Análise de injeções - Transformada de Fourier

Augusto Boranga
Jonathan Martins
Lucas Assis
Murilo Wolfart

Introdução

Para a elaboração desse trabalho, foram injetadas entre 3000 e 5000 falhas do tipo *Single Bit Flip* através do CAROL-FI em 5 aplicações diferentes. Todos os testes foram executados em processadores da família Intel Core, portanto assume-se que se trata da mesma arquitetura, apesar das pequenas diferenças que possam existir entre os diferentes modelos utilizados.

Foram utilizados o gcc 5.4 (implementação em C), gcc 6.3 (implementações em C++) e Python 3.5 para realização dos testes.

As aplicações foram:

- Algoritmo de Cooley-Tukey em C++
- Algoritmo de Cooley-Tukey em C
- Algoritmo de Cooley-Tukey em Python
- Cálculo da transformada através da biblioteca FFTW (implementada em C) utilizando 1 thread
- Cálculo da transformada através da biblioteca FFTW (implementada em C) utilizando 2 threads

Todas as implementações, dados e análises realizados estão disponíveis **aqui**.

Vulnerabilidade

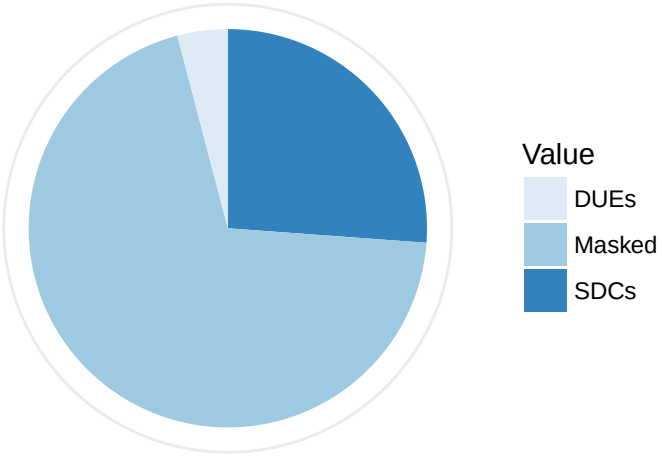
O resultado geral das injeções é mostrado na tabela abaixo.

Implementation	Injections	SDCs	DUEs	PVF.SDCs	PVF.DUEs
Cooley-Tukey C	3191	830	183	0.26	0.06
Cooley-Tukey CPP	3723	974	152	0.26	0.04
FFTW CPP	4907	1239	32	0.25	0.01
FFTW 2T CPP	3723	803	333	0.22	0.09
Cooley-Tukey Python	457	437	20	0.96	0.04

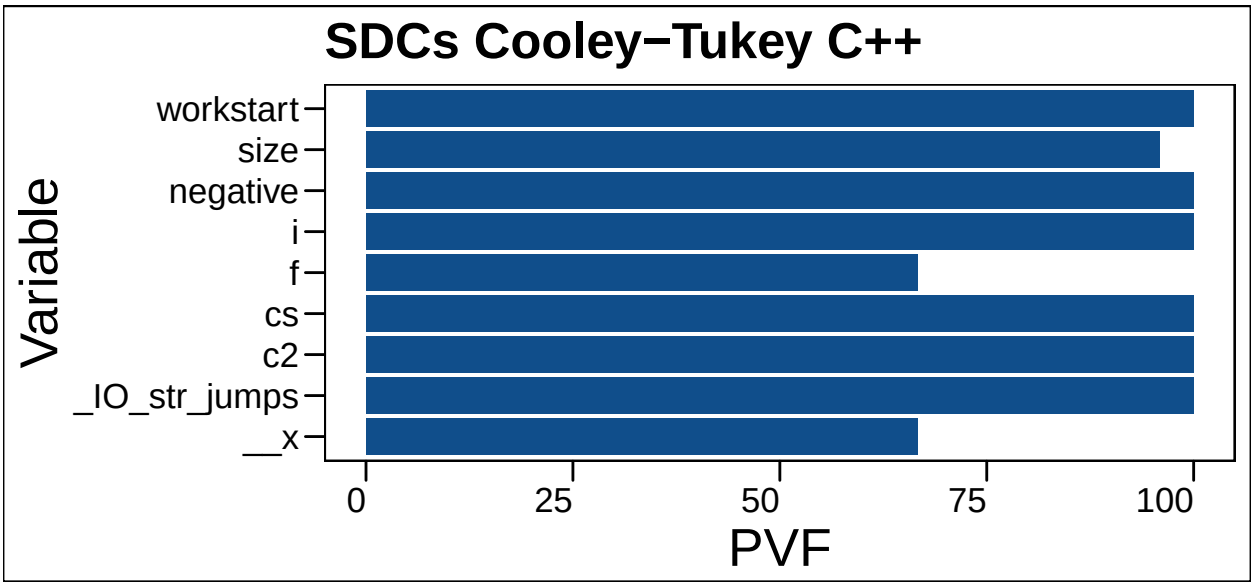
A implementação que se mostrou mais vulnerável foi a em Python que, devido aos resultados obtidos, teve muito menos injeções e foi descartada posteriormente. Além disso, as implementações que fazem uso de uma biblioteca para o cálculo da transformada se mostraram mais resilientes, possivelmente graças à verificações feitas internamente, com exceção da implementação com 2 threads que apresentou uma taxa maior de DUEs. Nas sessões seguintes faremos análises específicas para cada versão.

Análise Cooley-Tukey C++

Na implementação do algoritmo Cooley-Tukey em C++, foram injetadas 3723 falhas. No gráfico a seguir, é possível ver a distribuição dos resultados das injeções.

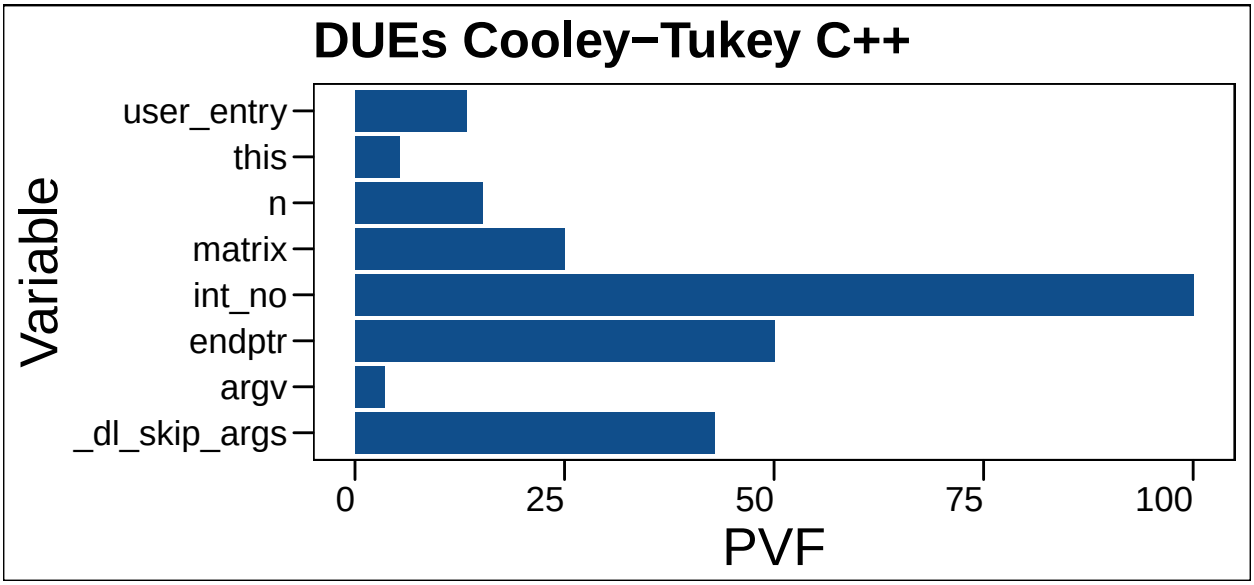


De acordo com os dados obtidos, cerca de 25% das falhas resultaram em SDCs, enquanto os DUEs representam menos de 5%. Felizmente, a grande maioria das injeções não resultou em alterações na saída.



Verificando as 10 variáveis mais vulneráveis, percebe-se que apenas duas entre elas fazem parte da aplicação em si, sendo as outras vindas das bibliotecas utilizadas. Com isso, concluimos que, para que a aplicação seja protegida da melhor maneira possível, é preferível que não sejam utilizadas bibliotecas (ou sejam utilizadas o mínimo possível). No caso deste trabalho, o foco ficará nas variáveis que podem ser protegidas, ou seja, as criadas pelos alunos.

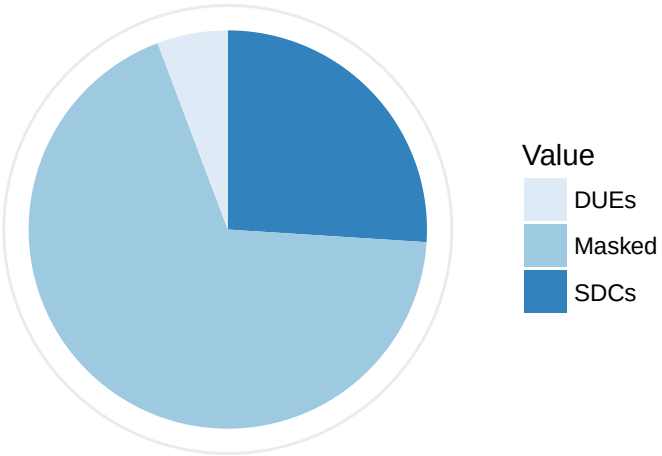
Nesta análise, temos as váriaveis *i* e *size* como mais vulneráveis em questão de SDCs. Como tratam-se de variáveis de controle, uma simples duplicação (ou triplicação) não deve resultar em um *overhead* significativo e, portanto, provavelmente será uma boa solução na etapa seguinte do trabalho.



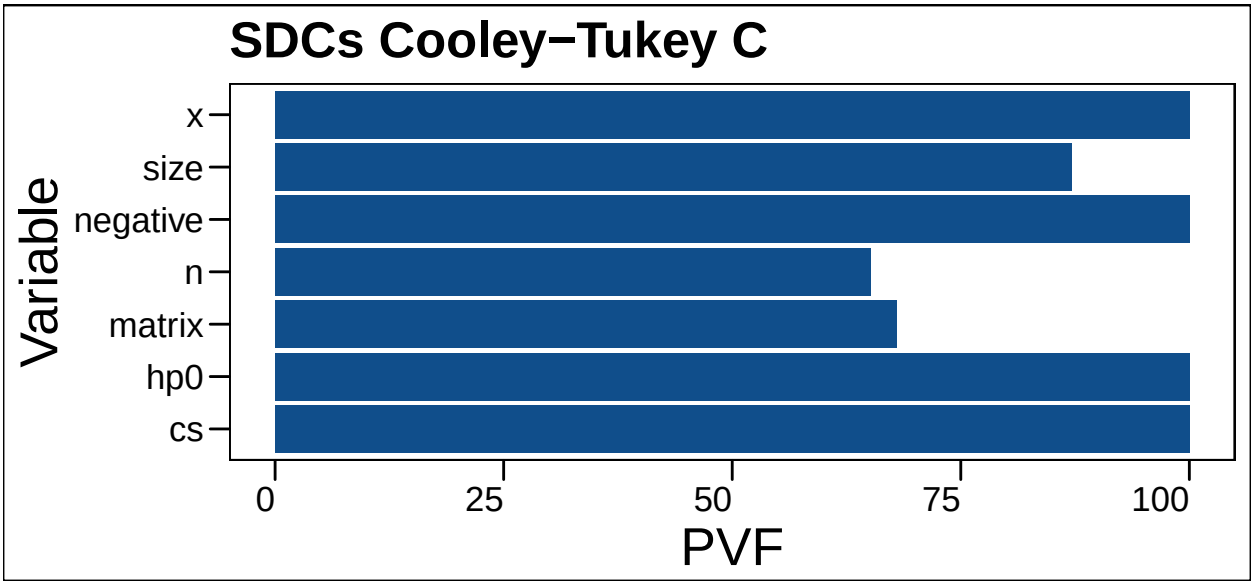
O mesmo se repete se tratando de DUEs; as variáveis que mais os causam são as externas. No entanto, também temos a variável *n* que, assim como as abordadas anteriormente, é uma variável de controle facilmente duplicável, e a variável *matrix*, responsável pela estrutura da aplicação. Para proteger esta, serão necessárias técnicas mais sofisticadas que uma simples duplicação, possivelmente explorando as redundâncias inerentes à transformada.

Análise Cooley-Tukey C

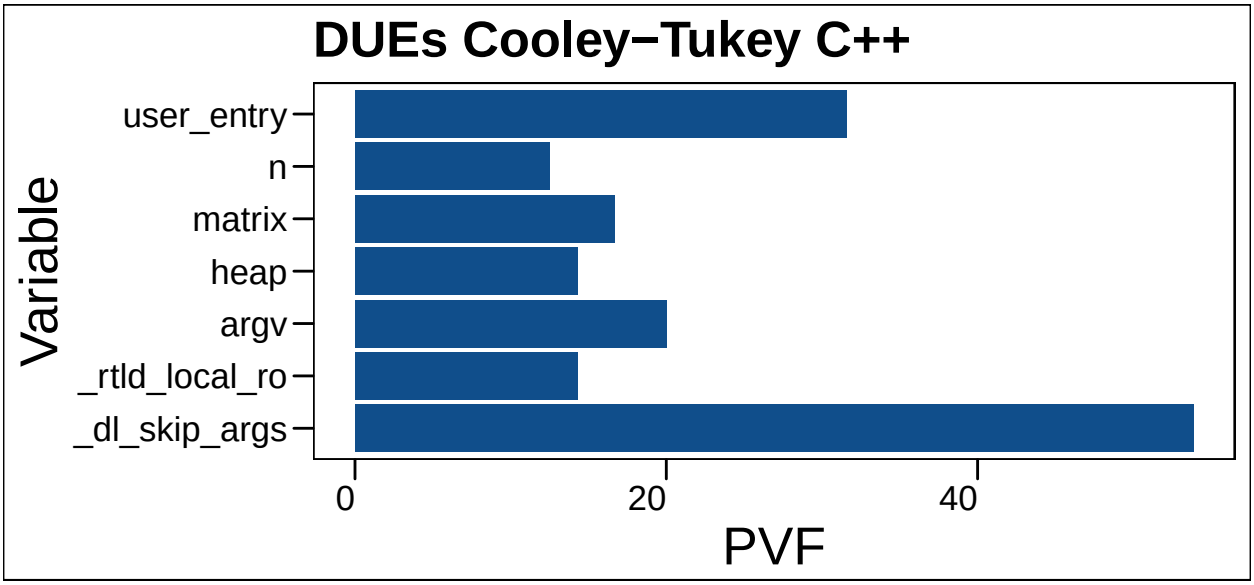
A implementação em C é extremamente similar à em C++, tendo como diferença o compilador e algumas funções de biblioteca, como o *malloc* que substitui o construtor *new* de C++.



Conforme esperado, os algoritmos apresentaram resultados muito similares, mostrando que mesmo variando alguns pequenos fatores as linguagens são praticamente equivalentes.



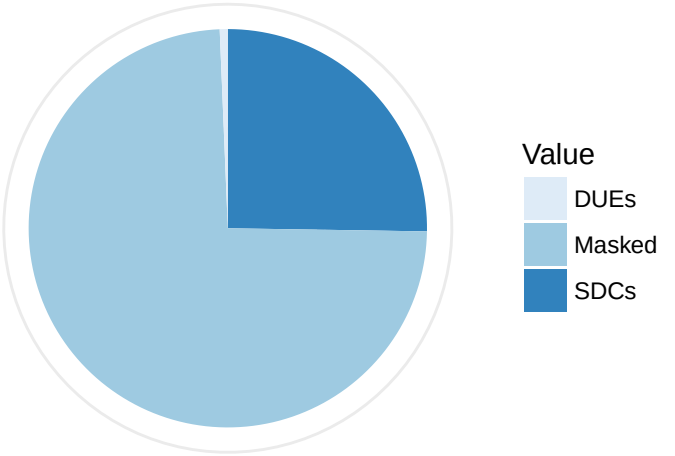
Nesse caso já são 4 as variáveis protegíveis (isto é, não pertencentes à bibliotecas), sendo 3 delas de controle e ainda a matriz, que não havia apresentado significância nos DUEs na implementação anterior. Portanto, acredita-se que essa implementação é a que será mais beneficiada através das técnicas de tolerância a falhas.



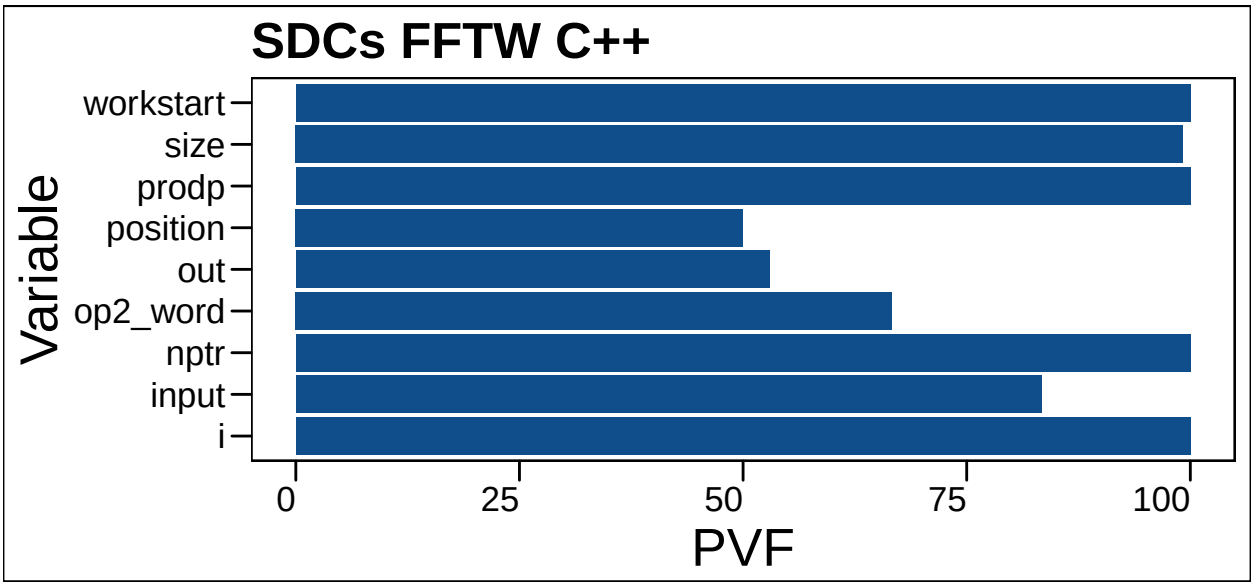
Em questão de DUEs, das três variáveis protegíveis, duas já haviam sido detectadas como vulneráveis na análise dos SDCs. A única novidade é a vulnerabilidade do *heap*, que não parece apresentar uma vulnerabilidade preocupante levando em conta o *overhead* de uma duplicação, por exemplo.

Análise FFTW C++

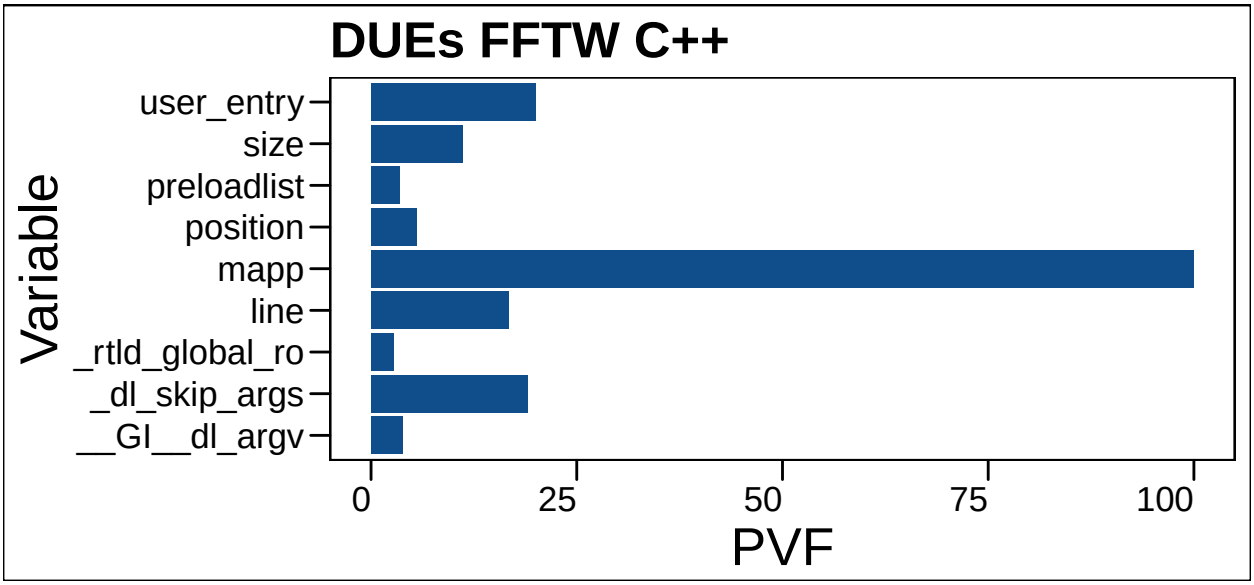
Em seguida, foram realizados testes utilizando uma biblioteca que calcula a transformada de Fourier.



Nessa implementação, novamente temos cerca de 25% de SDCs, mas os DUEs, em compensação, são ainda menos frequentes. Isso se deve provavelmente às medidas de asserção tomadas internamente na biblioteca, que foram capazes de impedir que o sistema entrasse em um estado irrecuperável.



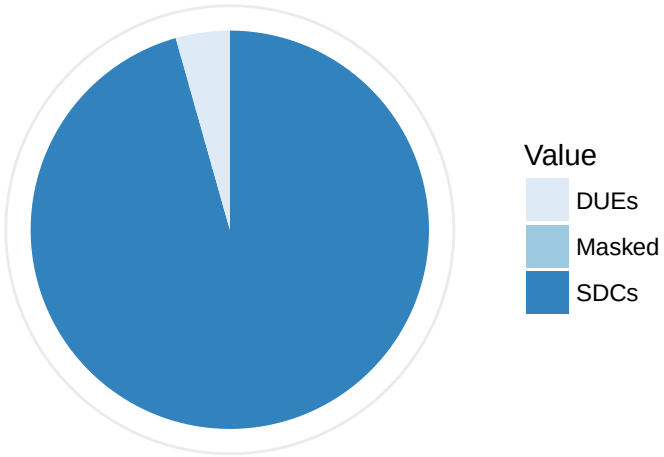
Assim como na implementação anterior, muitos dos SDCs são causados por variáveis fora do nosso controle. Porém, diferentemente da anterior, encontramos a matriz original (na variável *in*) e o arquivo de saída (na variável *output*) entre as variáveis mais vulneráveis. Provavelmente as verificações que a biblioteca implementa são capazes de barrar parte dos erros, conforme mencionado anteriormente; portanto, com maior resiliência entre as variáveis da biblioteca, valores que são menos sensíveis passam a aparecer devido à sua falta de proteção.



Já no caso de DUEs, as únicas variáveis implementadas pelos alunos que apresentaram algum risco foram variáveis de controle. No entanto, o ponto fraco claramente é a variável *mapp*, utilizada apenas pela biblioteca e, portanto, fora do alcance de nossa proteção.

Análise Cooley-Tukey Python

O grupo optou por incluir uma linguagem interpretada para verificar o impacto dessa escolha na vulnerabilidade da aplicação. Para isso, foi feita uma implementação do algoritmo de Cooley Tukey em Python.

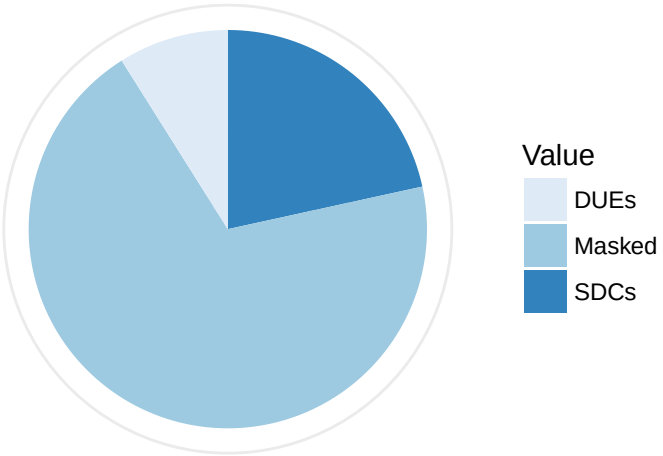


Nessa versão da aplicação, em cerca de 500 injeções não houve nenhum mascaramento. Ao verificar as variáveis que foram afetadas, percebeu-se que todas elas eram pertencentes ao interpretador Python. Com isso, surgem duas possibilidades: ou o grupo não soube utilizar o injetor, de forma que ele não foi capaz de injetar as falhas na aplicação corretamente, ou linguagens interpretadas simplesmente são extremamente vulneráveis a falhas devido à necessidade de um interpretador. Como o grupo conversou com o desenvolvedor responsável pelo CAROL-FI para elaborar os testes, pressupõe-se que trata-se realmente da vulnerabilidade de aplicações que exigem um interpretador executando suas instruções.

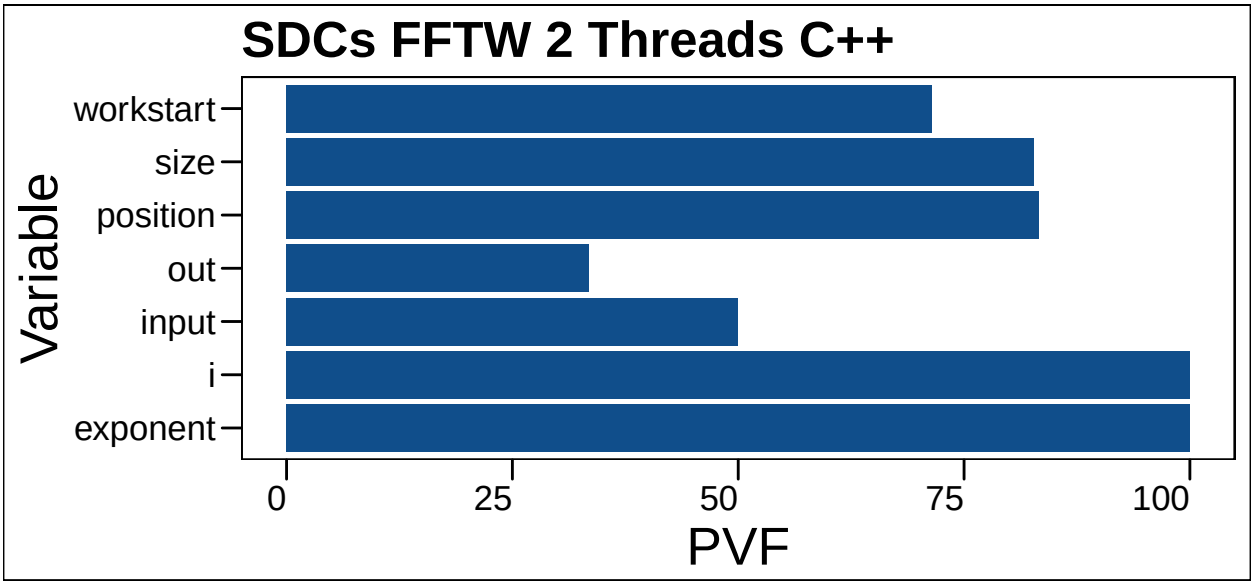
Dadas essas conclusões, não foram injetadas mais falhas nessa implementação.

Análise FFTW C++ 2 Threads

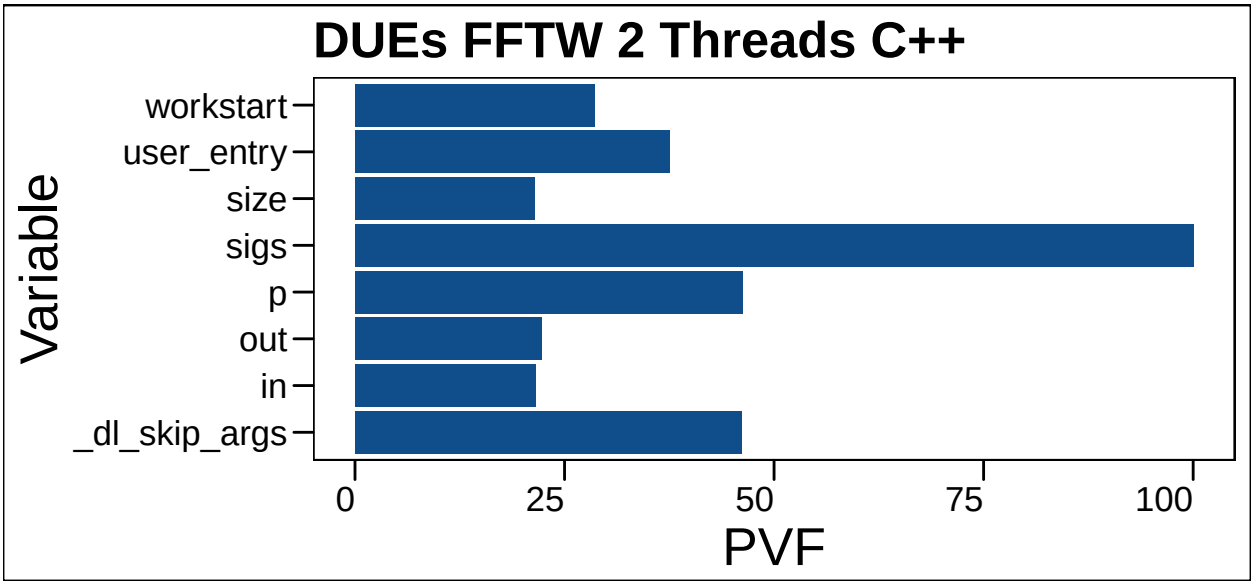
Após perceber que a análise em Python não seria capaz de fornecer os resultados apropriados, decidiu-se utilizar umas das funcionalidades da biblioteca FFTW, que permite que a transformada seja calculada em mais de uma thread.



Utilizando 2 threads, o percentual de DUEs foi o maior encontrado enquanto o de SDCs diminuiu. Isso sugere que, nessa implementação, o efeito das injeções é "mais nocivo" à execução, possivelmente devido às variáveis de controle utilizadas para utilização e sincronização das threads.



Apesar dos resultados anteriores, as variáveis mais vulneráveis nessa implementação não são muito diferentes das analisadas com uma única thread quando se trata de SDCs.



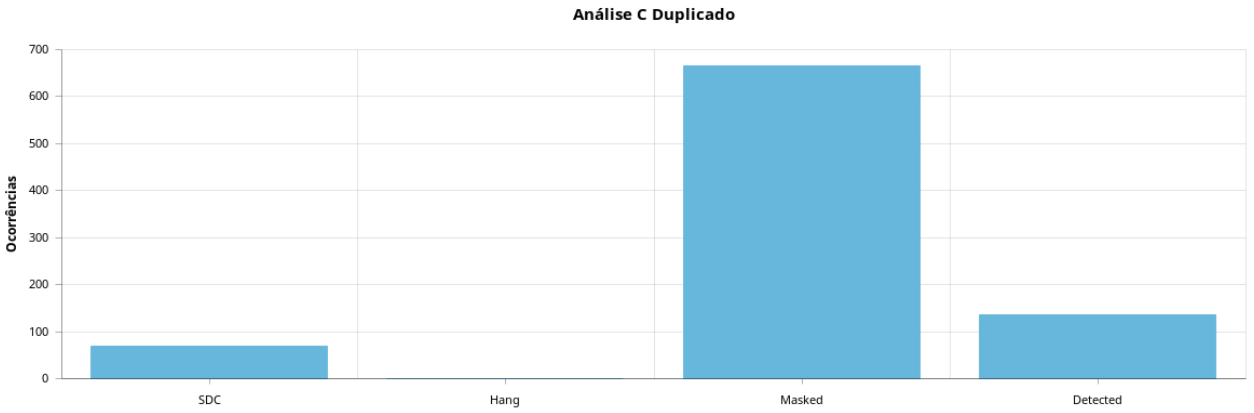
No entanto, se tratando de DUEs, surge uma nova variável extremamente vulnerável, possivelmente vinculada ao controle das diferentes threads, conforme suposto anteriormente. Além disso, surge a variável p, responsável por definir o método da transformada a ser aplicado, o que também sugere que as configurações para controle das threads estão relacionadas ao aumento dos DUEs nessa implementação.

Duplicação do Código

O código das versões em C e C++ foi duplicado. A versão em Python foi descartada devido aos resultados apresentados na avaliação das variáveis e a implementação usando a biblioteca FFTW não foi levada em consideração, já que a implementação da transformada em si não seria alterável.

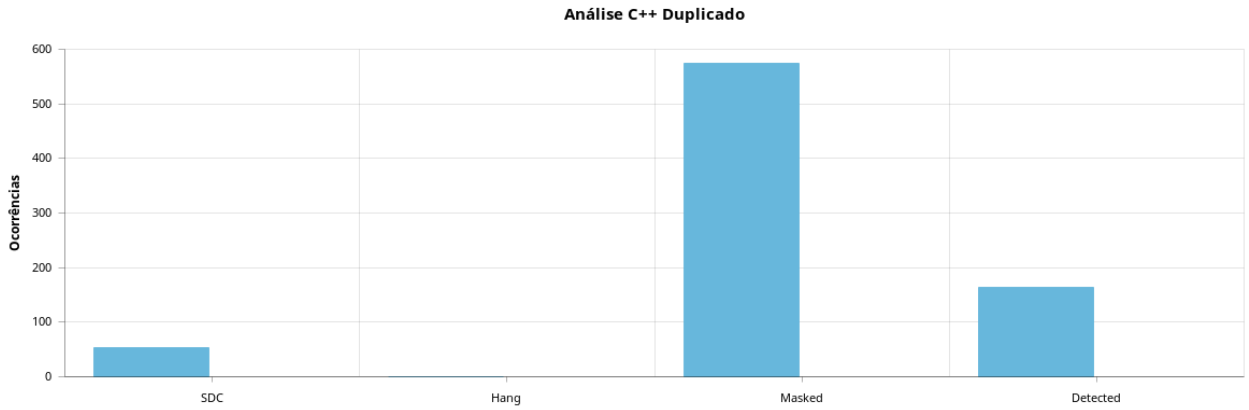
Para que fosse possível fazer tanto a verificação com o golden quanto o resultado da detecção de erros, a abordagem utilizada foi a seguinte: apenas a matriz resultante na primeira operação foi utilizada na comparação com o resultado original. Nos casos em que o algoritmo detectou uma discrepância entre a primeira e a segunda execução do algoritmo, um arquivo externo incrementou um contador de erros detectados.

Dessa maneira, foi possível realizar a contagem de erros detectados paralelamente aos resultados que o CAROL-FI fornece normalmente. No entanto, não é possível saber se os erros detectados vieram de um Hang, Masked ou SDC.



Em C, foram detectados mais erros do que houveram SDCs. Como dito anteriormente, o grupo não encontrou

uma maneira de encontrar quando houve detecção (se foi em um erro mascarado ou silencioso), mas pode-se perceber que foram detectados erros o suficiente para supormos que todos SDCs foram percebidos, apesar disso não poder ser confirmado.



Em C++, conforme esperado, o resultado foi o mesmo que o encontrado em C. Vale notar que em ambas essas análises, temos um overhead estimado de 100%, já que o código foi executado duas vezes para que fosse possível fazer essa avaliação.

Selective Hardening

O resultado da aplicação da Tranformada Discreta de Fourier é um vetor de números complexos, isto é, compostos de uma parte real e uma parte imaginária. Portanto, sabemos que os valores devem sempre contar com seus pares conjugados, ou seja, em um resultado de tamanho N, caso exista o valor a+i*b na posição k, deve haver o valor a-i*b na posição N-k.

Graças a isso, podemos detectar erros no vetor resultante em tempo linear, comparando o vetor do resultado consigo mesmo no sentido inverso. Sabendo em qual índice o valor está incorreto, é possível calcular o valor da transformada naquele ponto específico para corrigí-lo. No entanto, neste trabalho o grupo implementou apenas a detecção de inconsistências no resultado, e não sua correção.

Para verificar experimentalmente esse efeito, foram realizadas 5 execuções de cada versão em C para que fosse contabilizado o tempo. A matriz utilizada tinha dimensões 1024x1024 e, com o código duplicado, precisou de 2,9s para realizar seu processamento, contra 2,1s utilizando a verificação anteriormente descrita, ou seja, tendo uma economia de cerca de 30% no tempo de execução.

Conclusão

Após analisados os dados, concluiu-se que uma duplicação (ou triplicação) das variáveis de controle apresentaria um impacto considerável nas implementações do algoritmo Cooley-Tukey. Isso foi confirmado pelo número de erros detectados na duplicação. No entanto, infelizmente o grupo não encontrou maneiras de rastrear se os erros detectados foram oriundos de um SDC ou uma falha mascarada.

Já nas implementações que utilizam a biblioteca FFTW, a maior preocupação é com a estrutura do problema em si. Para proteger esses trechos, o grupo planejava utilizar algumas redundâncias presentes na transformada de Fourier, que permitem, no mínimo, detectar quando o resultado não é correto. Essas redundâncias, no entanto, foram implementadas somenta na versão em C, que apresentou resultados positivos conforme o esperado mas não teve a correção das falhas implementada.