programa original. O espaço de busca de possiveis otimizações é explorado, transformando o programa utilizando mutação aleatória e cruzamento. No cruzamento, dois pais com alto fitness são escolhidos por seleção torneio e combinados.

7.3 Avaliação

Avalia empiricamente a habilidade do GOA de reduzir o consumo de energia entre populações de aplicações benchmark e plataformas hardware e avaliada se foi mantida a funcionalidade do programa.

Avalia-se a efetividade do GOA contra aplicações benchmark PARSEC.

7.3.1 Benchmarks

Utiliza-se conjuntos de programa benchmarks conjuntos PARSEC. Avalia-se GOA em todas aplicações PARSEC que produzem saídas testáveis e incluem mais de uma entrada. Saída testável assegura que a otimização mantem a funcionalidade. Multiplos conjuntos de entrada são requeridos pois GOA utiliza uma entrada durante a otimização e outras como teste apos a otimização.

7.3.2 Testes realizados

Utilizou-se conjuntos de teste para avaliar a otimização encontrada pelo GOA.

Cada teste foi executado utilizando o programa original e seus resultados validaram a saída do programa otimizado. Utilizou-se combinações de flags aceitáveis pelo programa original.

7.3.3 Modelo de energia

A função fitness utiliza modelo de energia linear baseado em processos específicos de contadores de hardware.

Os valores foram obtidos empiricamente para cada arquitetura testada, utilizando coletores de dados entre execuções de cada benchmark PARSEC.

7.3.4 Resultados da redução de energia

O numero de divergências entre a versão otimizada e a original, indicam o tamanho do executável compilado.

GOA encontrou otimizações no consumo de energia em muitos casos.

7.3.5 Resultados de programas de correção

A força do GOA inclue a habilidade para mudar a semântica do programa e customizar o programa para a maquina de treinamento, ambiente e carga de trabalho. No entanto existe a possibilidade das mudanças mudarem o programa semanticamente.

Descobriu-se que a maioria das otimizações encontradas preservou completamente o programa.

7.3.6 Caso de estudos

Descreve exemplos, diferentes tipos de optimização de energia encontrado pelo GOA.

Blackscholes implementa uma model de equação diferencial parcial financeiro. A execução é tap rapida, que benchmarks artificiais adicionam laçoes exteriores que executam o modelo multiplas vezes. A redundância de calculação não são detectadas pelo analisador do compilador estático padrão. GOA descobriu a redundância do calculo em hardwares AMD e Intel.

GOA também encontrou otimizações especificas de hardware em swaptions.

7.4 Discussão

Descreve o algoritmo de otimização genética GOA, e aplicações de busca com redes neurais definidas pela propriedades do programa funcional, para otimizar as propriedades do programa não funcionais. GOA é capas de reduzir significativamente o consumo de energia.

8 Direções futuras e conclusão

Descreve uma investigação empírica da robustez de softwares do mundo real em mutações aleatórias e caracterizar os resultados da redes neurais do espaço do programa. A robustez é o resultado do desenvolvimento evolucionário do eco sistema de desenvolvimento de software e indica a existência de ferramentas evolutivas para manutenção e aprimoramento.

As técnicas demonstradas aprimoram automaticamente a robustez, correção e eficiência.

8.1 Desafios

Interface de desenvolvimento: inclui as áreas relatadas de preservação semântica, objetivo de desenvolvimento de comunicação para processo evolutivos, e integração no ciclo de desenvolvimento.

Funcionalidade nova: foca em aprimorar funcionalidades existentes de software, atraves de defeitos de caminhos e vulnerabilidade ou propriedades do tempo de execução do mudanças não funcionais.

8.2 Oportunidades

Detalha o numero de propriedades para trabalhos futuros.

8.2.1 Verificação

O obstaculo de incorporar as tecnicas evolutivas em software padrões de desenvolvimento, se da pela falta de verificação formal dos efeitos ou das transformações evolucionárias limitadas do programa. Opções de trabalho na área: incorporar ferramentas de emergente e de longa data. Podendo ser aplicada a aplicações apresentadas no trabalho.

8.2.2 Analise de divergência

AS transformações evolucionárias são apresentadas como divergência padrões do codigo fonte, assembler ou nivel binário para AST, ASM e ELF.

Há tecnicas emergentes e modelos para provar propriedade do codigo assembler, que pode ser aplicadas para provar equivalência semântica entre sequências de codigo assembler.

8.2.3 Analise estática

Ferramentas automatizada para análise estática de codigo são amplas para pesquisa e desenvolvimento de software. Ferramentas de análise estatística incluem produtos comerciais. ferramentas de codigo aberto, procurando por falhas medindo padrões ou fluxo de dados pelo programa.

As ferramentas podem ser incorporadas nas tecnicas evolutivas, como componentes da função fitness ou processo de post final. Garantindo métricas de qualidade de codigo ou aprimorando a complexidade durante a execução, ou garantir valor de qualidade minima no fim da execução.

8.2.4 Analise dinâmica

Para resultados de programas de teste, métodos sofisticados de avaliação dinâmica pode aprimorar a confidencialidade das variantes do programa evoluído.

8.2.5 Avaliação de funções contínuas

Descreve métodos para suavização automática de paisagem fitness para guias providos para tecnicas de busca evolucionária, por porções de paisagem fitness.

8.2.6 Cruzamento heterogêneo

Mover funcionalidades entre pedações de software é importante para funcionalidades novas de evolução. Cruzamento é o processo evolutivo que mistura material genético entre indivíduos.

Populações homogêneas que compartilham ancestrais tende a ter funcionalidades equivalentes. Não é uma técnica apropriada para compartilhar funcionalidades enter indivíduos heterogêneos.

possível abordagem é trocar o uso de pontos de cruzamento no genoma, como busca de localização de cruzamento em cada pai que compartilha contexto sintático similar.

8.2.7 Endurecimento evolucionário

Executáveis binários de proveniência desconhecida, podem ser significantes para segurança ou validar a confiabilidade do sistema. Ou invalidar plataformas.

8.3 Conclusão

Nos ultimos cinquenta anos desenvolvedor tem selecionado, reutilizado e modificado ferramentas de desenvolvimento de software, codigo e padrões de design. Através de espelhamento de seleção natural, produziu-se software com características biológicas.

O trabalho apresenta problemas do mundo real acessível a modificações por transformações aleatórias. Podendo explicar o sucesso de tecnicas evolutivas de reparo de software.