UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E
ENGENHARIA DE COMPUTAÇÃO

FELIPE ZORZO, LUCAS PIZZO E WILLIAM WILBERT

RELATÓRIO FINAL SOBRE OS RESULTADOS DA INJEÇÃO DE FALHAS E DO HARDENING

Relatório final sobre as injeções de falhas realizadas para a disciplina de Fundamentos de Tolerância à Falhas, contendo medida PVF (program vulnerability factor) de três aplicações escolhidas, bem como a análise de o quão críticas foram as falhas e uma comparação dos resultados dessas três aplicações. Além de análise do resultado de técnicas de hardening para cada uma das aplicações.

Porto Alegre
2018

SUMÁRIO

1 INTRODUÇÃO

Este relatório busca explorar a vulnerabilidade de três aplicações através da medida do PVF (Program vulnerability factor) usando a ferramenta de injeção de falhas CAROL-FI para realizar a medição. Em seguida, cada aplicação foi modificada com técnicas para detecção de SDCs, com objetivo de explorar como técnicas de tolerância a falhas podem ser aplicadas em diferentes cenários. Cada aplicação passou por duas modificações, uma que usa duplicação total protegendo toda a execução e outra que protege apenas alguns trechos das aplicações, para que possamos comparar o quão eficiente cada versão foi em detectar SDCs e qual foi o custo computacional extra que essas modificações requisitaram.

As três aplicações escolhidas foram:

1. Alinhamento de sequências com Smith-Waterman.
2. Compressão de áudio.
3. Regressão linear.

A seguir cada uma será explicada mais detalhadamente, bem como os resultados de suas medidas de vulnerabilidade e como se comportaram após a aplicação de técnicas de tolerância a falhas. O relatório será concluído comparando os resultados dessas três aplicações buscando respostas para eventuais diferenças.

2 ALINHAMENTO DE SEQUÊNCIAS COM SMITH-WATERMAN

2.1 O algoritmo

O algoritmo de Smith-Waterman é um algoritmo de alinhamento de sequências local, utilizado especialmente na área de bioinformática para a comparação de sequências de nucleotídeos ou sequências de proteínas. Por ser um algoritmo de alinhamento local, não busca o melhor alinhamento em toda a extensão de duas sequências sendo comparadas: ele tenta apenas alinhar regiões de alta similaridade.

O algoritmo funciona por meio da criação de uma matriz S de tamanho MxN, em que M é o comprimento da primeira sequência e N o comprimento da segunda sequência. Inicialmente é definido, pelo usuário do algoritmo, um valor de *match*, *mismatch* e *gap*. Se o usuário não informar valores, os utilizados em nossa implementação, por padrão, são 1 para *match*, -1 para *mismatch* e -2 para *gap*. A primeira coluna e a primeira linha da matriz são completadas com 0. As demais posições da matriz são completadas de acordo com a seguinte regra:

$$S_{i,j} = max \begin{cases} 0 \\ S_{i-1,j-1} + s(a_i, b_j); \\ S_{i,j-1} + w; \\ S_{i-1,j} + w; \end{cases}$$

Nela, $s(a_i, b_j)$ é o valor de match ou mismatch. Para definir qual dos dois usar, basta comparar os caracteres que deram origem à coluna j e à linha i da matriz; se forem iguais, utiliza-se *match*, se forem diferentes utiliza-se *mismatch*. Por fim, w é o valor de *gap*.

Com a matriz S em mãos pode-se, finalmente, alinhar as sequências. Para tanto, partimos da posição da matriz de maior valor. Caso existam várias posições com o mesmo maior valor, escolhemos o mais para baixo e para a direita. A partir daí, voltamos para uma de três posições: (i-1, j-1), (i, j-1) ou (i-1, j), de acordo com a direção de movimento utilizada para construir a matriz a partir da regra da imagem acima. Essa volta continua até chegarmos em um elemento da matriz com valor igual a zero.

O caminho obtido para chegar até a célula de maior valor nos dá, então, como deve ser feito o alinhamento. Caso haja um movimento na diagonal, significa que os caracteres que deram origem à linha e coluna atual da matriz são postos no alinhamento da sequência que

originou as linhas e no alinhamento da sequência que original as colunas, respectivamente; caso haja um movimento para baixo, significa que iremos botar um "-" no alinhamento da sequência que deu origem às colunas e o caractere que representa a linha atual será posto no alinhamento da sequência que deu origem às linhas; por fim, caso haja um movimento para a direita, significa que iremos botar um "-" no alinhamento da sequência que deu origem às linhas e o caractere que representa a coluna atual será posto no alinhamento da sequência que deu origem às colunas. Ao chegar na célula de maior valor da matriz, o processo de alinhamento acabou.

2.2 A implementação:

O algoritmo foi implementado inteiramente em C. As bibliotecas utilizadas foram: stdlib.h, para ter acesso a malloc(), calloc() e free(); stdio.h, para ter acesso a snprintf(), fopen() e fclose(); string.h para ter acesso a strlen(); e, por fim, unistd.h para obter acesso a getopt() e permitir que o programa receba argumentos pela linha de comando.

2.3 A injeção de falhas:

Usando o CAROL-FI foram injetadas 50104 falhas divididas entre três modelos de falhas. Os efeitos de cada modelo foram os seguintes:

Tabela 2.3.1 - Efeito das falhas injetadas.

Efeito das Falhas	Quantidade	PVF (%)
Masked	36838	73.5231
SDCs	1168	2.3312
Crashes	6807	13.5857
Hangs	5291	10.5600

Tabela 2.3.2 - SDCs para cada modelo de falhas.

Modelo de falhas	SDCs
Random bit-flip	641
Single bit-flip	274
Double bit-flip	253

Tabela 2.3.3 - Crashes para cada modelo de falhas.

Modelo de falhas	Crashes
Random bit-flip	2476
Single bit-flip	2169
Double bit-flip	2162

Tabela 2.3.4 - Hangs para cada modelo de falhas.

Modelo de falhas	Hangs
Random bit-flip	1492
Single bit-flip	1926
Double bit-flip	1873

2.4 Análise das falhas:

Analisando os resultados obtidos com o CAROL-FI é possível perceber que cerca de 75% dos SDCs ocorreram devido a falhas injetadas em variáveis que armazenam algum score: *gap_score*, *match_score*, *mismatch_score* e *max_score*. Os demais SDCs provêm de alterações em variáveis diversas, com destaque para os índices utilizados para acessar posições da matriz, os ponteiros que apontam para as strings utilizadas, as variáveis que

armazenam o tamanho de cada uma das duas sequências sendo alinhadas e, por fim, os ponteiros que apontam para os arquivos que contém as sequências a serem alinhadas.

Tabela 2.4.1 - Número de SDCs causados pelas cinco variáveis de maior interesse.

Variável	**SDCs**
gap_score	301
match_score	246
mismatch_score	226
max_score	101
i	47

O programa sendo analisado é *memory bound*, logo faz uso extensivo de ponteiros. Portanto, é de se esperar que ocorram diversos *crashes* devido a *flips* em ponteiros. Esses ponteiros passam a apontar para regiões da memória inválidas, causando *crash*. Por isso mais de 13% das falhas injetadas resultam em *crash*. As principais variáveis afetadas são os ponteiros para as *strings* utilizadas, para os arquivos a serem lidos e o ponteiro para ponteiros que representa a matriz S. Também merecem destaque as variáveis utilizadas para acessar posições de estruturas como *strings* e a própria matriz: flips nessas variáveis podem facilmente fazer o programa tentar acessar um endereço de memória inválido, resultando em *crash*. Por fim, duas variáveis utilizadas pela biblioteca que controla a passagem de argumentos pela linha de comando geraram diversos *crashes*, mas não estão no escopo de análise desse trabalho.

Tabela 2.4.2 - Número de *crashes* causados pelas seis variáveis de maior interesse.

Variável	SDCs
x_num	595
y_num	558
i	540
str_x	515
str_y	486
matrix	479

Quanto aos *hangs*, cerca de 97% deles foram gerados por *flips* nas variáveis que controlam o valor de *mismatch*, *match* e *gap*. Isso faz completo sentido, uma vez que, após a matriz estar construída e acharmos o maior valor dela, teremos que fazer um caminhamento de volta até achar uma posição de valor zero. Para decidir se o caminhamento de volta irá ir para cima, para a esquerda, ou para a diagonal devemos descobrir qual das igualdades é satisfeita:

1. valor da posição acima + valor de gap = valor da posição atual;
2. valor da posição à esquerda + valor de gap = valor da posição atual;
3. valor da diagonal à esquerda e acima + valor de match ou mismatch (depende se os caracteres que deram origem à diagonal são iguais ou não) = valor da posição atual.

Claramente, se algum dos valores de *match*, *mismatch* ou *gap* for alterado durante a execução do programa, possivelmente nenhuma das igualdades será satisfeita e não haverá caminhamento na matriz, gerando um *loop* infinito que causa *hang*. Outras variáveis de destaque que causam *hang* são as utilizadas para definir o máximo entre quatro valores, utilizado pela regra demonstrada na explicação do algoritmo, e os ponteiros que apontam para as *strings* (eles são utilizados para definir se devemos utilizar o valor de match ou mismatch, então podem acabar afetando as mesmas igualdades já citadas).

Tabela 2.4.3 - Número de *hangs* causados pelas três variáveis de maior interesse.

Variável	SDCs
match_score	1746
mismatch_score	1731
gap_score	1653

Conclui-se, portanto, que a vulnerabilidade do programa é causada, em grande parte, devido às variáveis *match_score*, *mismatch_score* e *gap_score*. Além disso, como o programa utiliza bastante memória e, consequentemente, ponteiros, ele acaba por ser extremamente suscetível a *crashes* quando há falhas nesses ponteiros. O programa se mostrou, por outro lado, bastante tolerante a falhas: mais de 74% delas foram mascaradas. Isso provavelmente se deve ao fato de que as falhas alteram algum valor que vai parar na matriz, mas que nunca é utilizado durante o alinhamento. Como as sequências testadas são extremamente grandes, podendo ter 15000 caracteres cada uma, a matriz resultante é gigante, podendo ter até 225 milhões de posições. Com isso, é claro que a grande maioria das posições nunca será acessada na hora de alinhar as sequências e, portanto, falhas nesses valores não afetarão o resultado.

2.5 Hardening por duplicação total:

O primeiro passo para proteger o algoritmo contra SDCs foi duplicá-lo inteiramente. Isso significa que as variáveis globais para *score* são duplicadas e testadas a fim de detectar a ocorrência de SDC em alguma delas; além disso, o próprio algoritmo é executado duas vezes e ambas as respostas são comparadas para detectar a possível ocorrência de SDC em alguma das execuções. Ressalta-se que a variável *max_score* foi retirada de todas as técnicas de *hardening*, uma vez que é apenas uma informação adicional, não fazendo realmente parte do algoritmo de Smith-Waterman.

Foram injetadas 13010 falhas com essa técnica. Dessas falhas, 2131 causaram SDCs, todos detectados. Nota-se que houve um aumento de sete vezes na porcentagem de SDCs ocorridos. Isso se deve ao fato de que, para qualquer falha injetada em alguma das variáveis de *score*, considera-se que houve um SDC detectado, mesmo que a falha seja mascarada na resposta do algoritmo.

Optou-se por essa técnica conservadora pois as duas execuções do algoritmo de Smith-Waterman utilizam as mesmas variáveis de *score*, que são globais; logo, se apenas fossem comparadas as respostas das duas execuções, possivelmente ambas teriam a mesma resposta defeituosa e o SDC não seria detectado. Para evitar isso, as variáveis de *score* são comparadas individualmente ao fim das duas execuções do algoritmo de Smith-Waterman.

Tabela 2.5.1 - Efeito das falhas injetadas.

Efeito das Falhas	Quantidade	PVF (%)
Masked	7518	57.7863
SDCs	2131	16.3797
Crashes	1607	12.3520
Hangs	1754	13.4820

2.6 Hardening por duplicação seletiva inicial:

Com base na tabela 2.4.1, optou-se por proteger as quatro variáveis restantes após a eliminação de *max_score*: *gap_score*, *match_score*, *mismatch_score* e i. Essas são as variáveis que inicialmente foram consideradas críticas para o algoritmo, já que são as que mais causam SDCs.

Tal técnica de *hardening* foi implementada por meio da duplicação dessas variáveis críticas. Ao fim da execução do algoritmo, em caso de diferença entre alguma das variáveis, é considerado que um SDC foi detectado. A única variável que é protegida de forma diferente é a variável i, já que ela é reutilizada em diferentes *loops* durante o algoritmo para percorrer a matriz. Portanto, a cada vez que i é incrementada, incrementamos também sua duplicação e as comparamos: em caso de diferença, é acionada uma *flag* que indica que um SDC foi detectado. Essa *flag* também é duplicada e testada ao fim do algoritmo.

Foram injetadas 20895 falhas na implementação com essa técnica. Delas, 5306 causaram SDCs, dos quais cerca de 99.13% foram detectados. A principal variável que não teve seus SDCs detectados foi j, também utilizada em diversos *loops* do algoritmo. Nessa técnica optou-se por não protegê-la pois imaginou-se que o *overhead* de tempo sofreria drasticamente, já que ela é utilizada dentro de dois *loops* aninhados, enquanto i é utilizada

sempre no *loop* mais externo. Outras variáveis que provocaram SDCs não detectados foram as que armazenam as *strings* sendo alinhadas: considerou que o *overhead* de espaço para duplicá-las não valia a pena.

Tabela 2.6.1 - Efeito das falhas injetadas.

Efeito das Falhas	Quantidade	PVF (%)
Masked	11096	53.1036
SDCs	5306	25.3936
Crashes	1996	9.5525
Hangs	2497	11.9503

2.7 Hardening por duplicação seletiva com menos hangs:

Após analisar a duplicação seletiva inicial, percebeu-se que a grande maioria dos *hangs* poderiam ser facilmente evitados se a comparação das variáveis de *score* com suas duplicações fosse feita conforme elas são utilizadas, em vez de apenas compará-las no final. Em caso de diferença, bastaria parar o algoritmo detectando SDC. Isso traz o ponto positivo de não deixar o usuário da aplicação em dúvida se o algoritmo está executando normalmente e irá dar a resposta um dia ou se ele irá ficar parado pra sempre.

Além dessa proteção diferente das variáveis de *score*, optou-se por também proteger a variável j, a fim de notar quão melhor ficaria a detecção de SDCs. A variável i, por sua vez, teve sua proteção um pouco modificada. Em vez de apenas acionar uma *flag* quando algum SDC afeta tal variável e testar essa *flag* ao fim da execução do algoritmo, quando algum SDC afeta i o algoritmo é imediatamente encerrado detectando esse SDC; a proteção de j é feita da mesma maneira.

Isso acabou por aumentar bastante a porcentagem de *maskeds* em relação à duplicação seletiva inicial, já que a *flag* também era protegida e cerca de 36% dos SDCs ocorridos e detectados se deviam a ela.

Por fim, das 21674 falhas injetadas na implementação que utiliza essa técnica de *hardening*, 6187 geraram SDCs, dos quais 99.45% foram detectados. Quanto aos *hangs*, houve uma gigantesca melhora: ocorreram apenas 48.

Tabela 2.7.1 - Efeito das falhas injetadas.

Efeito das Falhas	Quantidade	PVF (%)
Masked	13587	62.6880
SDCs	6187	28.5457
Crashes	1852	8.5448
Hangs	48	0.2215

2.8 Conclusão:

Cada uma das implementações foi rodada cinco vezes com mesmos dados de entrada a fim de se medir o tempo médio de execução de cada uma. Notar que, apesar de as implementações posteriores à original não incluírem *max_score* na resposta final, elas ainda o calculam, mantendo a comparação do tempo de execução justa. Os resultados estão na tabela abaixo.

Tabela 2.8.1 - Tempo de execução das diferentes implementações.

Efeito das Falhas	Tempo de execução (segundos)
Original	4.0776
Duplicação total	8.1790
Duplicação seletiva inicial	4.0978
Duplicação seletiva anti-hang	5.3834

A duplicação total possui o ponto positivo de detectar 100% dos SDCs, mas com o custo adicional de executar o algoritmo duas vezes. Isso faz com que o tempo de execução seja mais de duas vezes maior que o algoritmo original, já que ela também compara a resposta final das duas execuções. Apesar disso, não há grandes custos adicionais à alocação de

memória, uma vez que, entre uma execução e outra, a memória é liberada, mantendo-se apenas a resposta de cada uma das execuções.

A duplicação seletiva inicial demonstrou possuir *overhead* praticamente inexistente: o tempo adicional provêm apenas das checagens para determinar se i sofreu algum SDC, uma vez que ele faz parte de alguns *loops* externos. As demais variáveis protegidas provavelmente não causam aumento de tempo o suficiente para ser notado em apenas quatro casas decimais, já que são simples comparações ao fim da execução do algoritmo.

A duplicação seletiva com a finalidade de diminuir *hangs*, por sua vez, possui *overhead* de tempo considerável, levando cerca de 1.32 vezes mais tempo que a implementação original, mas ainda assim é bem menor que o *overhead* de tempo da duplicação total. Apesar desse *overhead*, tal implementação tem o grande ponto positivo de prevenir praticamente todos os *hangs*. A duplicação seletiva anti-hang também protege a variável j: com isso, a porcentagem de SDCs detectados passou de 99.13% na duplicação seletiva inicial para 99.45%. Essa diferença, entretanto, provavelmente não vale a pena, já que é necessário realizar comparações dentro de *loops* aninhados para proteger j. Existem dois desses *loops* no algoritmo, e cada um deles pode executar até 225 milhões de vezes as instruções mais internas, incluindo as que protegem j.

Portanto, considera-se que a duplicação total não tem vez na proteção do algoritmo de Smith-Waterman: ela possui *overhead* imenso e possui resultados pouquíssimo melhores que as demais técnicas. Entre as duas implementações de duplicação seletiva, escolher alguma depende da aplicação final em que o algoritmo de Smith-Waterman será utilizado. Caso os usuários da aplicação sejam muito leigos quanto ao algoritmo, vale a pena evitar *hangs*, pois eles provavelmente acreditarão que o algoritmo sempre devolverá uma resposta, mesmo que demorada. Por outro lado, se os usuários têm certa experiência com o algoritmo, eles saberão quando ele está demorando muito mais que o normal e poderão matar o processo manualmente, reiniciando em seguida.

3 COMPRESSÃO DE ÁUDIO

3.1 Sobre .WAV e .MP3:

WAV é um formato de arquivo de armazenamento de áudio, co-desenvolvido pela Microsoft e IBM em 1991. Sua estrutura básica consiste em "chunks" de metadados seguidos por uma área de dados contígua onde é armazenada a representação do áudio em si. Essa representação de áudio se dá em *Linear Pulse Code Modulation* (LPCM), onde um sinal analógico é quantizado em um número de amostras, cada uma com um valor de amplitude. WAV aceita várias combinações diferentes de números de canais, taxa de amostra, e profundidade de amplitude (*bit depth*), mas a combinação mais comum é a utilizada pelo padrão *Redbook*, utilizado em CDs de áudio: taxa de amostragem de 44100Hz, *16-bit depth*, e dois canais para áudio estéreo.

Devido a limitações da capacidade humana de reconhecimento de áudio (20-20KHz) e de acordo com o Teorema de Nyquist, que diz que qualquer sinal analógico pode ser reconstruído sem perdas a partir de uma discretização, desde que tal discretização tenha sido feita a uma taxa de amostragem mais que duas vezes superior à frequência máxima presente/relevante no sinal original, o formato *Redbook* é reconhecido como capaz de produzir um sinal sem perdas *relevantes à audição humana*, em uma gama dinâmica de 96dB (16 bit). Nesse contexto, o formato *Redbook*, e portanto arquivos WAV utilizando-o, são considerados não-comprimidos e sem perdas (*uncompressed lossless*).

MP3 é um tipo de codificação de áudio que seletivamente descarta informações consideradas menos relevantes - geralmente de frequência superior à 17KHz - gerando um arquivo muito menor, mas que precisa ser decodificado para resultar em um sinal LPCM e é apenas uma aproximação do sinal *lossless* que (idealmente) o originou. MP3 é, portanto, uma codificação comprimida com perdas (*lossy compression*). Estruturalmente, MP3 consiste em um *chunk* de metadados seguido por *chunks* alternados dos tipos *Header* e *Data.* Um *Header* pode indicar, por exemplo, que o chunk *Data* que o segue possui taxa de amostragem de 1000Hz; isso significa que o sinal reconstruído com base nesse *chunk* descarta toda informação acima de 500Hz, mas a informação restante é armazenada em menos de 1/44 do espaço de uma LPCM *Redbook*.

3.2 O Codificador Shine:

Para medir os efeitos de falhas durante um processo de codificação WAV (Redbook) -> MP3 (128Kbps), foi utilizada uma versão antiga de um codificador MP3 chamado Shine. Desenvolvido por Gabriel Bouvigne, que participou da criação do codificador MP3 LAME, hoje considerado padrão de qualidade para o formato. Enquanto que LAME é complexo e faz uso de várias bibliotecas para implementar um modelo psicoacústico que consiga a aproximar perceptivamente um sinal original tanto quanto possível, Shine foi concebido como uma implementação mínima capaz de produzir arquivos MP3 válidos, sem preocupações quanto a performance ou qualidade-percebida-por-bitrate. Shine foi escolhido pela sua complexidade mais manejável. Utiliza as bibliotecas stdlib, stdio, string, time, e math. Pode ser encontrado em http://www.rarewares.org/rrw/shine.php

3.3 Resultados da Injeção de Falhas:

Utilizando o CAROL-FI foram injetadas 15280 falhas durante a codificação da música "God Only Knows", dos Beach Boys. O arquivo de entrada foi um WAV *Redbook*, e a saída, um MP3 de *bitrate* de 128kbps. As injeções foram feitas nos modelos Single, Double, e Random. Os efeitos das falhas foram os seguintes:

Tabela 3.3.1 - Efeito das falhas injetadas

Efeito das Falhas	Quantidade	PVF (%)
Masked	11342	74.2277
SDCs	1420	9.2931
Crashes	2258	14.7774
Hangs	260	1.7015

Tabela 3.3.2 - SDCs para cada modelo de falhas.

Modelo de falhas	SDCs
Random bit-flip	541
Single bit-flip	467
Double bit-flip	412

Tabela 3.3.3 - Crashes para cada modelo de falhas.

Modelo de falhas	Crashes
Random bit-flip	887
Single bit-flip	725
Double bit-flip	646

Tabela 3.3.4 - Hangs para cada modelo de falhas.

Modelo de falhas	Hangs
Random bit-flip	138
Single bit-flip	63
Double bit-flip	59

3.4 Análise das falhas:

As dez variáveis com maior número resultante de SDCs, representando 53% do total, são todas relacionadas ao processo de filtragem (como a variável #1, a matriz *filter*) ou a algum cálculo auxiliar usado pelo mesmo (como a variável #2, *frac_slots_per_frame*). É notável que nenhuma dessas dez variáveis é uma variável de estrutura de controle - nenhuma

é índice de laço. O primeiro índice de laço a aparecer nessa lista figura como #12; seu PVF de 33% é o maior de todos os índices de laço na lista, mas ainda é menor que a média das "top 10": 56%. Não há uma resposta simples para melhorar o índice de SDCs com poucas alterações: vários dos maiores responsáveis são arrays ou matrizes - fazendo a sua proteção consideravelmente mais cara que de outros tipos de variáveis mais simples. Como o sinal representado pela codificação MP3 é apenas uma aproximação do sinal original, é possível que a solução com melhor custo-benefício para diminuir o índice de SDCs seja uma comparação do resultado da decodificação com o sinal original, com um nível de tolerância por amostra de sinal igual ao pior erro esperado por uma aplicação sem falhas do codificador, e repetição da codificação caso esse nível seja ultrapassado.

As dez variáveis com maior número resultante de crashes representam um similar 50% do total, mas ao contrário de SDCs, temos maior representação de índices de laço: os índices i e j da função *filter_subband* aparecem como #3 e #5, o que é mais notável se desconsiderarmos o #1 e o #2 - juntos somando 19% do total de crashes - por serem variáveis relacionadas a *dynamic linking* e portanto não diretamente relacionados ao algoritmo em si. As variáveis i, j, e s (#7, também parte de *filter_subband*) juntas somam 13% do número total de crashes - um valor alto para uma função de apenas 8 linhas de código. Protegendo essa função e dispensando o uso de bibliotecas dinâmicas conseguiríamos uma redução potencial de quase um terço dos crashes.

As cinco variáveis com maior número resultante de hangs representam 85% dos hangs totais. A variável #1, *sideinfo_len*, responsável por um terço dos hangs, é apenas um int e é praticamente uma constante - portanto um alvo fácil para proteção. Outras três dessas cinco também são *int*s simples e armazenam algum valor que facilmente leva o programa a uma execução estendida e errônea. A proteção dessas quatro variáveis têm o potencial de reduzir o número de hangs em 80%.

Tabela 3.4.1 - Variáveis que mais causaram SDCs

Variável	Tipo e Tamanho	SDCs	PVF
filter	double[32][64]	138	62.72%
frac_slots_per_frame	double	105	78.35%

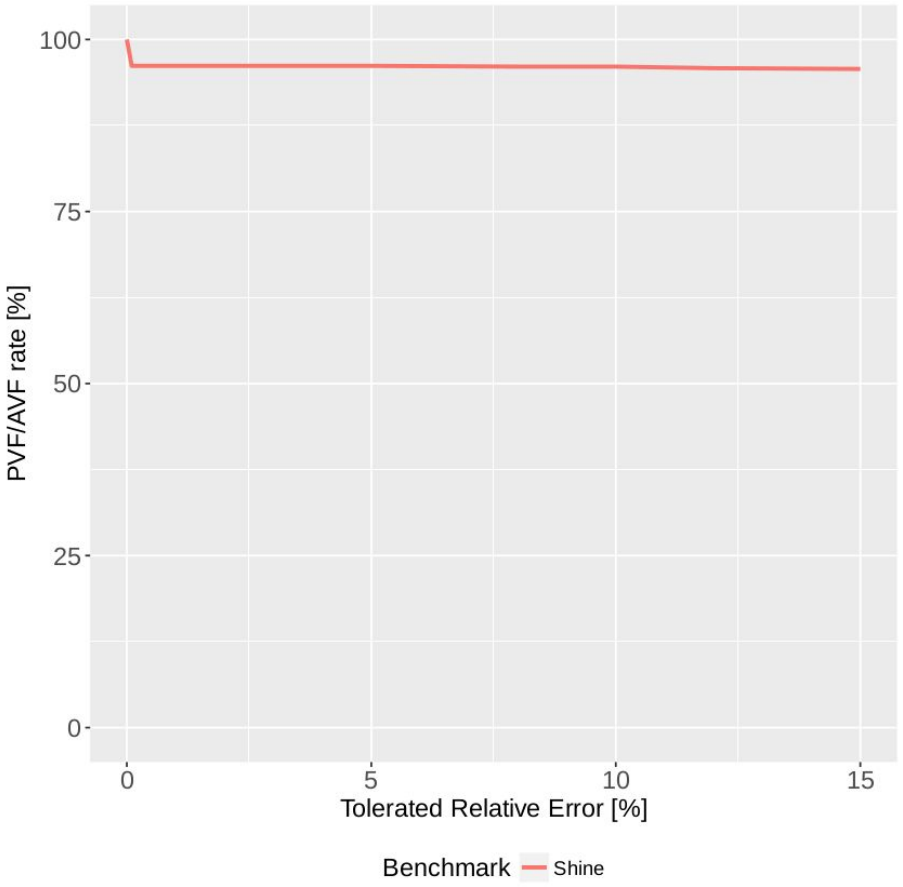
slot_lag	double	80	63.49%
x	double[2][512]	78	37.14%
off	int[2]	71	35.85%

Tabela 3.4.2 - Variáveis que mais causaram DUEs

Variável	**Tipo e Tamanho**	**DUEs**	**PVF**
user_entry	Ponteiro externo	233	99.14%
_dl_skip_args	int externo	201	100%
i	int	123	82%
off	int[2]	119	60.10%
j	int	101	67.33%

A própria natureza de um arquivo MP3, especialmente um bastante comprimido como o utilizado de exemplo - bitrate 128kbps e 2.8MB, equivalente a apenas 9% do WAV original - o torna um bom candidato a algum nível de tolerância para SDCs. Utilizando uma comparação direta byte-a-byte das saídas corrompidas *versus* a saída esperada, no entanto, obtemos o seguinte gráfico de queda de PVF efetivo em relação a níveis de tolerância:

Figura 3.4.3 - PVF vs Tolerância



Observamos que nem mesmo com uma alta tolerância não obtemos uma queda notável no PVF efetivo após o primeiro 0.1%: como o log de 11.1GB para apenas 914 arquivos corrompidos atesta, a grande maioria das saídas com SDCs tiveram grandes quantidades de corrupção. No entanto, devido a peculiaridades de representação de áudio, essa pode não ser a melhor perspectiva sobre o assunto.

Para obtermos a Figura 3.4.3, é calculada a diferença relativa entre cada posição de memória afetada. Isso não leva alguns fatores em consideração - amostras erradas isoladas no meio de passagens altas dificilmente serão percebidas, por exemplo, mesmo se elas diferem uma ordem de magnitude da amostra original, caso tal amostra original seja um valor baixo - o que acontece com frequência em sinais periódicos como ondas de pressão sonora. Além disso, essa comparação atribui valor igual para headers e para dados dos frames do MP3. Uma comparação ideal compararia os sinais finais gerados pela decodificação do MP3 e teria gamas de tolerância diferentes para amostras dependendo da sua vizinhança e de outros fatores psicoacústicos (como os que os próprios codificadores modernos usam em seus algoritmos).

Como experimento, as 1363 das saídas corrompidas das sessões iniciais de injeção que possuíam tamanho igual à saída esperada foram convertidos para WAV e comparados com o resultado da conversão da saída esperada para WAV também. A comparação registrou a quantidade de amostras com qualquer erro, assim como com erros absolutos de 0.1% (-60dB), 1% (-40dB) e 10% (-20dB), respectivamente, no arquivo wavcrit.csv. Por "erro absoluto", entenda-se como erro relativo ao teto de representação de uma amostra. Por exemplo, uma amostra lida como 67 mas cujo valor deveria ser 1 possui erro de 6600% relativo ao valor esperado, mas esse valor ainda é apenas um pouco mais que 0.1%, ou -60dB em relação ao teto.

Isso é importante pois um desvio dessa magnitude dificilmente será ouvido no meio de uma passagem musical; além disso, é possível que esse desvio não seja percebido mesmo em uma passagem silenciosa dependendo da frequência do ruído e da amplitude à qual o ouvinte é exposto. Por exemplo, se um sinal com pico em -0dB for ouvido como 80dB por um indivíduo, um ruído de -60dB seria ouvido como 20dB - valor abaixo do nível ruído de fundo de muitos tipos de ambientes. 7% dos arquivos comparados apresentaram nenhuma amostra acima de 0.1% de erro, valor que aumenta para 15% para 1% de erro, e para 36% se adicionalmente incluirmos arquivos com menos de 441 amostras com mais de 1% de erro - equivalente a 0.1s.

3.5 Hardening por duplicação total

Com o objetivo de detectar SDCs durante o tempo de execução, foi realizada a duplicação total do código em tempo. Foram injetadas com sucesso 7498 falhas distribuídas entre três modelos de falhas, com os seguintes resultados:

Tabela 3.5.1 - Resultados da duplicação total

Resultado da Falha	Ocorrências	PFV (%)
Masked	4987	66.51%
SDC detected	909	12.12%
Crash	1428	19.04%
Hang	169	2.25%
SDC undetected	5	0.55%

Tabela 3.5.2 - SDCs para cada modelo de falhas

Modelo de falhas	SDCs
Random bit-flip	353
Single bit-flip	281
Double bit-flip	280

Tabela 3.5.3 - Crashes para cada modelo de falhas

Modelo de falhas	Crashes
Random bit-flip	353
Single bit-flip	459
Double bit-flip	431

Tabela 3.5.4 - Hangs para cada modelo de falhas

Modelo de falhas	Hangs
Random bit-flip	84
Single bit-flip	45
Double bit-flip	40

A versão duplicada da aplicação detectou 99.45% dos SDCs e executou em aproximadamente 2.125x mais tempo, ou seja, resultou em um overhead de 112.5%.

3.6 Hardening por duplicação seletiva

Para obtermos uma detecção razoável de SDCs em tempo de execução com overhead significativamente menor, diversas variáveis e operações sobre as mesmas foram duplicadas,

com igualdade entre as originais e as duplicadas sendo verificadas no mínimo a cada passo do laço principal da função de compressão. As variáveis escolhidas todas pertencem a l3subbandc., l3mdct.c, ou layer3 (onde compress() reside). Segue a lista das variáveis duplicadas:

Tabela 3.6.2 - Lista das variáveis protegidas na duplicação seletiva

Nome da variável	Tipo e Tamanho	Módulo
off	int[2]	l3subbandc.c
x	double[2][512]	l3subbandc.c
filter	double[32][64]	l3subbandc.c
i (init)	int	l3subbandc.c
j (init)	int	l3subbandc.c
k	int	l3subbandc.c
y	double[64]	l3subbandc.c
i (filter)	int	l3subbandc.c
j (filter)	int	l3subbandc.c
ca	double[8]	l3mdct.c
cs	double[8]	l3mdct.c
win	double[36]	l3mdct.c
cos_l_dup	double[18][36]	l3mdct.c
i	int	layer3.c
avg_slots_per_frame	double	layer3.c
frac_slots_per_frame	double	layer3.c
whole_slots_per_frame	long	layer3.c
slot_lag	double	layer3.c
mean_bits	int	layer3.c
sideinfo_len	int	layer3.c

Foram injetadas 7500 falhas entre três modelos de injeção com os seguintes resultados:

Tabela 3.6.3 - Resultados da duplicação seletiva

Resultado da Falha	Ocorrências	PFV (%)
Masked	4996	66.61%
SDC detected	1266	12.72%
Crash	1092	14.56%
Hang	146	1.94%
SDC undetected	138	4.16%

Podemos observar 75% de cobertura da detecção de SDCs, em uma redução de mais de 50% de SDCs em relação à versão não-protegida (9.29% -> 4.16%). Isso é atingido com um tempo de execução 1.375x maior, ou 37.5% de overhead.

Tabela 3.6.7 - Resumo das diferenças de detecção de SDCs

Modelo	SDCs (não detectados, %)	Overhead
Original	9.29	1.00x
Duplicação total	0.55	2.125x
Duplicação seletiva	4.16	1.375x

4 REGRESSÃO LINEAR

4.1 O Algoritmo:

O algoritmo de regressão linear trata-se de um método automático para determinar funções com base em um conjunto de treinamento de entradas mapeadas em saídas. O método tem objetivo de determinar a saída correspondente a novas entradas que não estão presentes no conjunto de treinamento da forma mais adequada possível. Portanto o algoritmo busca prever uma função f no seguinte formato:

$$f_\theta(x_1, \ldots, x_N) = \theta_0 + \theta_1 x_1 + \ldots + \theta_N x_N$$

Onde:

- θi : são os parâmetros que deverão ser calculados pelo método
- xi : é a entrada

A função f é determinada minimizando o erro médio quadrado dos valores previstos por f com os exemplos de treinamento, portanto o algoritmo usa várias operações matemáticas para calcular a derivada parcial dessa função em relação a cada parâmetro e para realizar o ajuste dos parâmetros iterativamente até que o número de iterações estipulada pelo usuário se esgote ou até que os parâmetros da função parem de mudar.

4.2 A implementação:

O algoritmo foi implementado em C++ usando funções da biblioteca ‘cmath’ para os cálculos e usando a estrutura de dados ‘vector’ para armazenar o conjunto de treinamento e os parâmetros atuais da função. A implementação aceita um conjunto de treinamento de qualquer tamanho e depois calcula saídas com base na função que encontrou para um conjunto de qualquer tamanho de entradas.

4.3 A injeção de falhas:

Usando o CAROL-FI foram injetadas 10876 falhas divididas entre três modelos de falhas, os efeitos das falhas foram os seguintes:

Tabela 4.3.1 - Efeito das falhas injetadas.

Efeito das Falhas	Quantidade	PVF (%)
Masked	8551	78.6227
SDCs	586	5.3880
Crashes	1587	14.5918
Hangs	152	1.3976

Tabela 4.3.2 - SDCs para cada modelo de falhas.

Modelo de falhas	SDCs
Random bit-flip	238
Single bit-flip	174
Double bit-flip	174

Tabela 4.3.3 - Crashes para cada modelo de falhas.

Modelo de falhas	Crashes
Random bit-flip	561
Single bit-flip	521
Double bit-flip	505

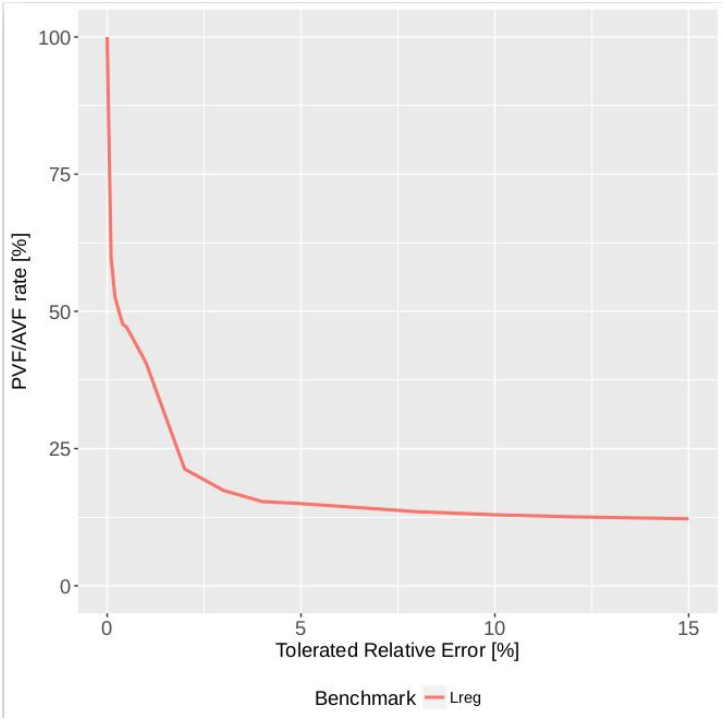
Tabela 4.3.4 - Hangs para cada modelo de falhas.

Modelo de falhas	Hangs
Random bit-flip	59
Single bit-flip	58
Double bit-flip	35

4.4 Análise das falhas:

Como dito anteriormente, o algoritmo de regressão linear busca prever o resultado de uma função desconhecida baseado num exemplo de entradas e saídas correspondentes. Para tanto, o algoritmo usa muitos cálculos em ponto flutuante para adaptar seus parâmetros que descrevem a função e para calcular o valor resultante de acordo com esses parâmetros para uma nova entrada. Como se trata de cálculos em ponto flutuante, é de se esperar que boa parte dos SDCs não sejam de fato críticos, pois podem ter sido causados somente nas primeiras iterações do cálculo dos parâmetros θ e seu efeito vai ser quase completamente removido pela convergência das iterações seguintes, ou ainda, podem ter sido causado nos bits menos significativos da mantissa do valor em ponto flutuante. Para confirmar essa ideia, se realizou o gráfico a seguir que compara o percentual de SDC que continuam considerados se existir certa tolerância de erro relativo para o resultado final.

Figura 4.4.1 - Gráfico PVF [%] por tolerância



A partir do gráfico acima, podemos notar que a existência de uma pequena tolerância como 2% já faz com que 80% dos SDCs sejam toleráveis, outra coisa que é importante notar é que com menos de 1% de tolerância metade dos SDCs já eram toleráveis, no entanto, a partir dos 4% de tolerância pouca mudança quanto a quantidade de SDCs toleráveis foi observada.

Como existe uma ampla gama de aplicações do algoritmo de regressão linear, definir um erro relativo a partir do qual os SDCs serão considerados críticos, não é uma tarefa

genérica e vai depender de qual aplicação está sendo dada ao algoritmo. Por exemplo, se uma aplicação gostaria de prever o preço do litro da gasolina baseado no histórico do preço do barril de petróleo, e no cenário econômico atual, um SDC que altere o resultado em 10% pode ser tolerado, em muitos casos. Por outro lado, se a regressão linear estiver tentando prever a temperatura de um reator nuclear considerando a quantidade de material radioativo adicionado ao reator, um erro de 10% pode causar um desastre se outras medidas de segurança não forem tomadas. Para manter a simplicidade, nesse relatório será usado 5% como tolerância para separar erros críticos de não críticos. Considerando isso, apenas 15% dos SDCs da implementação original são considerados críticos.

Seguindo na análise da injeção, a seguir foi feita uma tabela das oito variáveis mais críticas considerando todos os SDCs:

Tabela 4.4.1 - Variáveis mais críticas.

Variável	**Quantidade de SDCs**	**Uso da variável**
pos	58	Identifica o parâmetro θ para o cálculo de sua derivada parcial na função que retrata o erro atual da predição.
alpha	56	Taxa de aprendizado em cada iteração, indica o quanto o valor atual do parâmetro deve mudar dado o cálculo da nova iteração
i	51	Índice de laço que controla as iterações do cálculo dos parâmetros.
x	46	exemplos de entrada.
y	46	vetor que representa às saídas correspondentes às aos exemplos de entrada na variável ‘x’
iterations	33	valor que indica a quantidade de iterações usadas para o cálculo dos parâmetros
T	30	vetor com os parâmetros θ atualmente calculados
predictions	29	valor que indica a quantidade de valores de entrada cuja saída correspondente deve ser previstos após o cálculo dos parâmetros.

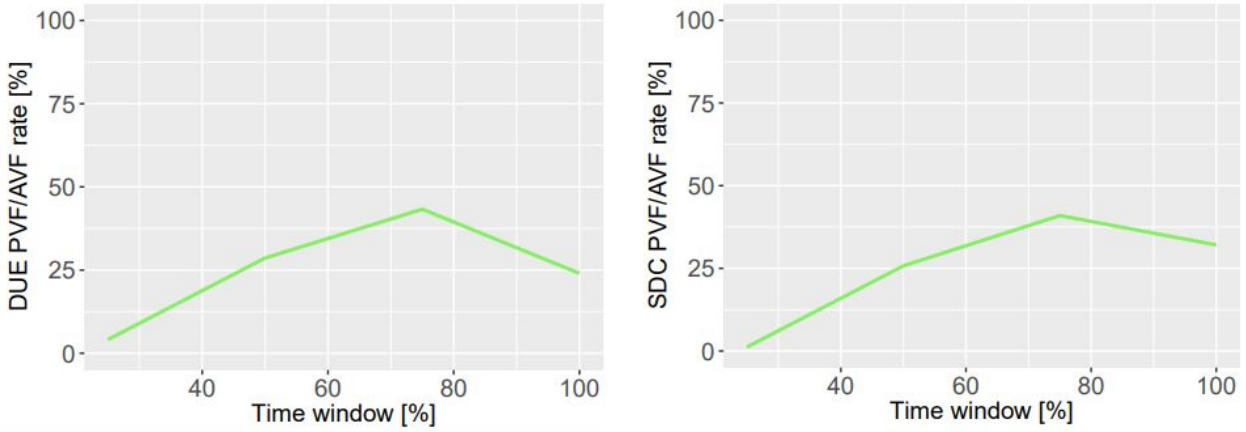
Considerando os SDCs críticos, notou-se das variáveis citadas acima, somente a “iterations”e a “i” tinham a criticidade notavelmente mais baixa que as demais. Isso ocorre

devido ao que foi suposto anteriormente, a quantidade de iterações realizadas no algoritmo não afeta tanto a saída pois o valor final dos parâmetros tende a convergir.

Quanto aos crashes, notou-se que as variáveis índice de laço correspondiam a uma quantidade bastante grande de crashes. O motivo disso é que essas variáveis de controle de laço são geralmente usadas para o acesso a elementos de vetores. No caso de seu valor ser alterado para fora dos limites do vetor o programa vai acessar um endereço que não corresponde a si na memória causando um crash. Além dessas variáveis de controle de laço, as variáveis “x”, “y” e “T” também se destacaram quanto a quantidade de crashes.

A implementação original foi analisada também quanto a sua análise em tempo sobre a janela de tempo em que as falhas injetadas causam SDCs ou crashes. Nesse aspecto, obteve-se os seguintes gráficos quase idênticos:

Gráfico 4.4.2 - Criticidade em tempo para crashes e para SDCs



O gráfico da esquerda mostra a porcentagem dos crashes (DUE) gerados por falhas injetadas em cada uma das janelas de tempo de execução do algoritmo, o gráfico da direita é o equivalente para SDCs. Neles podemos ver que a quantidade de SDCs ou de crashes gerados foi aumentando quanto mais tarde a falha era injetada no algoritmo até chegar no final onde ocorreu uma leve queda que deve ser motivada pelo fato dos parâmetros já estarem ajustados no final e o código começa a usá-los para fazer as previsões, nesse ponto se uma falha é injetada, seu efeito é limitado a apenas um subconjunto das previsões de valor enquanto que se a falha for injetada antes existe uma probabilidade maior dela vir a afetar tudo o que será previsto.

4.5 Hardening por duplicação total:

Com objetivo de detectar os SDCs na execução, foi aplicada a técnica de duplicação total em tempo. O resultado da injeção de falhas nesse modelo é mostrado a seguir.

Foram injetadas 10102 falhas com os seguintes resultados:

Tabela 4.5.1 - Efeito das falhas injetadas.

Efeito das Falhas	Quantidade	PVF (%)
Masked	6716	66.4819
Detected SDCs	1276	12.7005
Crashes	1931	19.1150
Hangs	172	1.7026
Undetected SDCs	7	0.0693

Tabela 4.5.2 - SDCs não detectados

Modelo de falhas	Undetected SDCs
Random bit-flip	1
Single bit-flip	5
Double bit-flip	1

Tabela 4.5.3 - Crashes para cada modelo de falhas.

Modelo de falhas	Crashes
Random bit-flip	701
Single bit-flip	628
Double bit-flip	602

Tabela 4.5.4 - Hangs para cada modelo de falhas.

Modelo de falhas	Hangs
Random bit-flip	86
Single bit-flip	45
Double bit-flip	41

Foram detectados 99.4544% dos SDCs.

Na duplicação total duas execuções independentes do algoritmo são executadas e em seguida o resultado de ambas é comparado. Como a duplicação total busca primeiramente reduzir a quantidade de SDCs não detectados, notou-se que a quantidade de crashes e de SDCs detectados no programa não foi reduzida, ao contrário aumentou em relação a versão original. No entanto apenas das 10102 falhas injetadas apenas 7 geraram SDCs que não foram detectados, contudo 5 desses SDCs foram críticos. O overhead em tempo gerado na duplicação total foi de cerca de 113%. Ou seja, a execução da duplicação total usa pouco mais do que duas vezes o tempo da execução no modelo original.

4.6 Hardening seletivo:

Em seguida implementou-se um hardening seletivo no modelo original, os resultados na injeção de falhas nesse modelo são mostrados a seguir.

Foram injetadas 10017 falhas, com os seguintes efeitos:

Tabela 4.6.1 - Efeito das falhas no modelo de hardening seletivo.

Efeito das Falhas	Quantidade	PVF (%)
Masked	6843	68.3139
Detected SDCs	1340	14.246
Crashes	1620	16.1725
Hangs	127	1.2678
Undetected SDCs	87	0.8685

Tabela 4.6.2 - SDCs não detectados

Modelo de falhas	Undetected SDCs
Random bit-flip	36
Single bit-flip	29
Double bit-flip	22

Tabela 4.6.3 - Crashes para cada modelo de falhas.

Modelo de falhas	Crashes
Random bit-flip	566
Single bit-flip	527
Double bit-flip	527

Tabela 4.6.4 - Hangs para cada modelo de falhas.

Modelo de falhas	Hangs
Random bit-flip	51
Single bit-flip	28
Double bit-flip	48

Para o hardening seletivo, foram feitas as seguintes modificações no modelo original:

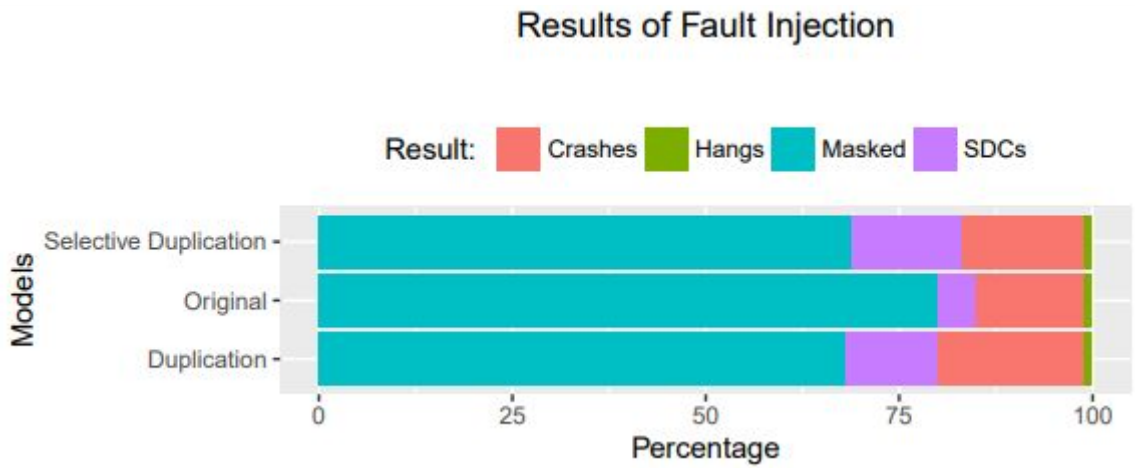
- Adiciona checksum a vetores de entrada “x” e “y” já que eles não são modificados
- Duplica variáveis de controle de laço.
- Duplica vetor com os parâmetros θ.
- Duplica valores que indicam: quantidade de iterações, taxa de aprendizado e quantidade de predições a serem realizadas

A execução do hardening seletivo apresentou um overhead em tempo de cerca de 54% e conseguiu detectar cerca de 94% dos SDCs. No entanto, cerca de 57% desses SDCs foram considerados críticos.

4.7 Comparação dos resultados:

Comparando esses três modelos para regressão linear, chegamos ao seguinte resultado, onde os SDCs detectados e não detectados estão juntos para facilitar a visualização:

Figura 4.7.1 - Comparação dos resultados para as três versões da aplicação



Neles podemos ver que a versão original foi a mais naturalmente resistente a falhas, no entanto todos seus SDCs são não detectados, portanto refletem em resultados errados na saída, enquanto que nas outras a grande maioria dos SDCs é detectado, o que possibilita a reexecução do código para obter os resultados corretos. É interessante notar que as técnicas de duplicação utilizadas tinham objetivo de reduzir a quantidade de SDCs não detectados, se formos comparar a quantidade de crashes que ocorreram, as versões que foram duplicadas superaram a versão original.

A diferença na quantidade de SDCs considerando detectados e não detectados, provavelmente ocorre devido a existência de falsos positivos que seriam mascarados posteriormente no códigos, no entanto as técnicas de Hardening detectaram que o valor estava errado e possivelmente causaria um SDC e marcaram a injeção como um SDC detectado, no geral, isso acaba não sendo um problema grande se comparado com a possibilidade de um SDC não detectado. Mesmo assim, existe certo overhead que não é detectado apenas pela comparação do tempo de execução, mas que afetaria uma execução real que é essa reexecução do algoritmo para os falsos positivos.

Comparando a quantidade de SDCs críticos e o overhead de cada modelo em relação ao modelo original obteve-se os seguintes resultados:

Tabela 4.7.1 - Comparação da quantidade de SDCs críticos

Modelo	**SDCs críticos (%)**	**Overhead**
Original	0.8091	1.00x
Duplicação total	0.0494	2.13x
Hardening seletivo	0.4992	1.54x

Analisando a tabela, notamos que o hardening seletivo não se saiu tão bem sucedido quanto pareceu uma vez que ele detectou 94% dos SDCs. O grande problema do hardening seletivo foi que ele não conseguiu detectar bem os SDCs críticos, uma vez que 57% dos SDCs não detectados foram críticos. Da tabela, notamos também que a implementação original, apenas de ser a com maior proporção de SDCs críticos, foi bastante tolerante a eles considerando que nenhuma técnica de tolerância a falhas foi aplicada nela, isso se deve ao fato de que apenas 15% dos seus SDCs eram críticos assim menos de 1% das falhas injetadas resultaram em SDCs críticos. Por sua vez, a duplicação possui uma grande diferença na proporção de SDCs críticos, quase 10 vezes menor que a proporção do hardening seletivo e cerca de 16 vezes menor que a aplicação original, no entanto seu overhead adiciona um custo bem elevado a cada execução portanto a aplicação de tais técnicas em um ambiente real vai depender da criticidade do contexto em que as aplicações se encontram.

5 COMPARAÇÃO DOS RESULTADOS

Dentre os três algoritmos analisados, o algoritmo de Smith-Waterman se mostrou o menos suscetível a SDCs, que é o defeito mais preocupante, já que aos olhos do usuário tudo ocorreu como deveria e ele possivelmente nem notará que aconteceu algo inesperado. Além disso, também é o menos suscetível a *crashes*, embora também sofra bastante com isso. Devido a isso, podemos afirmar que ele é o algoritmo mais naturalmente tolerante a falha dentre os três. Essa tolerância a falhas natural provêm do fato de o algoritmo ser *memory bound*, mas grande parte da memória utilizada não fazer diferença no resultado final da aplicação.

Todavia, Smith-Waterman é disparado o algoritmo que mais sofre com *hangs*, devido à impossibilidade de caminhamento na matriz causada por falhas específicas: cerca de 10% das falhas injetadas causam *hang*, enquanto para a regressão linear esse número é de apenas 4% e para a compressão de áudio menos de 2%.

O algoritmo de regressão linear mostrou-se como o que possui mais SDCs toleráveis. De fato, apenas 15% dos SDCs obtidos na injeção de falhas foram considerados críticos, notou-se que com pouquíssima tolerância, a quantidade de SDCs que era considerada crítica reduziu abruptamente, com isso notou-se que a natureza do algoritmo de possuir a computação dos valores em ponto flutuante aliada a existência de várias iterações que tendem a convergir para um valor determinado reduz a criticidade dos SDCs. No entanto, o mesmo efeito não esteve presente no hardening, onde se notou que uma proporção bem maior de SDCs era considerada crítica e era causada por variáveis de difícil proteção, no entanto tanto a duplicação total quanto o hardening seletivo apresentaram menos SDCs críticos do que a versão original, isso trouxe o custo de um overhead no tempo de execução que, num cenário real, o preço desse overhead precisaria ser avaliado pela criticidade do contexto de execução do algoritmo.

O algoritmo de compressão de áudio apresentou a maior taxa de SDCs. Isso pode ser explicado pela interdependência de muitas variáveis e pelo uso de variáveis grandes como as matrizes *filter*, *x,* e *y* sobre todos os frames da saída, o que torna seus PVFs altos. Esse algoritmo realiza uma grande quantidade de operações e age sobre uma área significativa de dados, mas a natureza *bitstream* do algoritmo, utilizando apenas um pequeno pedaço dos dados de cada vez, somado à natureza não-injetora das operações de filtro, resulta em um

meio-termo entre os algoritmos Smith-Waterman e de regressão linear nesse aspecto. O algoritmo respondeu bem à duplicação seletiva, ainda que não superlativamente como os outros aqui apresentados, mas a sua alta taxa de posições corrompidas e a dificuldade em estabelecer um critério satisfatório de criticidade impossibilitaram o uso dos conceitos de tolerância e criticidade para melhorar os resultados.

REFERÊNCIAS

- SILVA, B. Slides da disciplina de Inteligência Artificial - UFRGS.
- CARRER, F. Slides da disciplina de Biologia Computacional - UFRGS.
- https://en.wikipedia.org/wiki/Absolute_threshold_of_hearing