EXCVATE: Spoofing Exceptions and Solving Constraints to Test Exception Handling in Numerical Libraries

Jackson Vanover[*], James Demmel[†], Xiaoye Sherry Li[‡], Cindy Rubio-González[*]

[*]University of California, Davis
[†]University of California, Berkeley
[‡]Lawrence Berkeley National Laboratory

U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under award DE-SC0020286, and the National Science Foundation under award CCF-1750983.

Numerical code should be correct

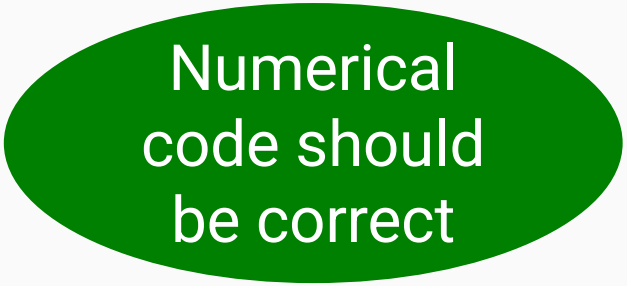
Numerical code should be correct

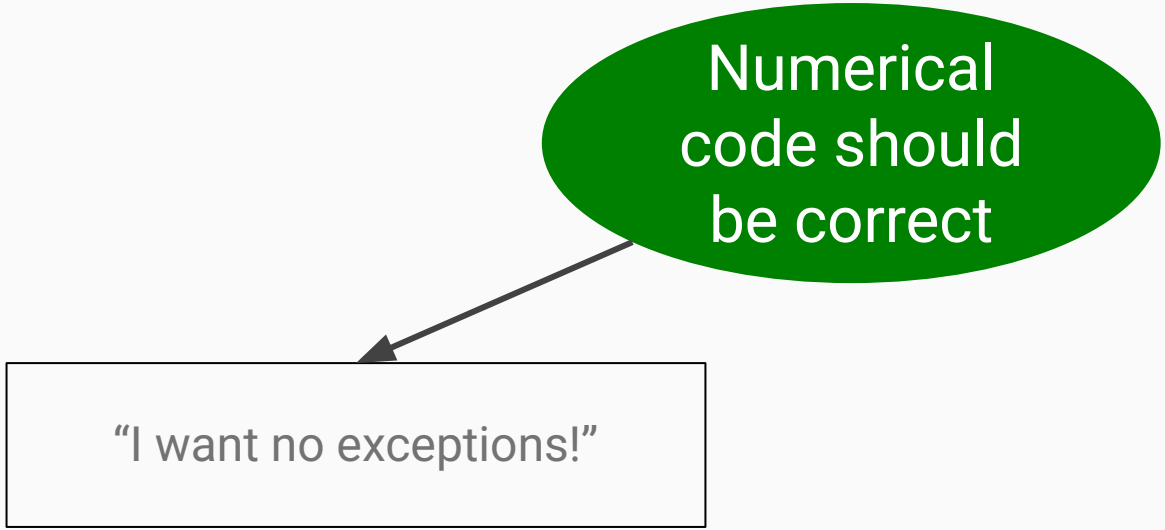
but floating-point arithmetic is inherently error-prone.

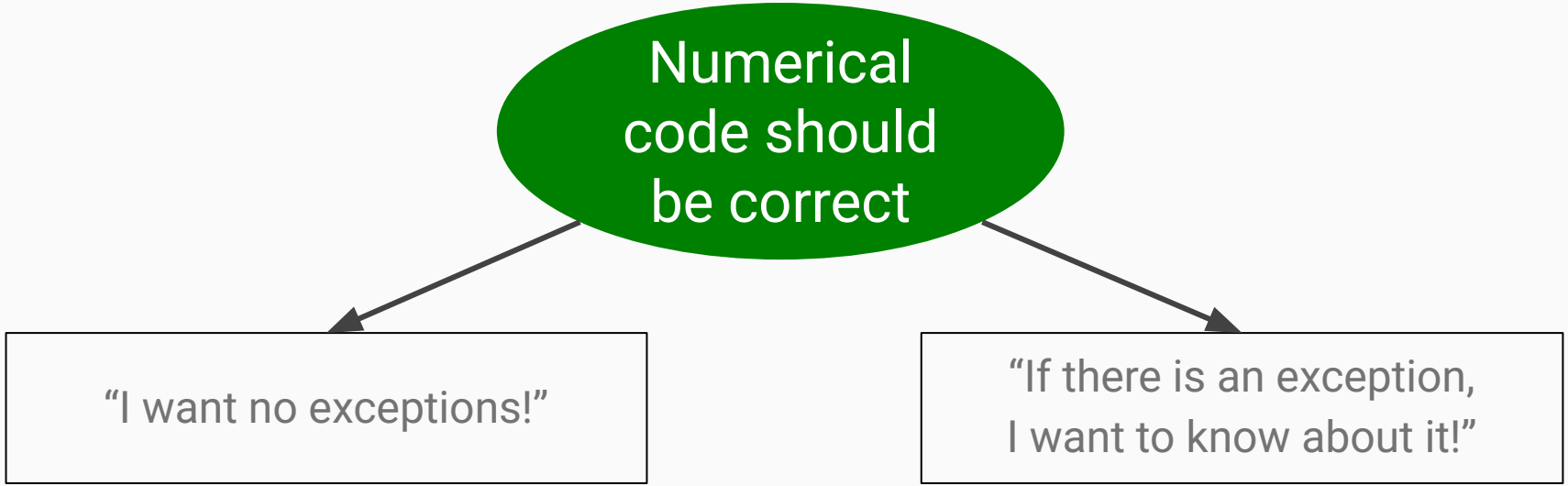
Numerical code should be correct

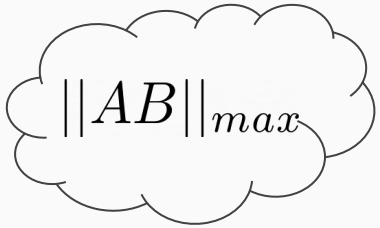
but floating-point arithmetic is inherently error-prone.

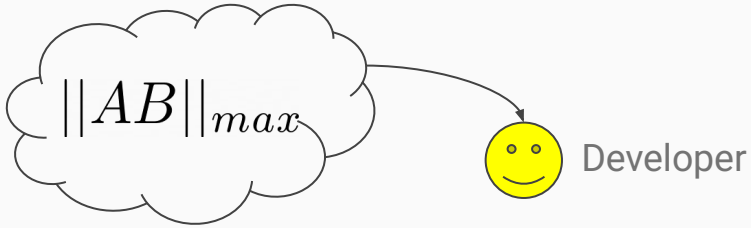
OVERFLOW
DIVIDE-BY-ZERO
INVALID
INEXACT
UNDERFLOW

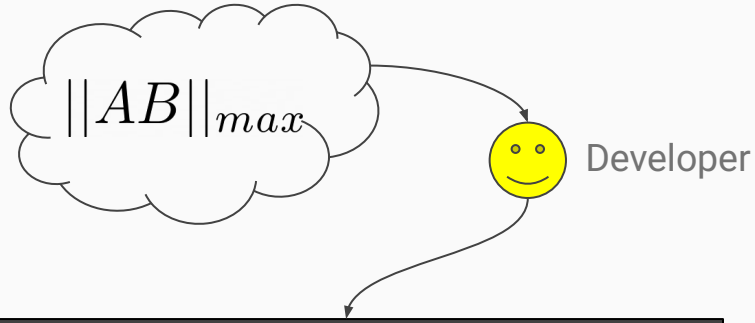

Numerical
code should
be correct


Numerical
code should
be correct
“I want no exceptions!”

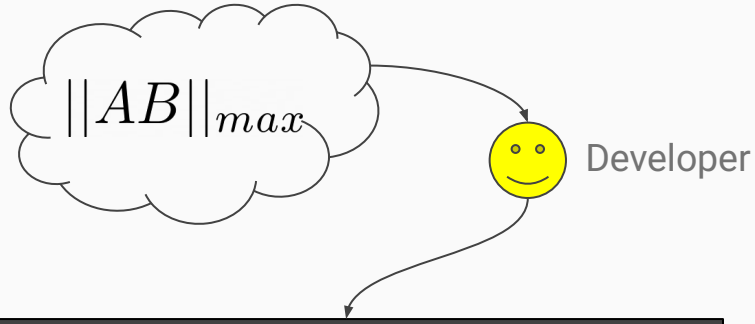

Numerical code should be correct
“I want no exceptions!”
“If there is an exception, I want to know about it!”


$||AB||_{max}$


$||AB||_{max}$
Developer



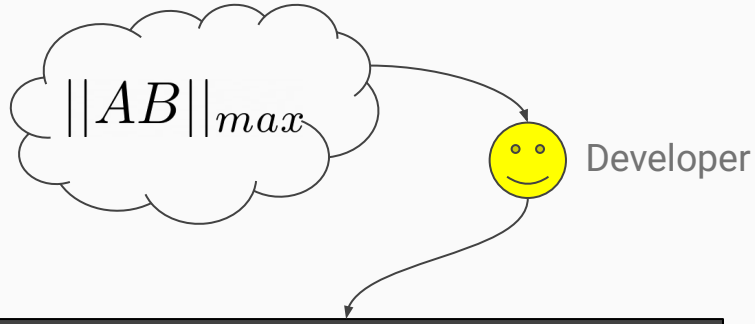

```
function mmMaxNorm(dim, A, B)
   ...
   result = 0.0
   do i = 1, dim
       do j = 1, dim
           acc = 0.0
           do k = 1, dim
               acc = acc + A(i,k) * B(k,j)
           end do
           result = max(result, abs(acc))
       end do
   end do
   ...
end function mmMaxNorm
```




```
function mmMaxNorm(dim, A, B)
   ...
   result = 0.0
   do i = 1, dim
       do j = 1, dim
           acc = 0.0
           do k = 1, dim
               acc = acc + A(i,k) * B(k,j)
           end do
           result = max(result, abs(acc))
       end do
   end do
   ...
end function mmMaxNorm
```

$$A = \begin{bmatrix} 0.0 & 0.0 \\ -3.40282 \times 10^{38} & -2.4245 \end{bmatrix}$$

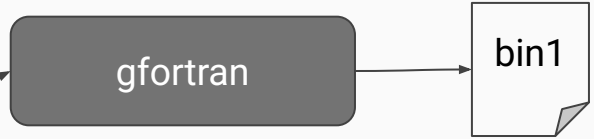
$$B = \begin{bmatrix} 0.0 & 3.06254 \times 10^{38} \\ 0.0 & -3.08694 \times 10^{38} \end{bmatrix}$$

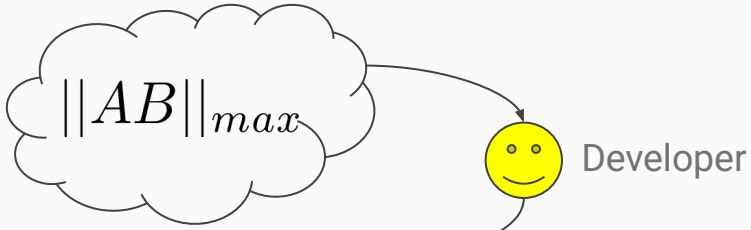


$$A = \begin{bmatrix} 0.0 & 0.0 \\ -3.40282 \times 10^{38} & -2.4245 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.0 & 3.06254 \times 10^{38} \\ 0.0 & -3.08694 \times 10^{38} \end{bmatrix}$$

```
function mmMaxNorm(dim, A, B)
    ...
    result = 0.0
    do i = 1, dim
        do j = 1, dim
            acc = 0.0
            do k = 1, dim
                acc = acc + A(i,k) * B(k,j)
            end do
            result = max(result, abs(acc))
        end do
    end do
    ...
end function mmMaxNorm
```





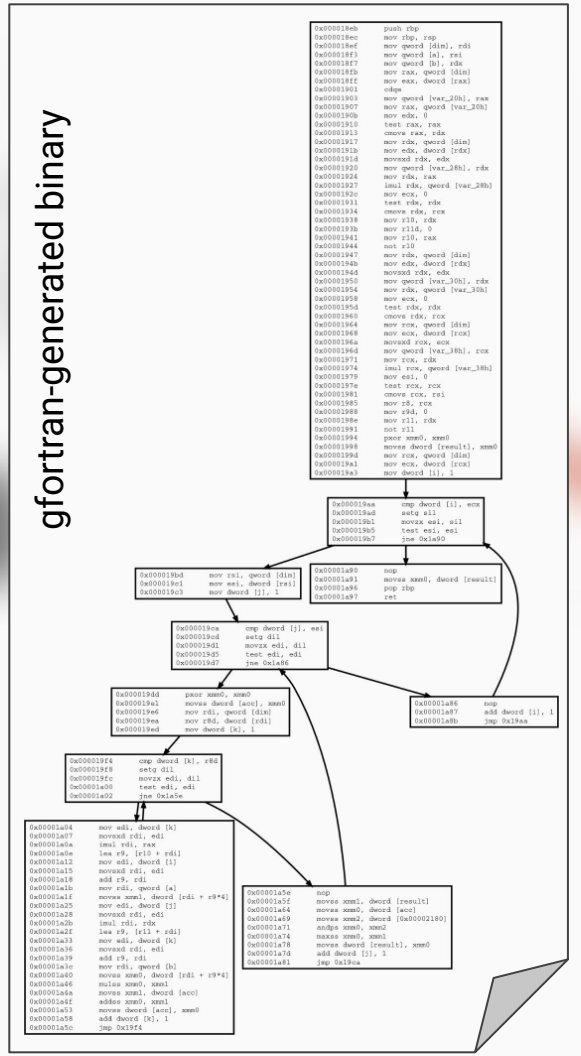

```
function mmMaxNorm(dim, A, B)
    ...
    result = 0.0
    do i = 1, dim
        do j = 1, dim
            acc = 0.0
            do k = 1, dim
                acc = acc + A(i,k) * B(k,j)
            end do
            result = max(result, abs(acc))
        end do
    end do
    ...
end function mmMaxNorm
```

$$A = \begin{bmatrix} 0.0 & 0.0 \\ -3.40282 \times 10^{38} & -2.4245 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.0 & 3.06254 \times 10^{38} \\ 0.0 & -3.08694 \times 10^{38} \end{bmatrix}$$

gfortran → bin1 → 0.0

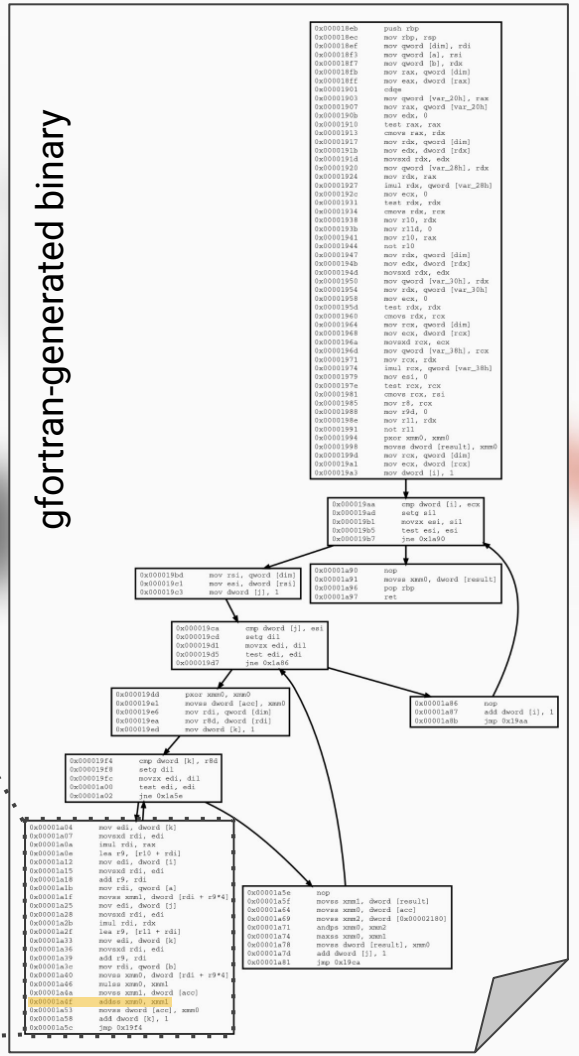
INCORRECT

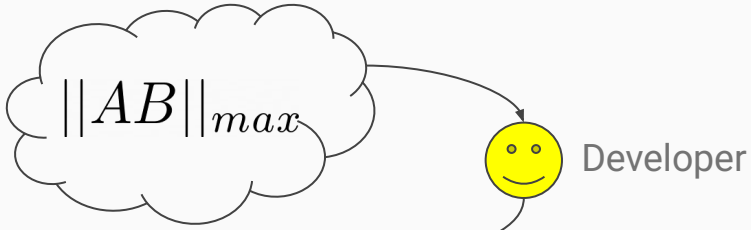

gfortran-generated binary

The eighth execution of this add instruction triggers an “Invalid” exception, generating a NaN that is ultimately lost.

```
0x00001a04      mov edi, dword [k]
0x00001a07      movsxd rdi, edi
0x00001a0a      imul rdi, rax
0x00001a0e      lea r9, [r10 + rdi]
0x00001a12      mov edi, dword [i]
0x00001a15      movsxd rdi, edi
0x00001a18      add r9, rdi
0x00001a1b      mov rdi, qword [a]
0x00001a1f      movss xmm1, dword [rdi + r9*4]
0x00001a25      mov edi, dword [j]
0x00001a28      movsxd rdi, edi
0x00001a2b      imul rdi, rdx
0x00001a2f      lea r9, [r11 + rdi]
0x00001a33      mov edi, dword [k]
0x00001a36      movsxd rdi, edi
0x00001a39      add r9, rdi
0x00001a3c      mov rdi, qword [b]
0x00001a40      movss xmm0, dword [rdi + r9*4]
0x00001a46      mulss xmm0, xmm1
0x00001a4a      movss xmm1, dword [acc]
0x00001a4f      addss xmm0, xmm1
0x00001a53      movss dword [acc], xmm0
0x00001a58      add dword [k], 1
0x00001a5c      jmp 0x19f4
```

gfortran-generated binary





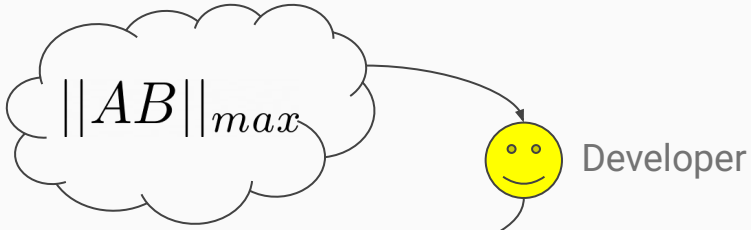

```
function mmMaxNorm(dim, A, B)
    ...
    result = 0.0
    do i = 1, dim
        do j = 1, dim
            acc = 0.0
            do k = 1, dim
                acc = acc + A(i,k) * B(k,j)
            end do
            result = max(result, abs(acc))
        end do
    end do
    ...
end function mmMaxNorm
```

$$A = \begin{bmatrix} 0.0 & 0.0 \\ -3.40282 \times 10^{38} & -2.4245 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.0 & 3.06254 \times 10^{38} \\ 0.0 & -3.08694 \times 10^{38} \end{bmatrix}$$

gfortran → bin1 → 0.0

INCORRECT




```
function mmMaxNorm(dim, A, B)
    ...
    result = 0.0
    do i = 1, dim
        do j = 1, dim
            acc = 0.0
            do k = 1, dim
                acc = acc + A(i,k) * B(k,j)
            end do
            result = max(result, abs(acc))
        end do
    end do
    ...
end function mmMaxNorm
```

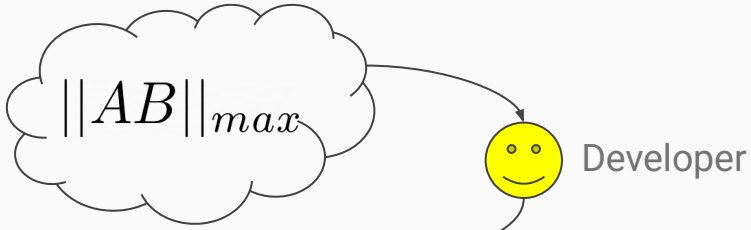
$$A = \begin{bmatrix} 0.0 & 0.0 \\ -3.40282 \times 10^{38} & -2.4245 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.0 & 3.06254 \times 10^{38} \\ 0.0 & -3.08694 \times 10^{38} \end{bmatrix}$$

gfortran → bin1 → 0.0 INCORRECT

gfortran -fdefault-real-8 → bin2

(increase FP precision to 64-bit)




```
function mmMaxNorm(dim, A, B)
    ...
    result = 0.0
    do i = 1, dim
        do j = 1, dim
            acc = 0.0
            do k = 1, dim
                acc = acc + A(i,k) * B(k,j)
            end do
            result = max(result, abs(acc))
        end do
    end do
    ...
end function mmMaxNorm
```

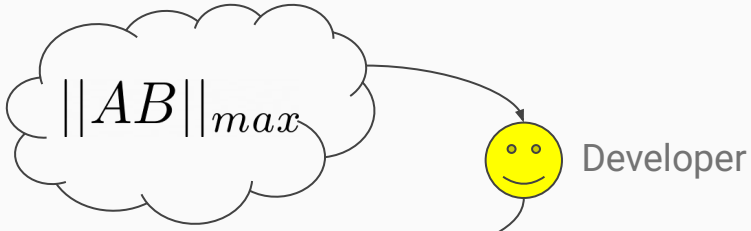
$$A = \begin{bmatrix} 0.0 & 0.0 \\ -3.40282 \times 10^{38} & -2.4245 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.0 & 3.06254 \times 10^{38} \\ 0.0 & -3.08694 \times 10^{38} \end{bmatrix}$$

gfortran → bin1 → 0.0 INCORRECT

gfortran -fdefault-real-8 → bin2 → 1.04212×10^{77}

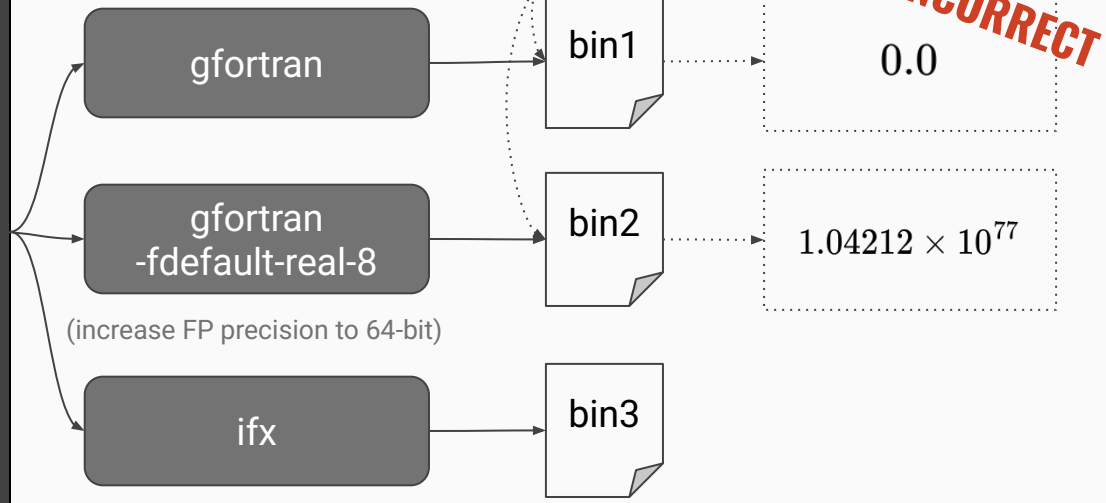
(increase FP precision to 64-bit)

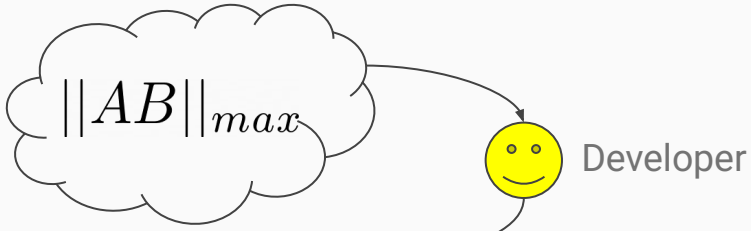



```
function mmMaxNorm(dim, A, B)
    ...
    result = 0.0
    do i = 1, dim
        do j = 1, dim
            acc = 0.0
            do k = 1, dim
                acc = acc + A(i,k) * B(k,j)
            end do
            result = max(result, abs(acc))
        end do
    end do
    ...
end function mmMaxNorm
```

$$A = \begin{bmatrix} 0.0 & 0.0 \\ -3.40282 \times 10^{38} & -2.4245 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.0 & 3.06254 \times 10^{38} \\ 0.0 & -3.08694 \times 10^{38} \end{bmatrix}$$

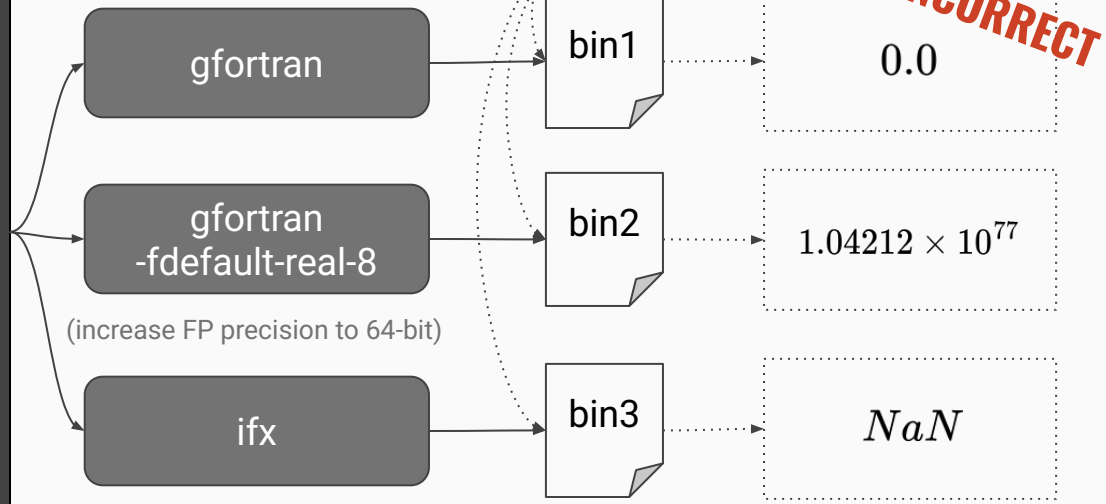





```
function mmMaxNorm(dim, A, B)
    ...
    result = 0.0
    do i = 1, dim
        do j = 1, dim
            acc = 0.0
            do k = 1, dim
                acc = acc + A(i,k) * B(k,j)
            end do
            result = max(result, abs(acc))
        end do
    end do
    ...
end function mmMaxNorm
```

$$A = \begin{bmatrix} 0.0 & 0.0 \\ -3.40282 \times 10^{38} & -2.4245 \end{bmatrix}$$

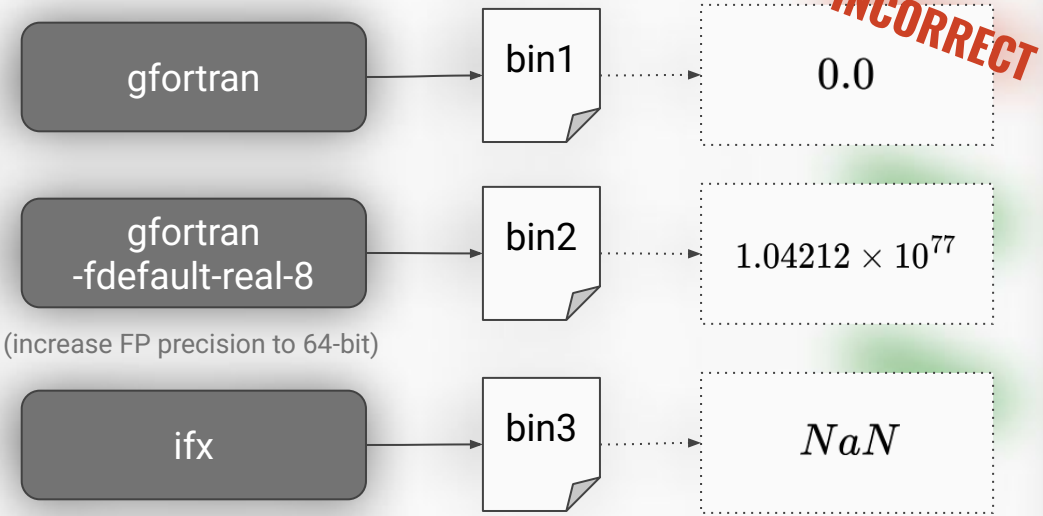
$$B = \begin{bmatrix} 0.0 & 3.06254 \times 10^{38} \\ 0.0 & -3.08694 \times 10^{38} \end{bmatrix}$$



Numerical code should be correct

"I want no exceptions!"

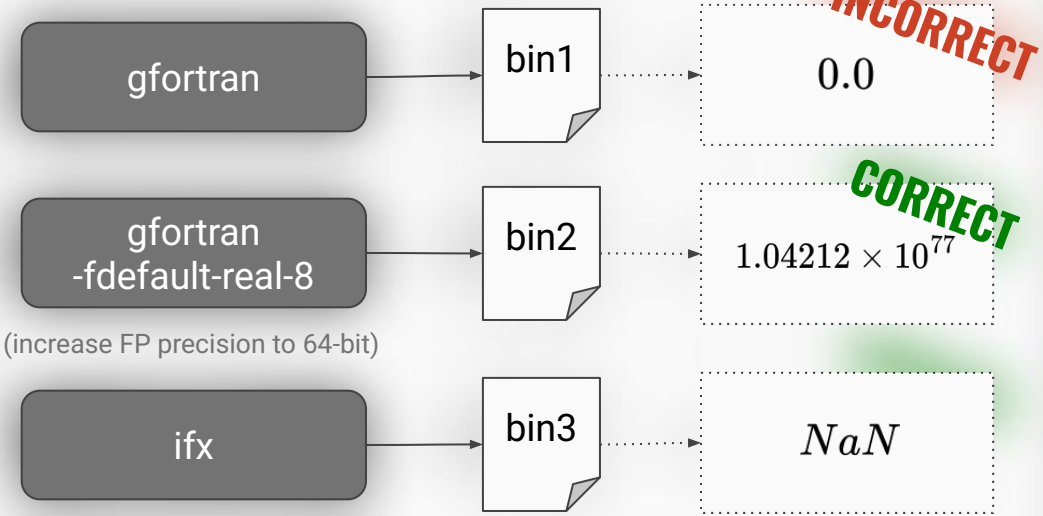
"If there is an exception, I want to know about it!"



Numerical code should be correct

"I want no exceptions!"

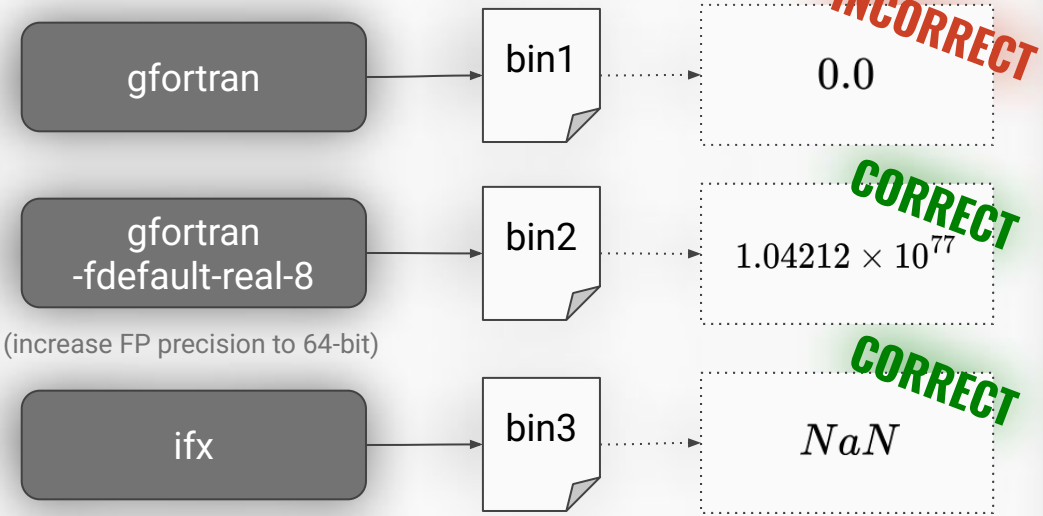
"If there is an exception, I want to know about it!"



Numerical code should be correct

"I want no exceptions!"

"If there is an exception, I want to know about it!"



Exception-Handling Specification:

When an exceptional value is input to a routine or created internally, it should ***propagate*** until it reaches the routine output or an appropriate error handler.

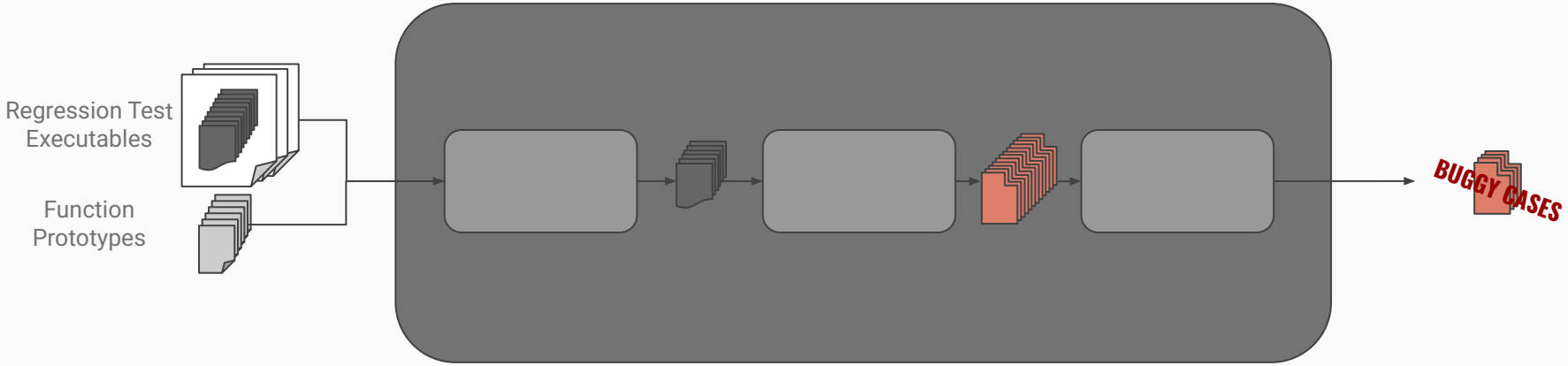
Exception-Handling Specification:

When an exceptional value is input to a routine or created internally, it should ***propagate*** until it reaches the routine output or an appropriate error handler.

This is the basis for the specification adopted by the reference implementations of LAPACK/BLAS [1]. **-> How can we test this?**

[1] J. Demmel, et al. “Proposed Consistent Exception Handling for the BLAS and LAPACK”.


Regression Test Executables
Function Prototypes
EXCVATE
Exceptional Value Tester
BUGGY CASES



EXCVATE is composed of three components that address three key challenges…

Challenge 1: Existing input generation techniques are not well-suited to testing exception handling

Challenge 1: Existing input generation techniques are not well-suited to testing exception handling

Pre-existing works use a variety of input generation techniques to yield floating-point inputs that trigger exceptions.

- e.g., fuzzing [1], symbolic execution [2,3], Bayesian optimization [4]

[1] A. Tran, I. Laguna, G. Gopalakrishnan. “FPBoxer: Efficient Input-Generation for Targeting Floating-Point Exceptions in GPU Programs”.
[2] E.T. Barr, T. Vo, V. Le, Z. Su. “Automatic Detection of Floating-Point Exceptions”.
[3] X. Wu, L. Li, J. Zhang. “Symbolic Execution with Value-Range Analysis for Floating-Point Exception Detection”.
[4] I. Laguna, G. Gopalakrishnan. “Finding Inputs that Trigger Floating-Point Exceptions in GPUs via Bayesian Optimization”.

Challenge 1: Existing input generation techniques are not well-suited to testing exception handling

Pre-existing works use a variety of input generation techniques to yield floating-point inputs that trigger exceptions.

- e.g., fuzzing [1], symbolic execution [2,3], Bayesian optimization [4]

They are designed with the “I want no exceptions” mindset; they only seek to trigger exceptions…

[1] A. Tran, I. Laguna, G. Gopalakrishnan. “FPBoxer: Efficient Input-Generation for Targeting Floating-Point Exceptions in GPU Programs”.
[2] E.T. Barr, T. Vo, V. Le, Z. Su. “Automatic Detection of Floating-Point Exceptions”.
[3] X. Wu, L. Li, J. Zhang. “Symbolic Execution with Value-Range Analysis for Floating-Point Exception Detection”.
[4] I. Laguna, G. Gopalakrishnan. “Finding Inputs that Trigger Floating-Point Exceptions in GPUs via Bayesian Optimization”.

Challenge 1: Existing input generation techniques are not well-suited to testing exception handling

Pre-existing works use a variety of input generation techniques to yield floating-point inputs that trigger exceptions.

- e.g., fuzzing [1], symbolic execution [2,3], Bayesian optimization [4]

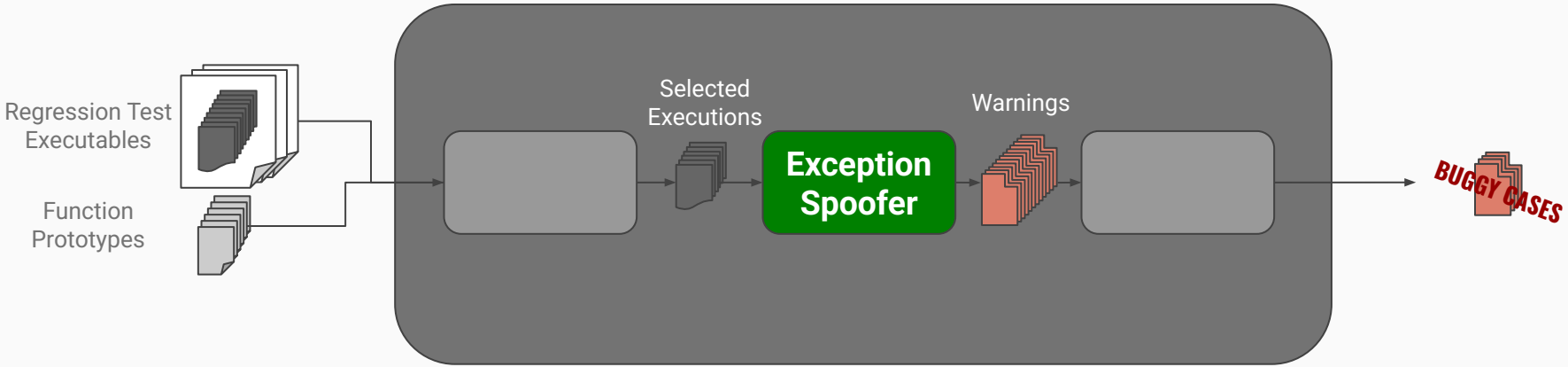
They are designed with the “I want no exceptions” mindset; they only seek to trigger exceptions… but most code should handle exceptions correctly!

- IEEE754 mandates propagation of exceptional-values in most cases
- Developers often do a good job!

[1] A. Tran, I. Laguna, G. Gopalakrishnan. “FPBoxer: Efficient Input-Generation for Targeting Floating-Point Exceptions in GPU Programs”.
[2] E.T. Barr, T. Vo, V. Le, Z. Su. “Automatic Detection of Floating-Point Exceptions”.
[3] X. Wu, L. Li, J. Zhang. “Symbolic Execution with Value-Range Analysis for Floating-Point Exception Detection”.
[4] I. Laguna, G. Gopalakrishnan. “Finding Inputs that Trigger Floating-Point Exceptions in GPUs via Bayesian Optimization”.

Challenge 1: Existing input generation techniques are not well-suited to testing exception handling

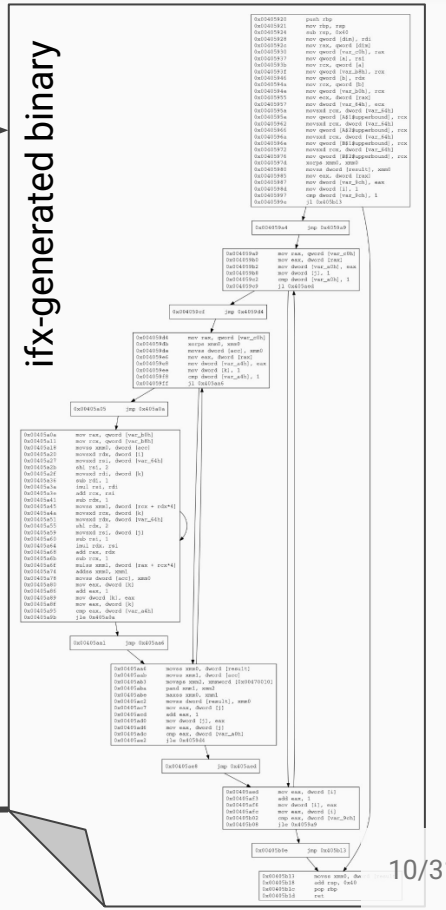
Approach: spoof exceptions via binary rewriting to cheaply find handling failures



Approach: spoof exceptions via binary rewriting to cheaply find handling failures

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$



20

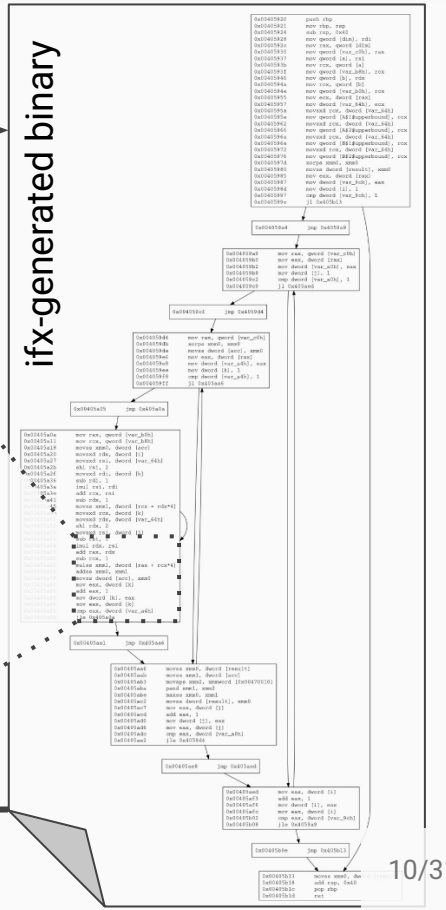
Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```



20

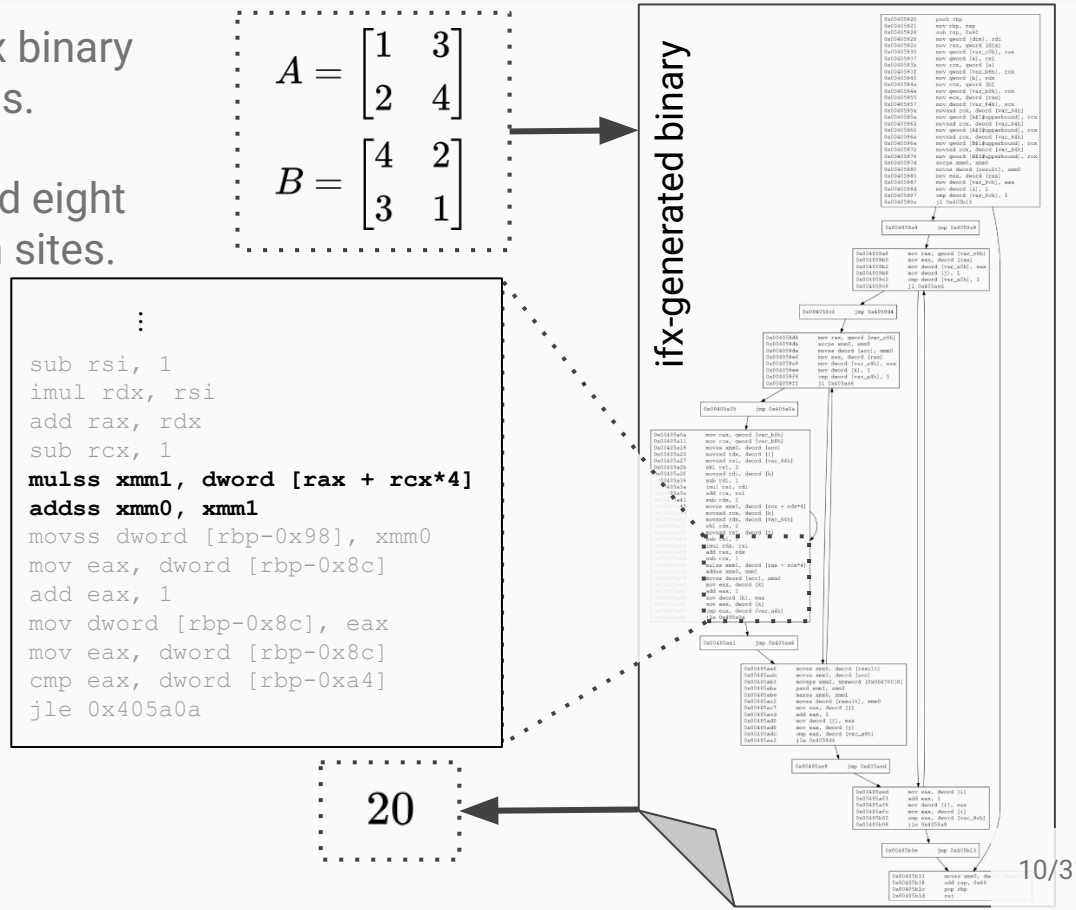
Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$



Approach: spoof exceptions via binary rewriting to cheaply find handling failures

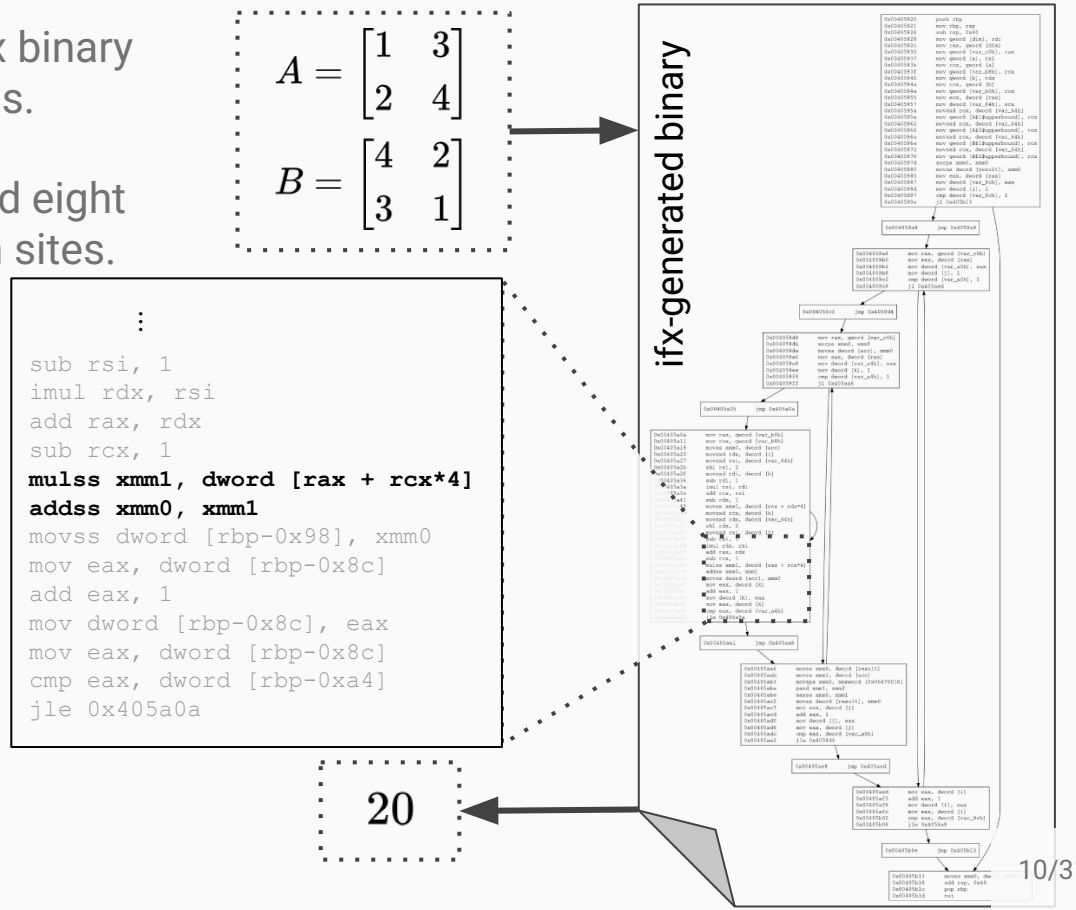
There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$



Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

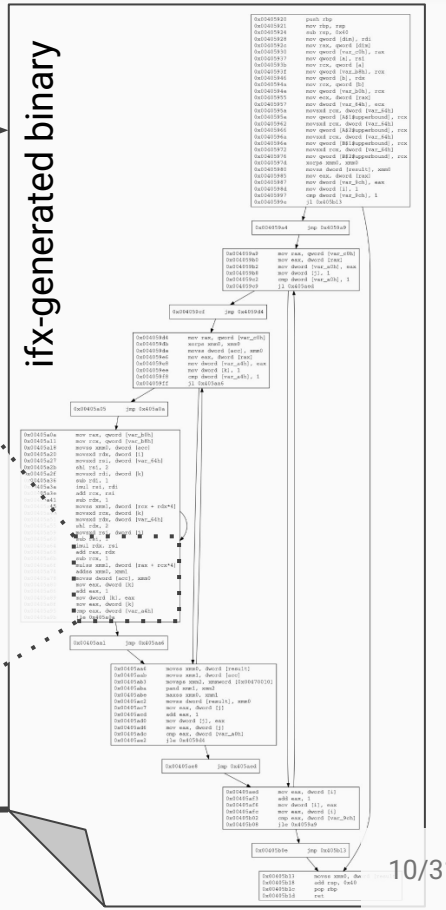
Table of Test Results

	1	2	3	4	5	6	7	8
mul								
add								

Iteration in which EXCVATE spoofs an exception

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

20

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Replay and overwrite xmm1 with NaN the ***1st*** time mulss is executed

ifx-generated binary

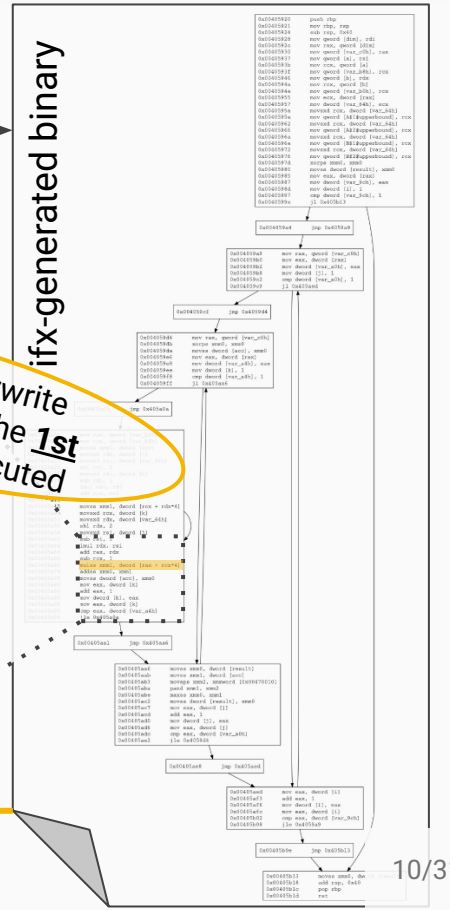


Table of Test Results

	1	2	3	4	5	6	7	8
mul	20							
add								

Iteration in which EXCVATE spoofs an exception

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Replay and overwrite xmm1 with NaN the **2nd** time mulss is executed

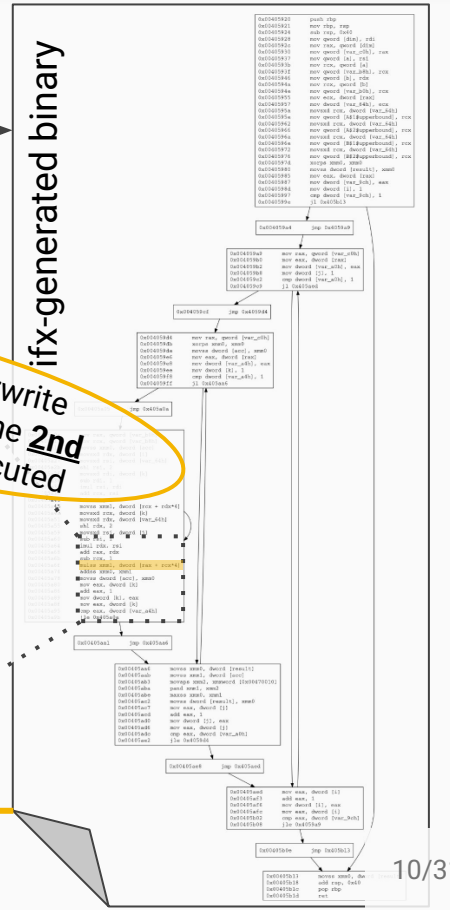


Table of Test Results

	1	2	3	4	5	6	7	8
mul	20	20						
add								

Iteration in which EXCVATE spoofs an exception

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Replay and overwrite xmm1 with NaN the ***3rd*** time mulss is executed

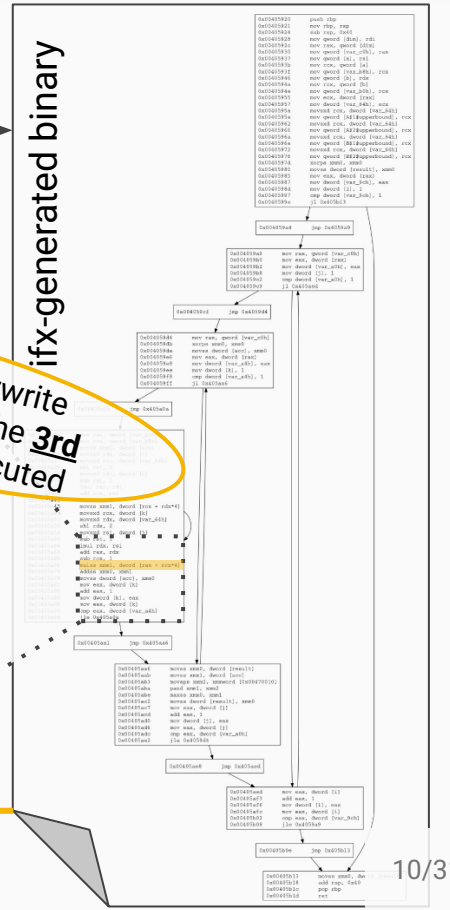


Table of Test Results

	1	2	3	4	5	6	7	8
mul	20	20	20					
add								

Iteration in which EXCVATE spoofs an exception

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Replay and overwrite xmm1 with NaN the **4th** time mulss is executed

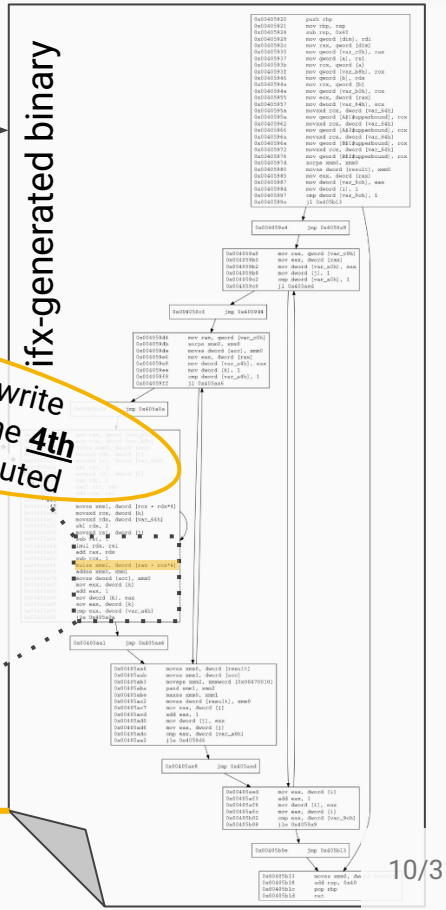


Table of Test Results

	1	2	3	4	5	6	7	8
mul	20	20	20	20				
add								

Iteration in which EXCVATE spoofs an exception

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Replay and overwrite xmm1 with NaN the ***5th*** time mulss is executed

ifx-generated binary

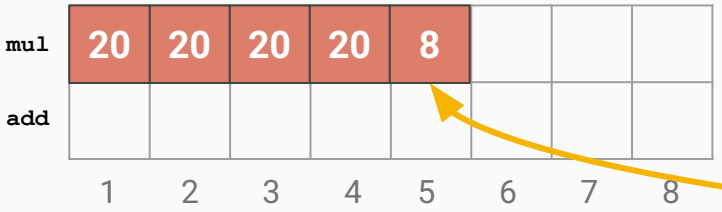


Table of Test Results

Iteration in which EXCVATE spoofs an exception

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Replay and overwrite xmm1 with NaN the ***6th*** time mulss is executed

ifx-generated binary

Table of Test Results

	1	2	3	4	5	6	7	8
mul	20	20	20	20	8	8		
add								

Iteration in which EXCVATE spoofs an exception

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Replay and overwrite xmm1 with NaN the **7th** time mulss is executed

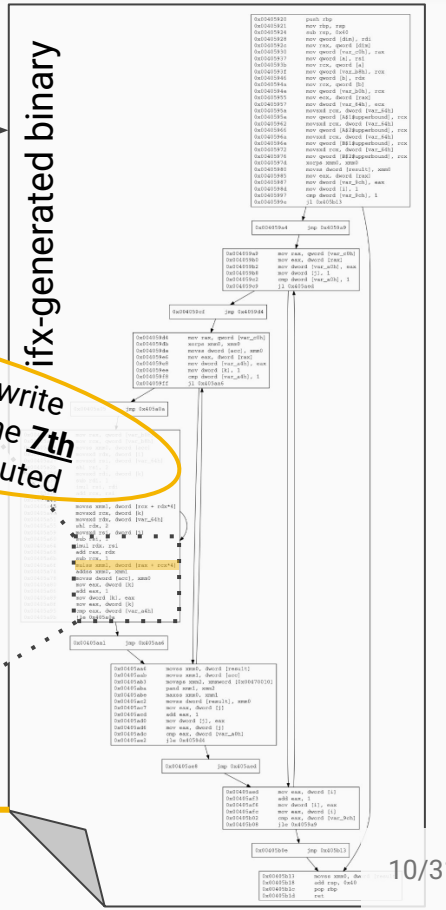


Table of Test Results

	1	2	3	4	5	6	7	8
mul	20	20	20	20	8	8	NaN	
add								

Iteration in which EXCVATE spoofs an exception

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Replay and overwrite xmm1 with NaN the **8th** time mulss is executed

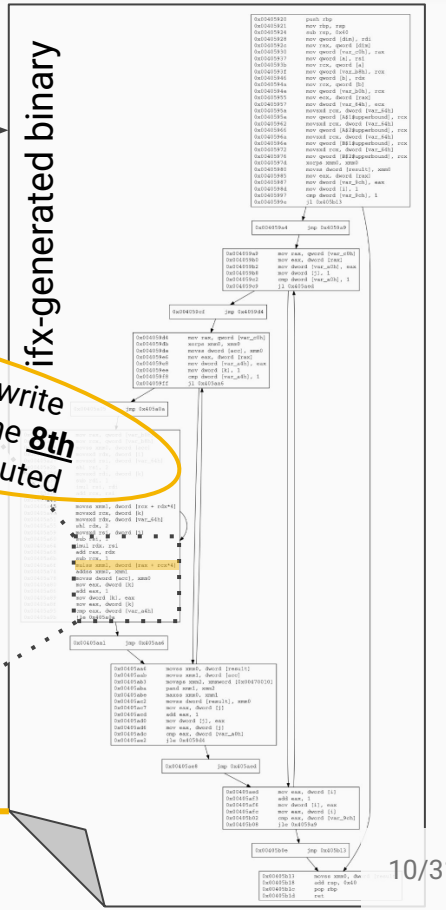


Table of Test Results

	1	2	3	4	5	6	7	8
mul	20	20	20	20	8	8	NaN	NaN
add								

Iteration in which EXCVATE spoofs an exception

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

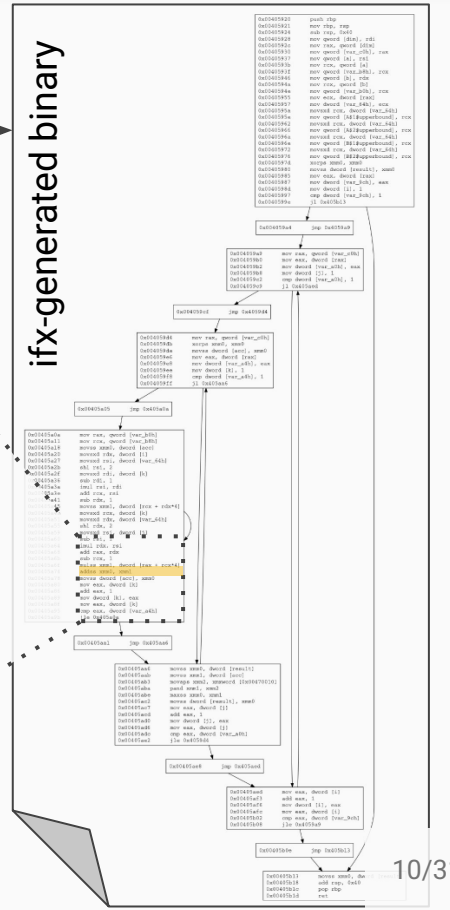
There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Table of Test Results

	1	2	3	4	5	6	7	8
mul	20	20	20	20	8	8	NaN	NaN
add								

Iteration in which EXCVATE spoofs an exception

Approach: spoof exceptions via binary rewriting to cheaply find handling failures

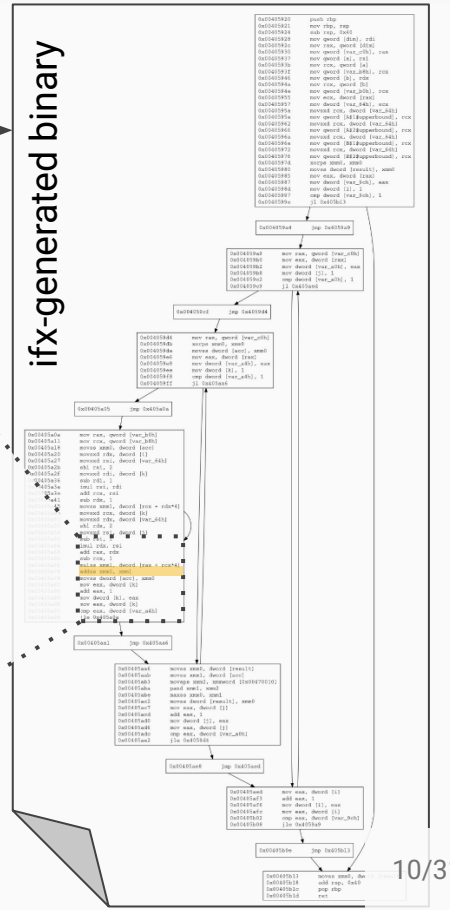
There are only two instructions in the ifx binary that could encounter “Invalid” exceptions.

These two are in a block that is executed eight times -> 2 x 8 = 16 possible exception sites.

So, replay the function execution 16 times, **spoofing** a different Invalid exception each time.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
⋮
sub rsi, 1
imul rdx, rsi
add rax, rdx
sub rcx, 1
mulss xmm1, dword [rax + rcx*4]
addss xmm0, xmm1
movss dword [rbp-0x98], xmm0
mov eax, dword [rbp-0x8c]
add eax, 1
mov dword [rbp-0x8c], eax
mov eax, dword [rbp-0x8c]
cmp eax, dword [rbp-0xa4]
jle 0x405a0a
```

Table of Test Results

	1	2	3	4	5	6	7	8
mul	20	20	20	20	8	8	NaN	NaN
add	20	20	20	20	8	8	NaN	NaN

Iteration in which EXCVATE spoofs an exception

Challenge 2: How do we “reify” a spoofed exception that results in an exception-handling failure?

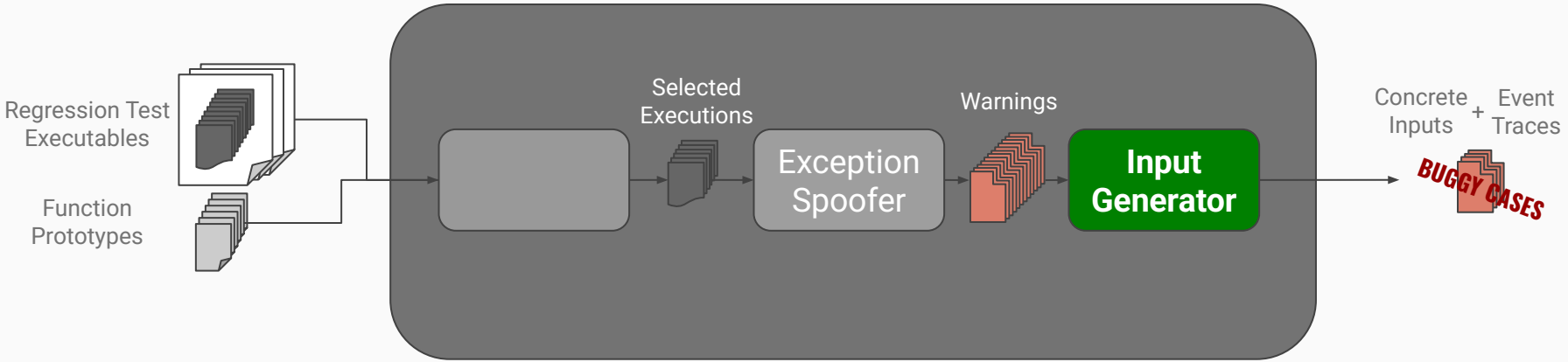
Challenge 2: How do we “reify” a spoofed exception that results in an exception-handling failure?

For a **true buggy case**, EXCVATE must create an input that:

1. triggers the spoofed Invalid exception
2. preserves the control flow that resulted in failed exception handling

Challenge 2: How do we “reify” a spoofed exception that results in an exception-handling failure?

Approach: encode the desired behavior (exception + control flow) into an SMT query

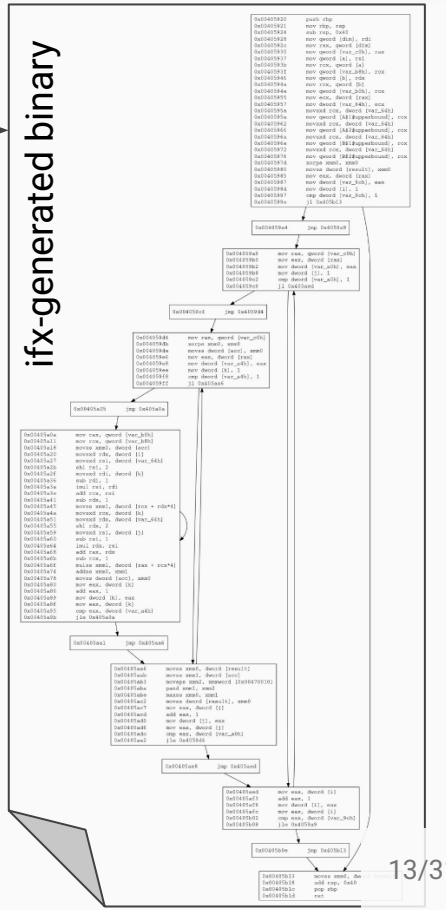


Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$



Approach: encode the desired behavior (control flow + exception) into an SMT query

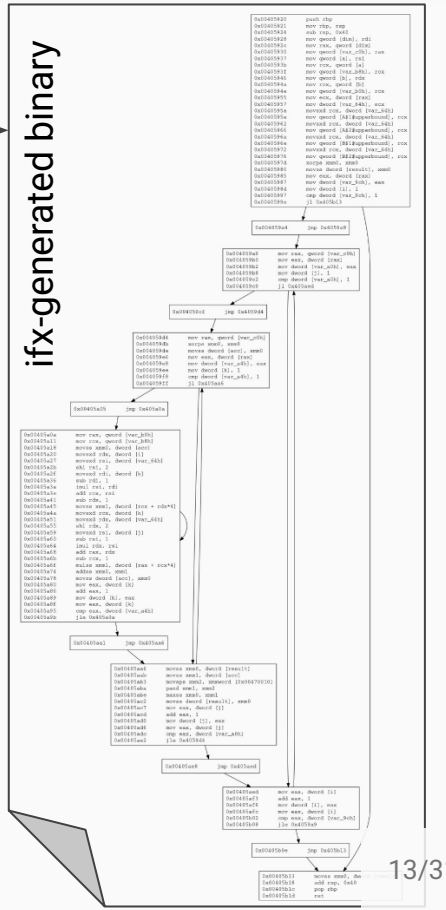
Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

$$A_{sym} = \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix}$$

$$B_{sym} = \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix}$$



Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

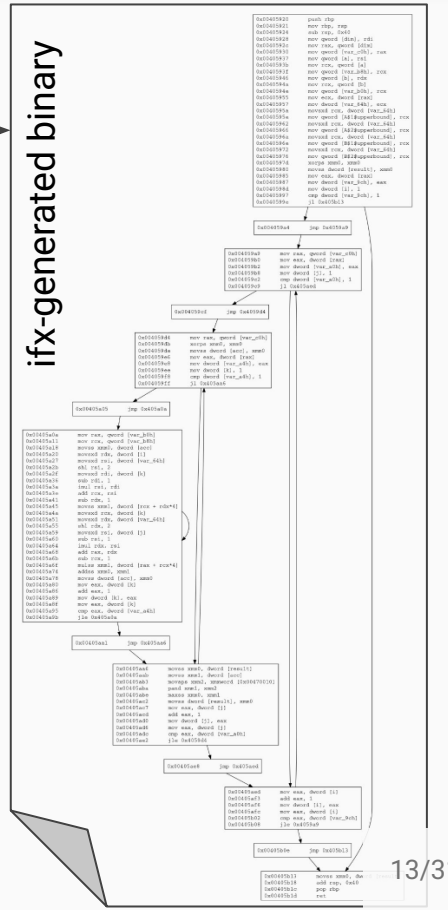
$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

$$A_{sym} = \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix}$$

$$B_{sym} = \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix}$$

ifx-generated binary



Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

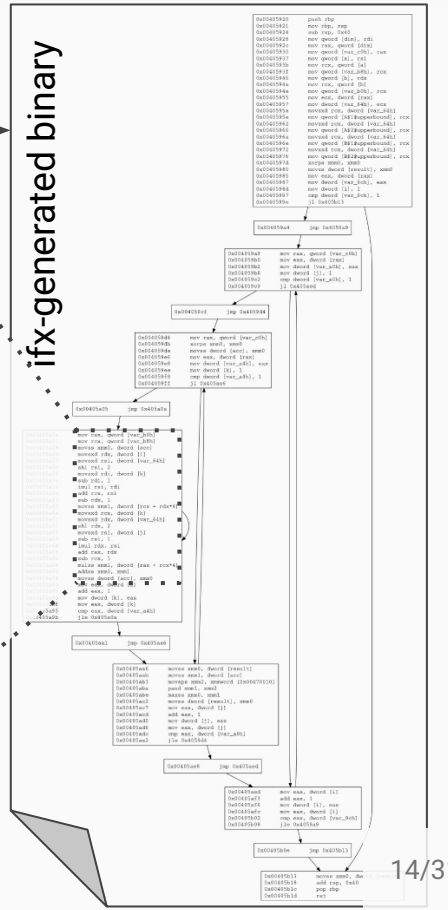
Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

```
result = 0.0
do i = 1, dim
    do j = 1, dim
        acc = 0.0
        do k = 1, dim
            acc = acc + A(i,k) * B(k,j)
        end do
        result = max(result, abs(acc))
    end do
end do
```

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
    ⋮
movss xmm0, dword ptr [rbp-0x98]
    ⋮
movss xmm1, dword ptr [rcx+rdx*4]
    ⋮
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
    ⋮
```

ifx-generated binary



Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

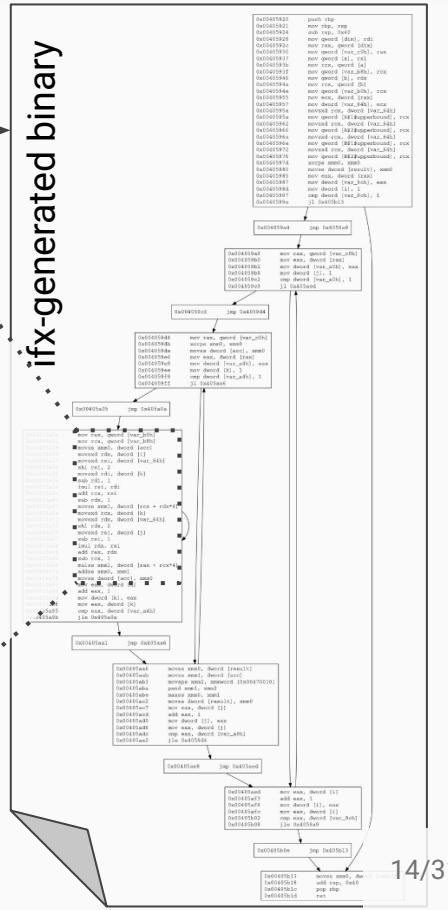
Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
result = 0.0
do i = 1, dim
    do j = 1, dim
        acc = 0.0
        do k = 1, dim
            acc = acc + A(i,k) * B(k,j)
        end do
        result = max(result, abs(acc))
    end do
end do
```

```
    ⋮
    -> load acc
movss xmm0, dword ptr [rbp-0x98]
    ⋮
movss xmm1, dword ptr [rcx+rdx*4]
    ⋮
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
    ⋮
```

ifx-generated binary



Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

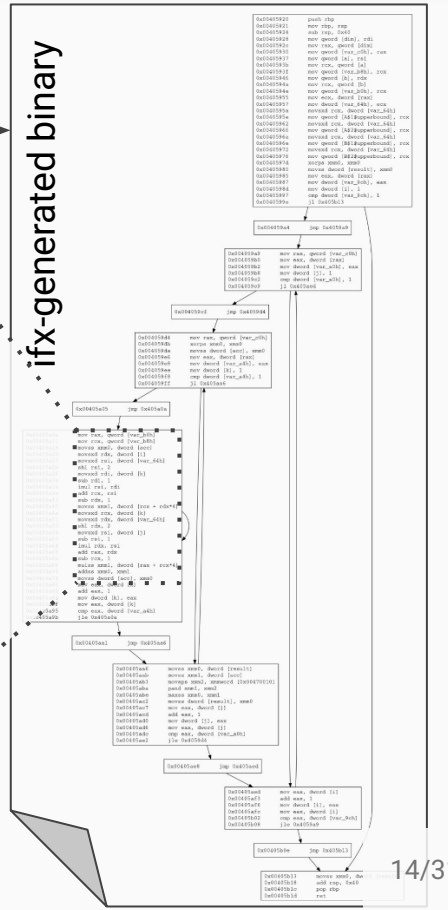
Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

```
result = 0.0
do i = 1, dim
    do j = 1, dim
        acc = 0.0
        do k = 1, dim
            acc = acc + A(i,k) * B(k,j)
        end do
        result = max(result, abs(acc))
    end do
end do
```

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
    ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
    ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
    ⋮
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
    ⋮
```

ifx-generated binary



Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

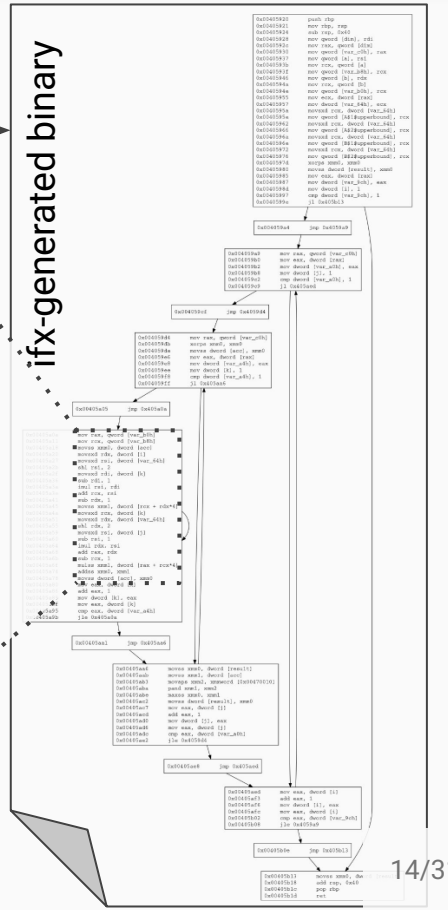
Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

```
result = 0.0
do i = 1, dim
    do j = 1, dim
        acc = 0.0
        do k = 1, dim
            acc = acc + A(i,k) * B(k,j)
        end do
        result = max(result, abs(acc))
    end do
end do
```

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
    ⋮
                -> load acc
movss xmm0, dword ptr [rbp-0x98]
    ⋮
                -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
    ⋮
                -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
    ⋮
```

ifx-generated binary



Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

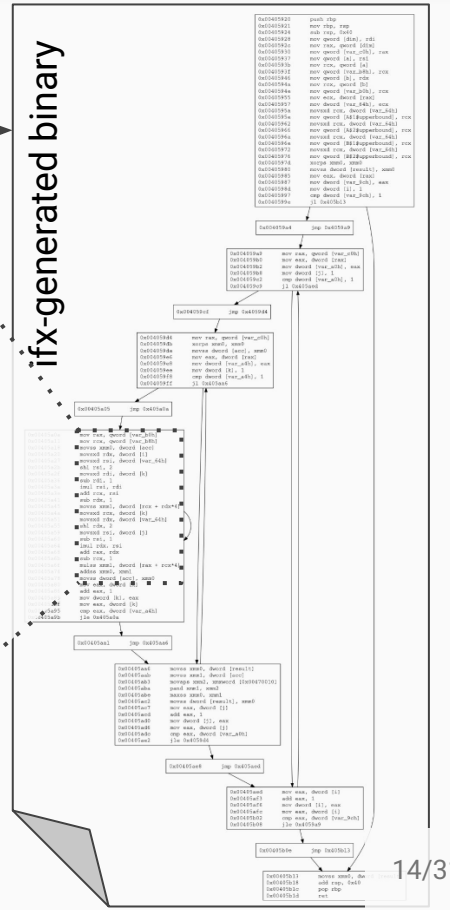
Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

```
result = 0.0
do i = 1, dim
    do j = 1, dim
        acc = 0.0
        do k = 1, dim
            acc = acc + A(i,k) * B(k,j)
        end do
        result = max(result, abs(acc))
    end do
end do
```

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
⋮
                -> load acc
movss xmm0, dword ptr [rbp-0x98]
⋮
                -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
⋮
                -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
⋮
                -> accumulate and save acc
```

ifx-generated binary



Active Symbols

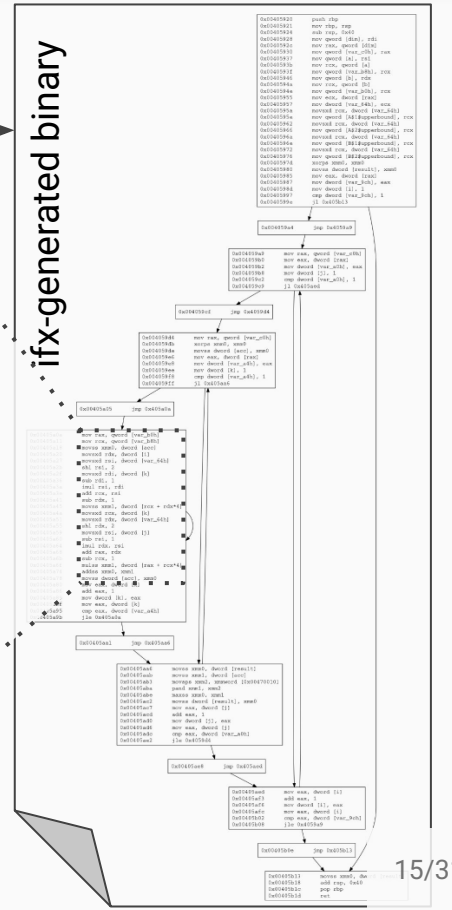
```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
⋮
     -> load acc
movss xmm0, dword ptr [rbp-0x98]
⋮
     -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
⋮
     -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
⋮
     -> accumulate and save acc
```

Active Symbols

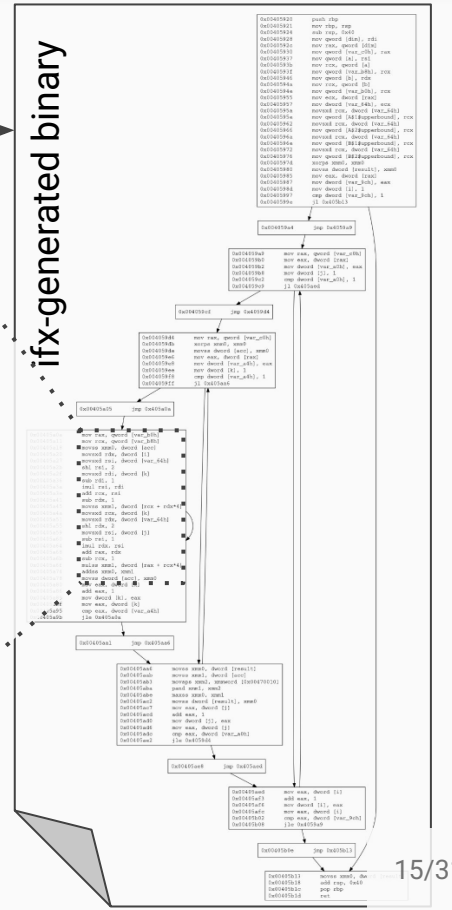
```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
⋮
    -> load acc
movss xmm0, dword ptr [rbp-0x98]
⋮
    -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
⋮
    -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
⋮
    -> accumulate and save acc
```

Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Read Operands

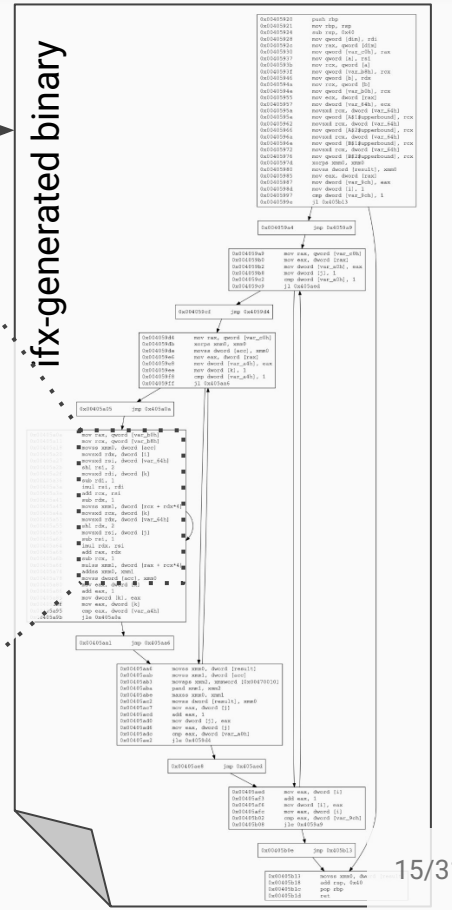
```
0x7ffef9945e28
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
⋮
                -> load acc
movss xmm0, dword ptr [rbp-0x98]
⋮
                -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
⋮
                -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
⋮
                -> accumulate and save acc
```

Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Read Operands

NOT IN ACTIVE SYMBOLS

Approach: encode the desired behavior (control flow + exception) into an SMT query

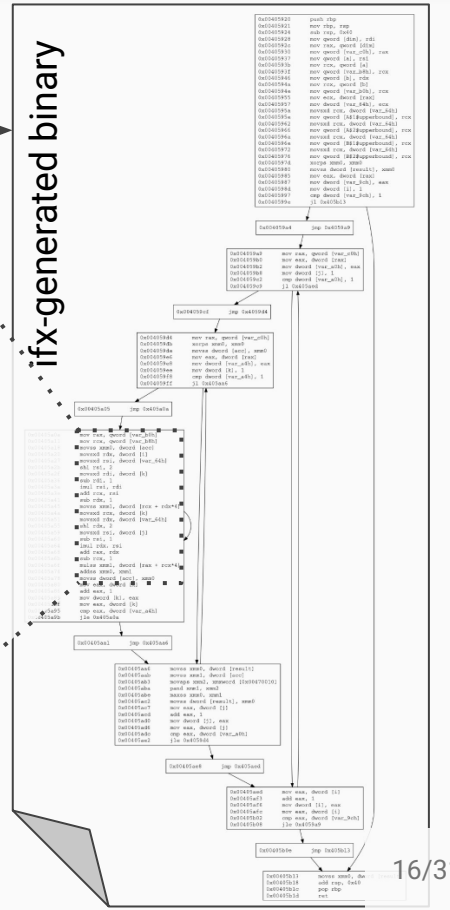
Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

ifx-generated binary



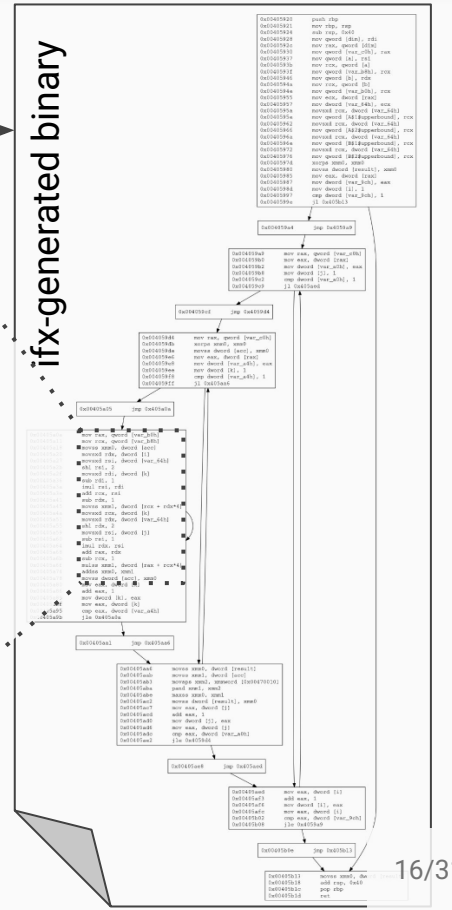
Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Read Operands

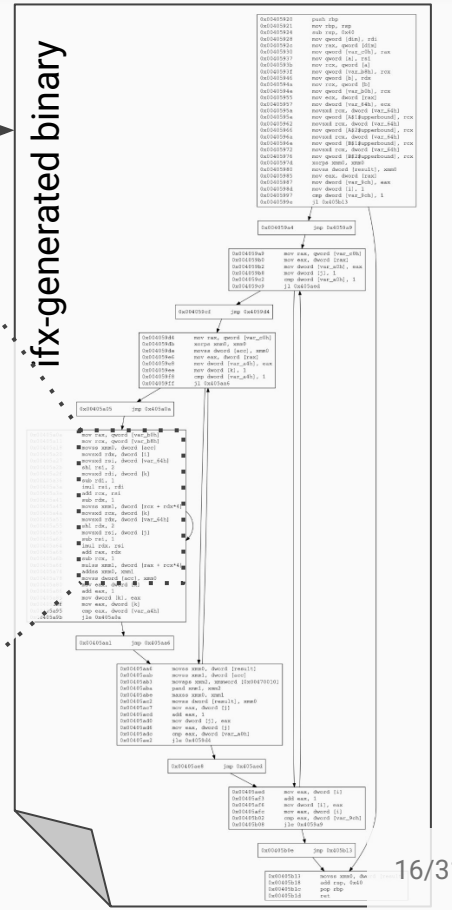
```
0x7f9dea0970a4
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
⋮
    -> load acc
movss xmm0, dword ptr [rbp-0x98]
⋮
    -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
⋮
    -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
⋮
    -> accumulate and save acc
```

Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Read Operands

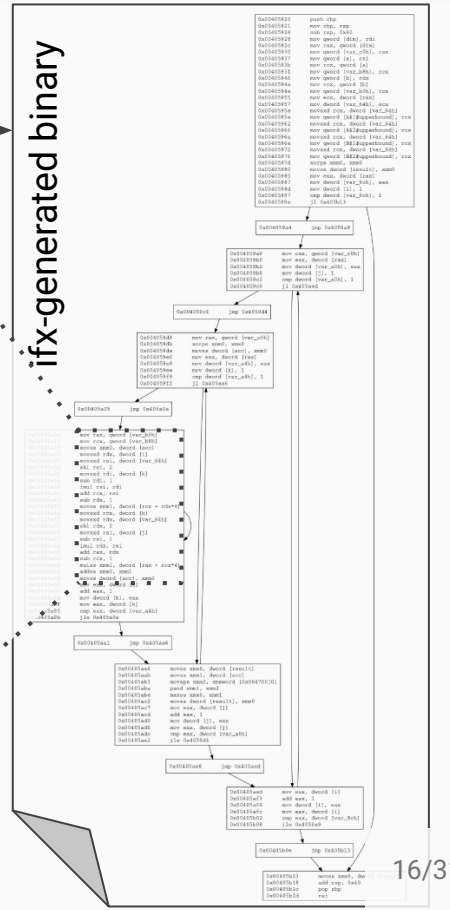
```
0x7f9dea0970a4
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

```
assert ( = xmm1_0 0x7f9dea0970a4_0 )
```

SMT query

Active Symbols

```
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Read Operands

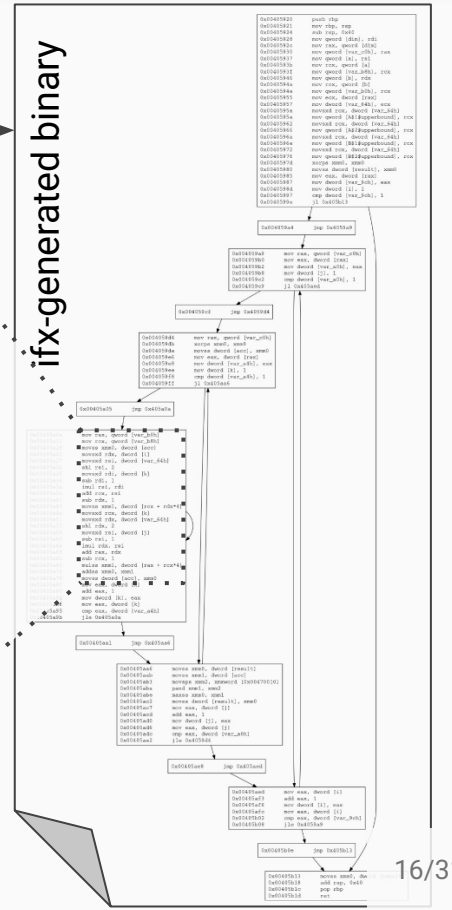
```
0x7f9dea0970a4
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

```
assert ( = xmm1_0 0x7f9dea0970a4_0 )
```

SMT query

Active Symbols

```
a_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Read Operands

```
0x7f9dea0970a4
```

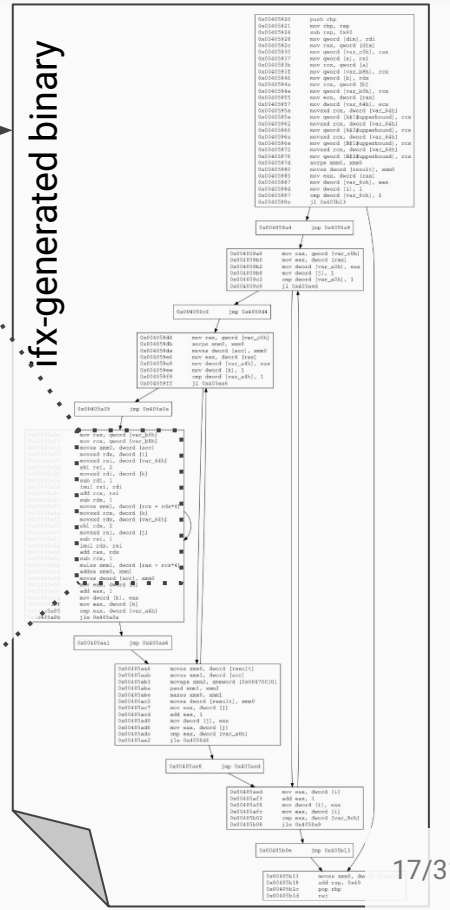
Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

ifx-generated binary



```
      ⋮
        -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
        -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
        -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
        -> accumulate and save acc
```

SMT query

Active Symbols

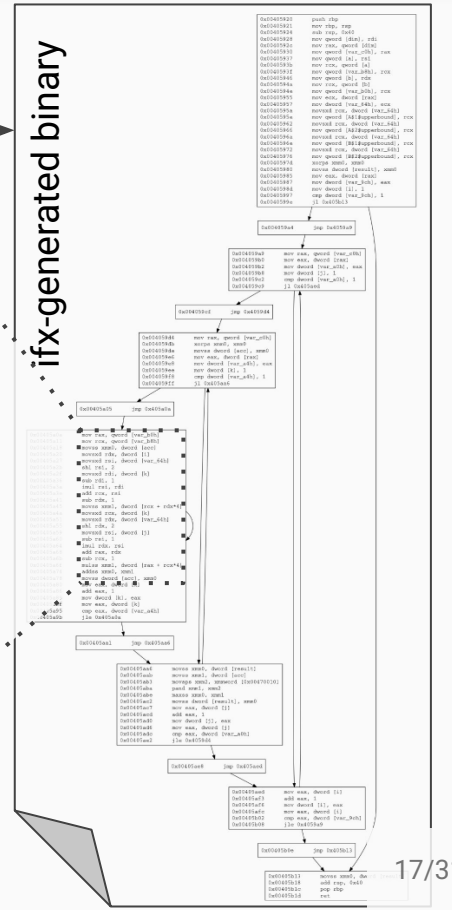
```
a_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

SMT query

Active Symbols

```
a_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Read Operands

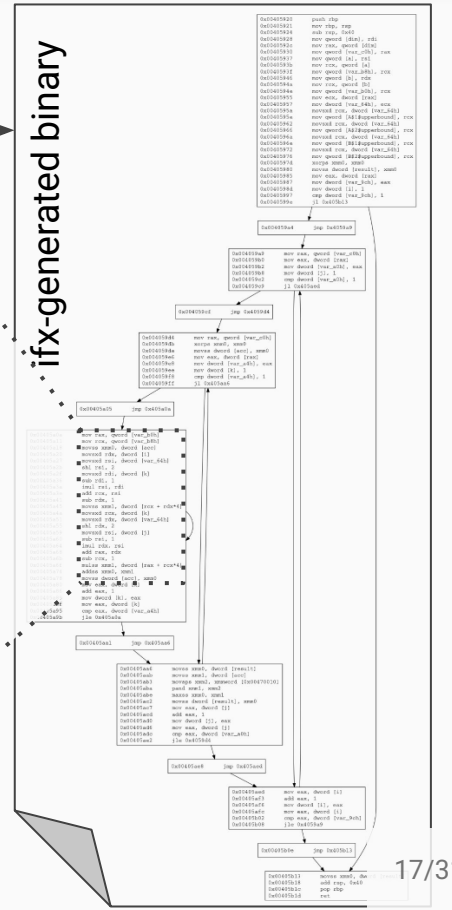
```
xmm1
0x7f9dea0970b4
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

SMT query

Active Symbols

```
a_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Read Operands

```
xmm1
0x7f9dea0970b4
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

```
assert ( =
          xmm1_1
          ( fp.mul rm
                xmm1_0
                0x7f9dea0970b4_0
          )
        )
```

SMT query

Active Symbols

```
a_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

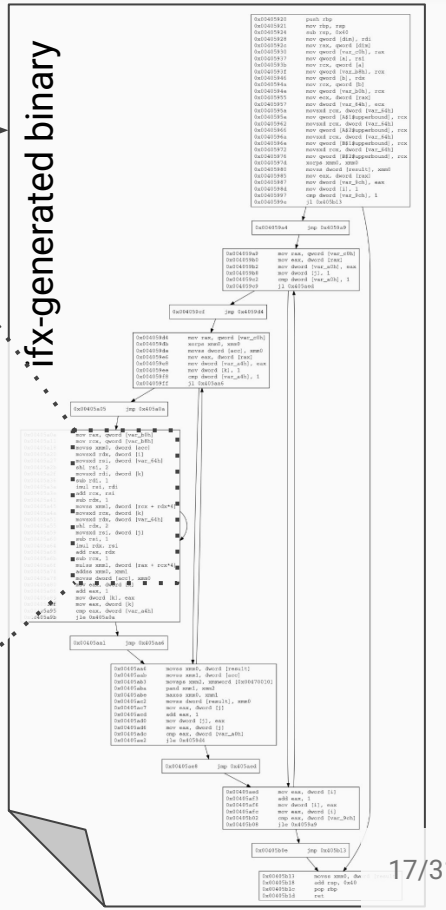
$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

Read Operands

```
xmm1
0x7f9dea0970b4
```

ifx-generated binary



Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
assert ( =
          xmm1_1
          ( fp.mul rm
                xmm1_0
                0x7f9dea0970b4_0
          )
        )
```

SMT query

Active Symbols

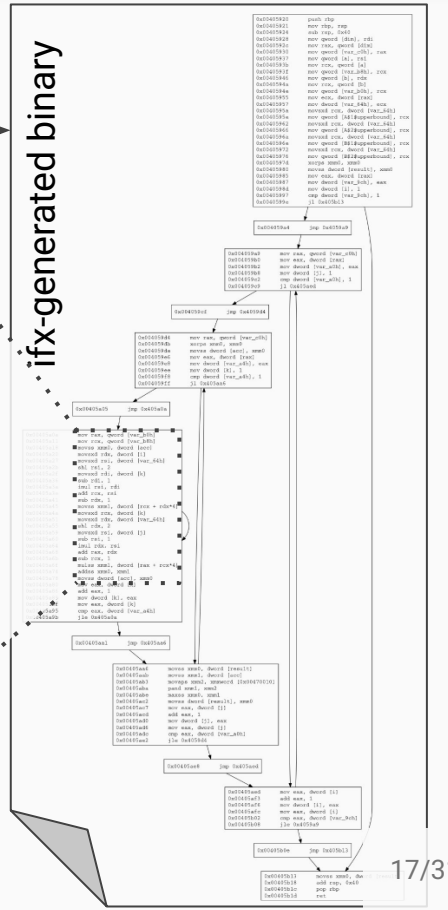
```
a_0 * b_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

```
⋮
   -> load acc
movss xmm0, dword ptr [rbp-0x98]
⋮
   -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
⋮
   -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
⋮
   -> accumulate and save acc
```

Read Operands

```
xmm1
0x7f9dea0970b4
```

ifx-generated binary



Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from **spoofing an Invalid exception in the first execution of the multiply instruction.**

```
assert ( =
          xmm1_1
          ( fp.mul rm
                xmm1_0
                0x7f9dea0970b4_0
          )
        )
```

```
assert ( fp.isNaN xmm1_1 )
```

SMT query

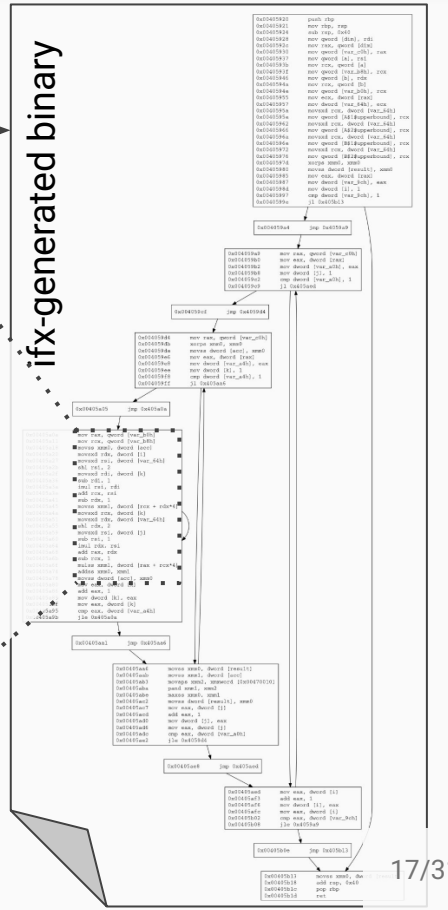
Active Symbols

```
a_0 * b_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

ifx-generated binary



Read Operands

```
xmm1
0x7f9dea0970b4
```

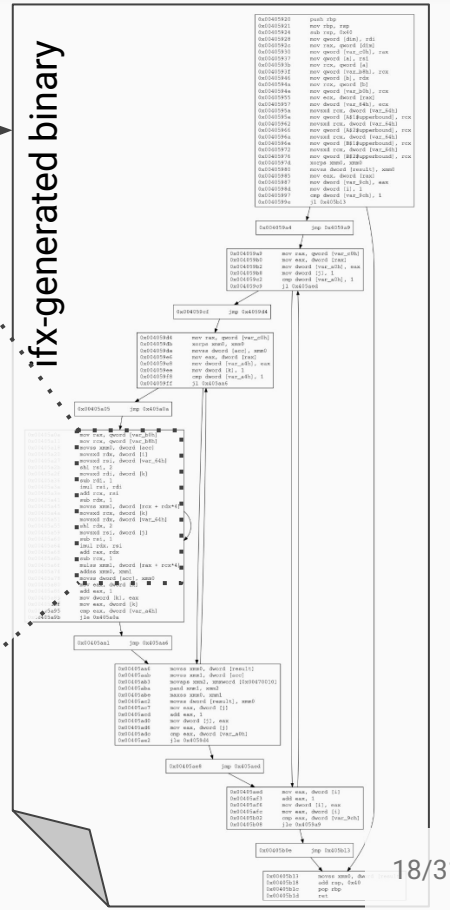
Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

ifx-generated binary



```
      ⋮
        -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
        -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
        -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
        -> accumulate and save acc
```

Read Operands

```
xmm1
xmm0
```

SMT query

Active Symbols

```
a_0 * b_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
assert ( =
          xmm0_0
          ( fp.add rm
               #b00000000000000000000000000000000
               xmm1_1
          )
        )
```

SMT query

Active Symbols

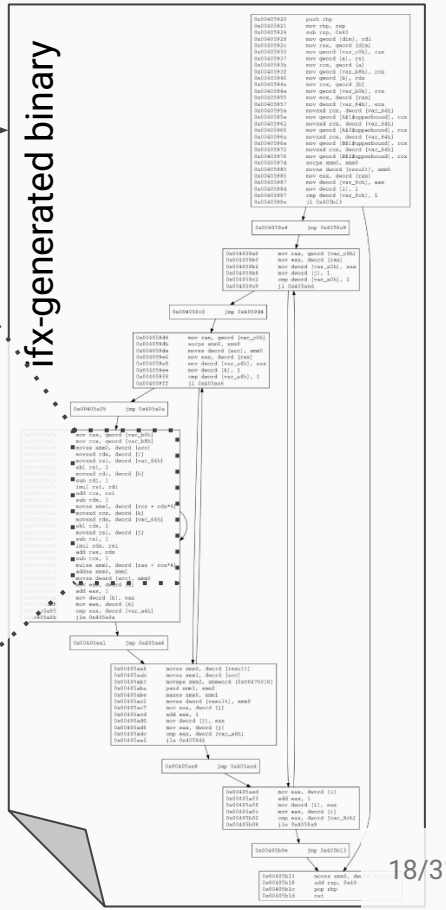
```
a_0 * b_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

```
⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
⋮
      -> accumulate and save acc
```

Read Operands

```
xmm1
xmm0
```

ifx-generated binary



Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

```
assert ( =
          xmm0_0
          ( fp.add rm
               #b00000000000000000000000000000000
               xmm1_1
          )
        )
```

SMT query

Active Symbols

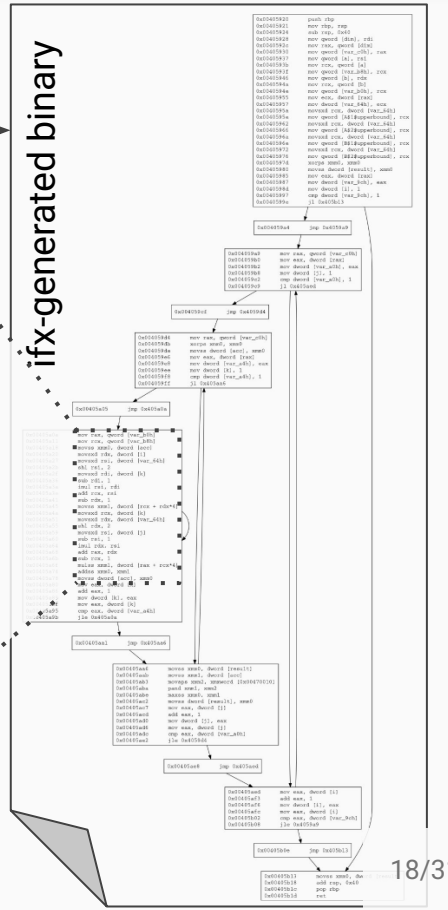
```
0 + a_0 * b_0 : xmm0
    a_0 * b_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

Read Operands

```
xmm1
xmm0
```

ifx-generated binary



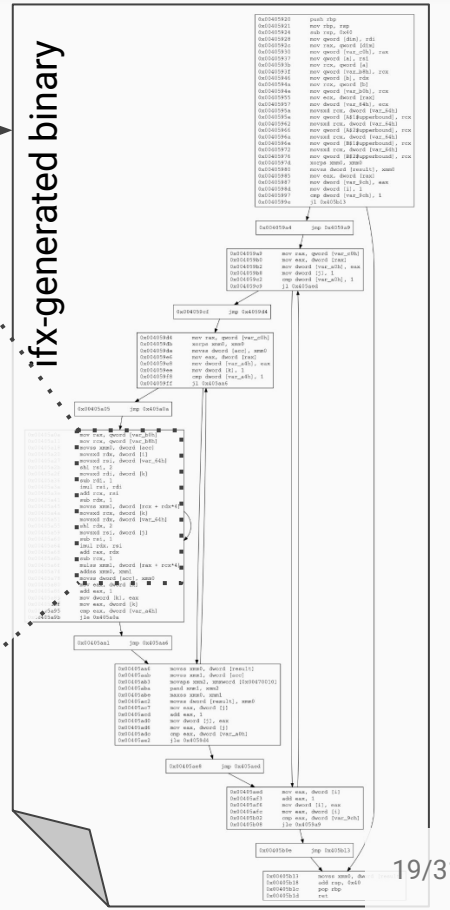
Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$

ifx-generated binary



```
      ⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
      ⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
      ⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
      ⋮
      -> accumulate and save acc
```

Read Operands

```
xmm0
```

SMT query

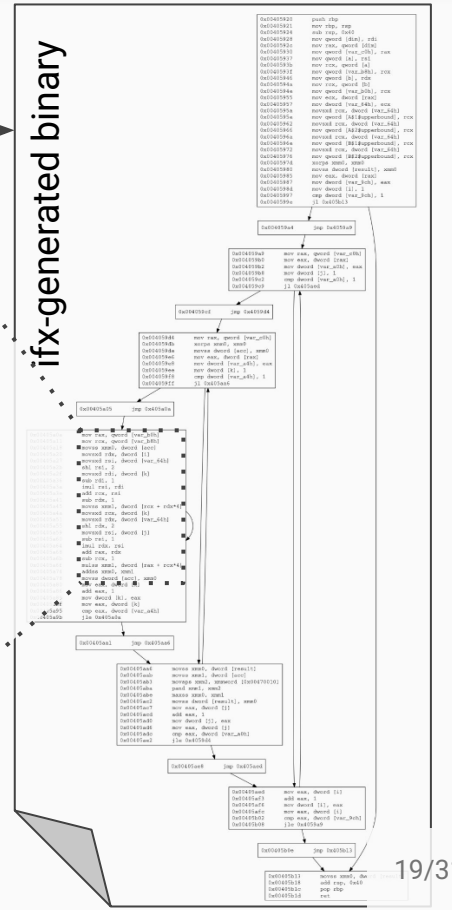
Active Symbols

```
0 + a_0 * b_0 : xmm0
  a_0 * b_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
assert ( = 0x7ffef9945e28_1 xmm0_0 )
```

SMT query

Active Symbols

```
0 + a_0 * b_0 : xmm0
  a_0 * b_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

```
⋮
     -> load acc
movss xmm0, dword ptr [rbp-0x98]
⋮
     -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
⋮
     -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
⋮
     -> accumulate and save acc
```

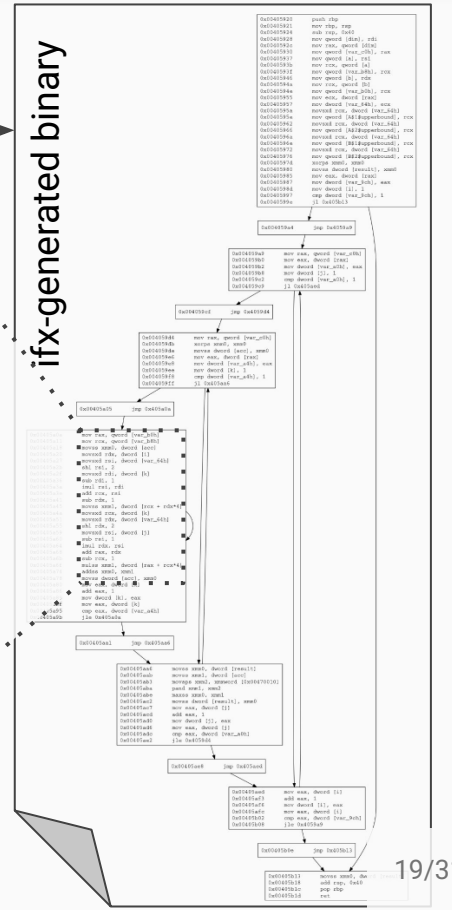
Read Operands

```
xmm0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}$$




```
assert ( = 0x7ffef9945e28_1 xmm0_0 )
```

SMT query

Active Symbols

```
0x7ffef9945e28
0 + a_0 * b_0 : xmm0
  a_0 * b_0 : xmm1
a_0 : 0x7f9dea0970a4
a_1 : 0x7f9dea0970a8
a_2 : 0x7f9dea0970ac
a_3 : 0x7f9dea0970b0
b_0 : 0x7f9dea0970b4
b_1 : 0x7f9dea0970b8
b_2 : 0x7f9dea0970bc
b_3 : 0x7f9dea0970c0
```

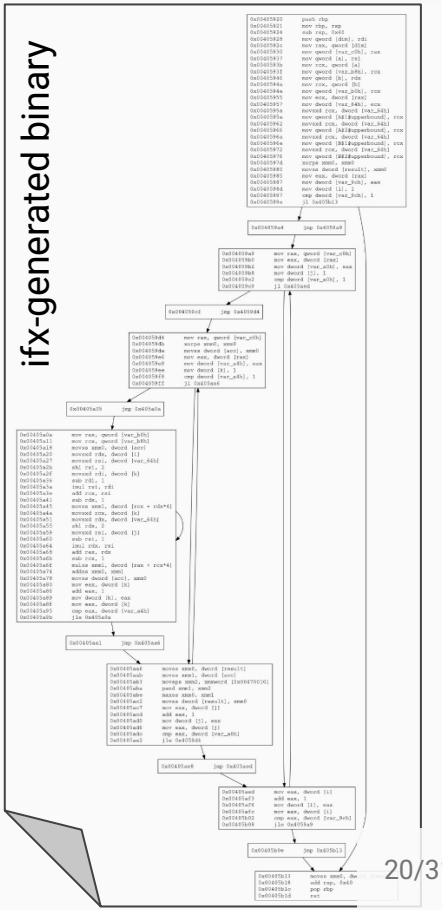
```
⋮
      -> load acc
movss xmm0, dword ptr [rbp-0x98]
⋮
      -> load A(i,k)
movss xmm1, dword ptr [rcx+rdx*4]
⋮
      -> multiply with B(k,j)
mulss xmm1, dword ptr [rax+rcx*4]
addss xmm0, xmm1
movss dword ptr [rbp-0x98], xmm0
⋮
      -> accumulate and save acc
```

Read Operands

```
xmm0
```

Approach: encode the desired behavior (control flow + exception) into an SMT query

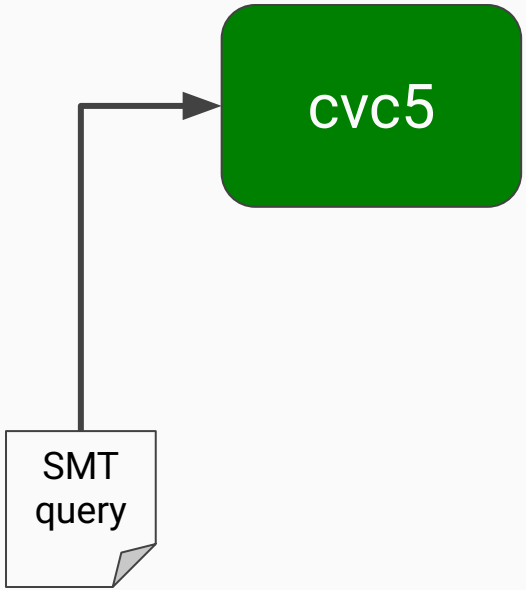
Let’s check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

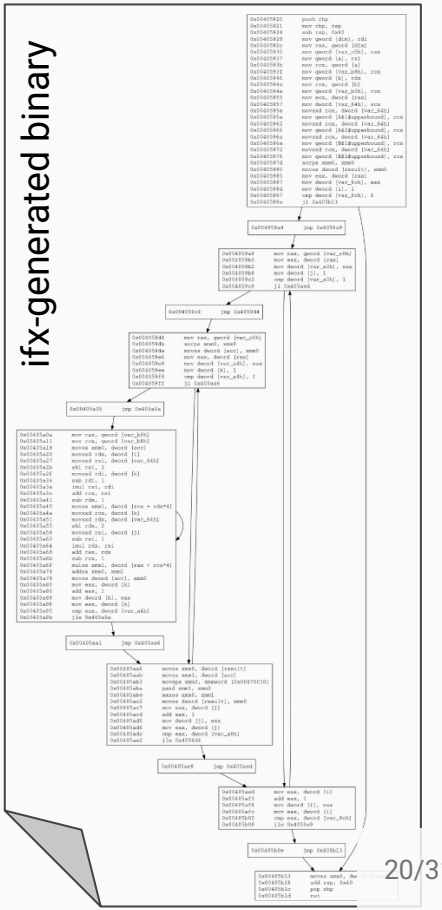


SMT query

Approach: encode the desired behavior (control flow + exception) into an SMT query

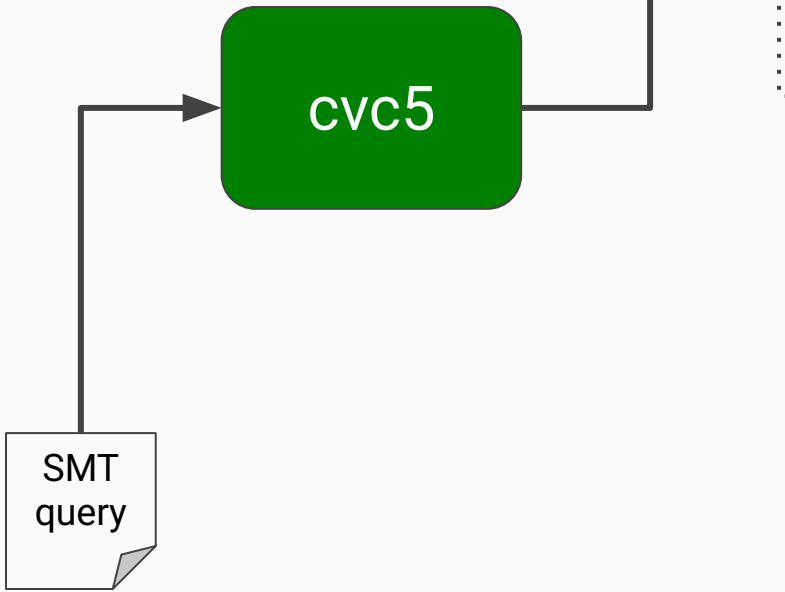
Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

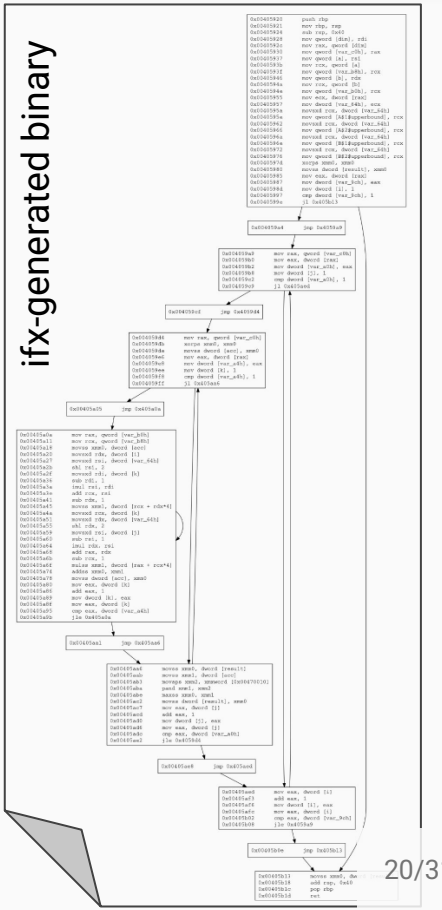




Approach: encode the desired behavior (control flow + exception) into an SMT query

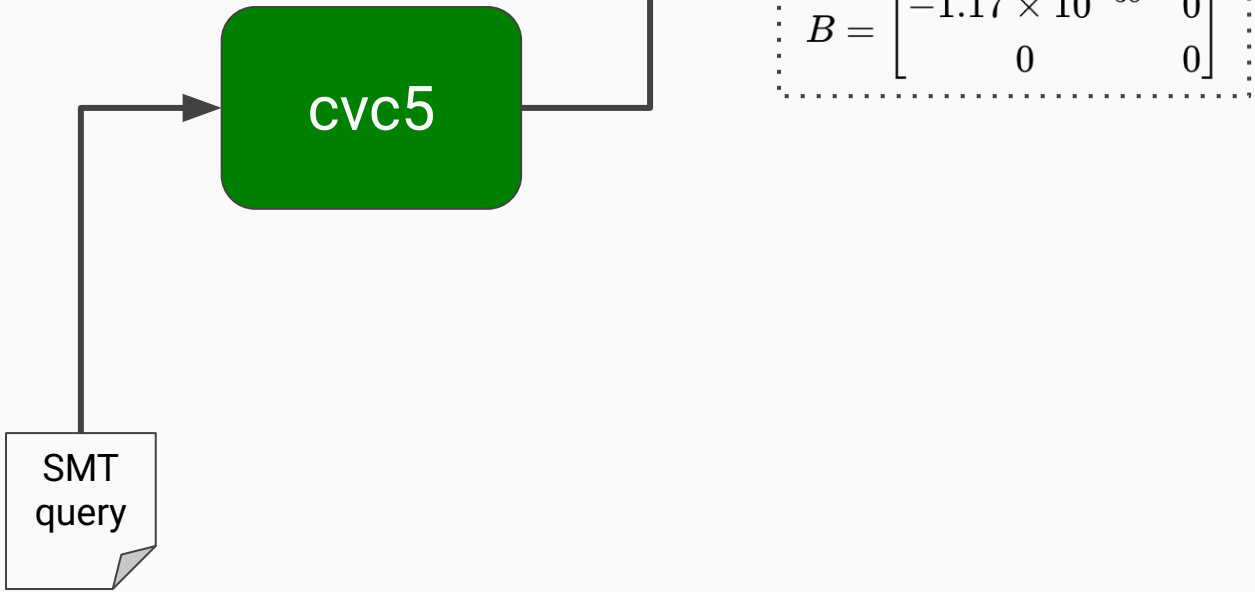
Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

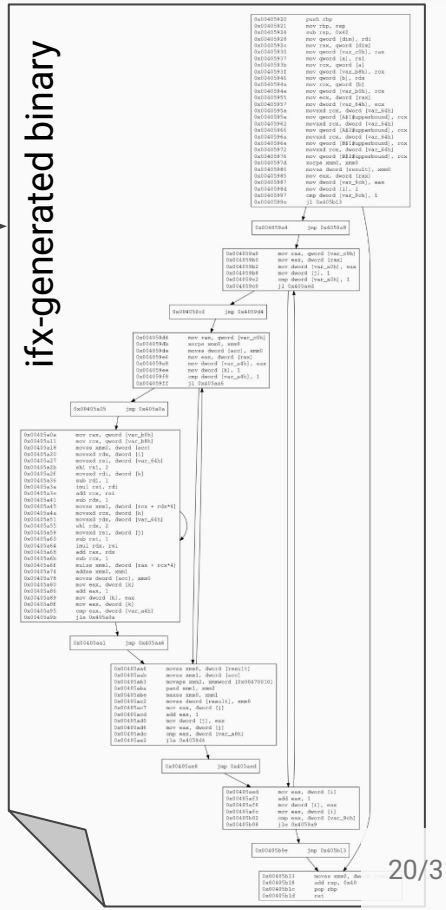




Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.





Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

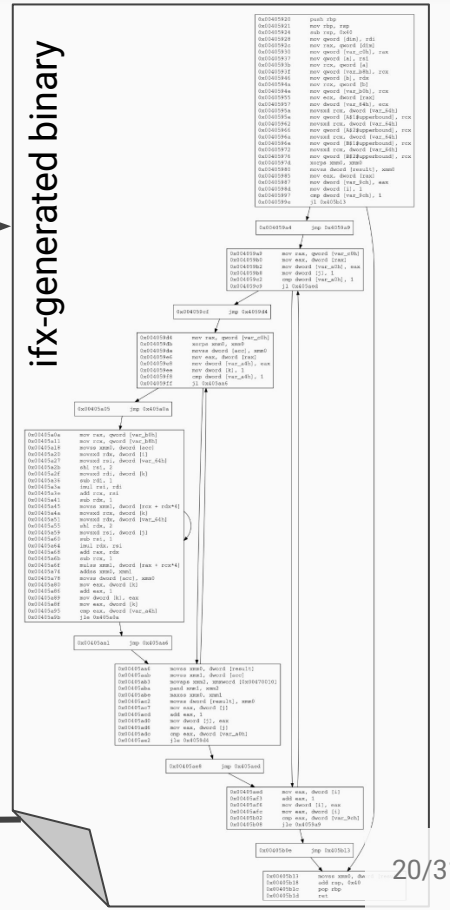
SMT query

cvc5

$$A = \begin{bmatrix} NaN & 0 \\ 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} -1.17 \times 10^{-38} & 0 \\ 0 & 0 \end{bmatrix}$$

ifx-generated binary



0

Approach: encode the desired behavior (control flow + exception) into an SMT query

Let's check the *first warning* which resulted from spoofing an Invalid exception in the first execution of the multiply instruction.

$$A = \begin{bmatrix} NaN & 0 \\ 0 & 0 \end{bmatrix}$$

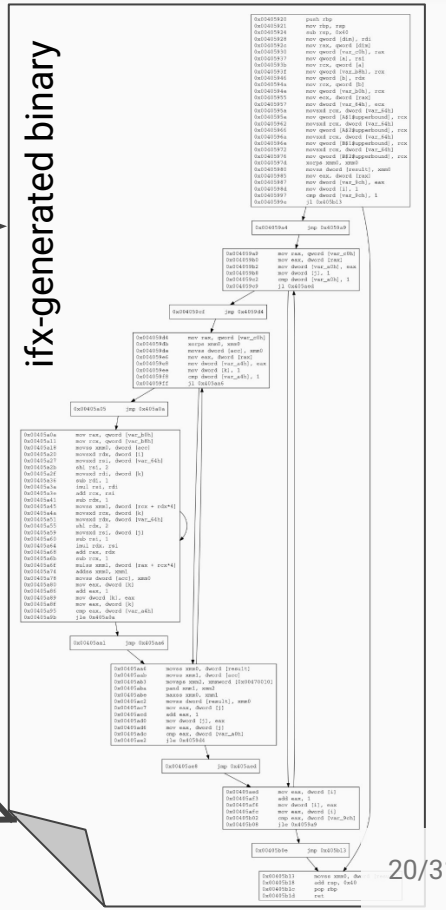
$$B = \begin{bmatrix} -1.17 \times 10^{-38} & 0 \\ 0 & 0 \end{bmatrix}$$

cvc5

SMT query

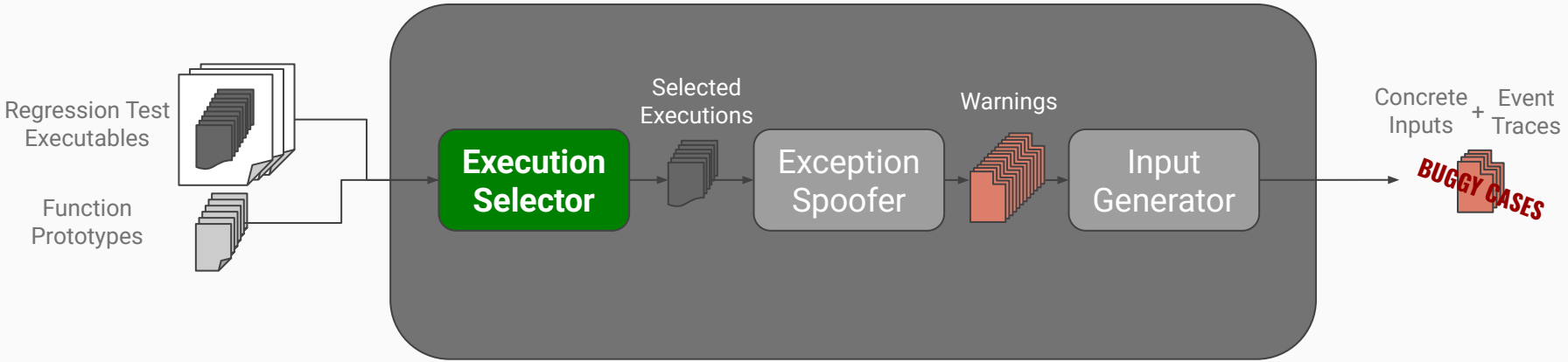
Disassembly	Source Location	Event	Taint Count
movss xmm1, dword ptr [rcx+rdx*4]	main.f90:17	G---	1
mulss xmm1, dword ptr [rax+rcx*4]	main.f90:17	-P-r	1
addss xmm0, xmm1	main.f90:17	-P-r	2
...			
maxss xmm0, xmm1	main.f90:19	--Kr	1
movss dword ptr [rbp-0x44], xmm0	main.f90:19	--K-	0

0



Challenge 3: How do we get a representative set of function executions to test?

Approach: Take executions from the library's regression tests



Evaluation: A BLAS Case Study

- **26 functions** from Levels 1 & 2
- **Reference BLAS** (v1.12), **OpenBLAS** (v0.3.28), and **BLIS** (v1.0)
- **GNU** (gfortran, gcc) and **Intel** (ifx, icx) compilers
  - **various optimization combinations**: default, -O3, and -ffast-math/-fp-model=fast=[1|2], etc...

-> 598 (function, implementation, compiler, optimizations) tuples

-> 12 hours of total testing time

Finding 1: Only 4.4% of spoofed exceptions resulted in warnings.

Finding 2: EXCVATE found inputs triggering exception-handling failures in 5/26 BLAS functions

`sgemv` / `sger`

$$\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

$$\mathbf{A} := \alpha\mathbf{x}\mathbf{y}^T + \mathbf{A}$$

EXCVATE found exception-handling failures caused by **compiler optimizations that change control flow** in the face of comparisons involving NaN

sgemv / sger

$$\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

$$\mathbf{A} := \alpha\mathbf{x}\mathbf{y}^T + \mathbf{A}$$

EXCVATE found exception-handling failures caused by **compiler optimizations that change control flow** in the face of comparisons involving NaN

```
(in) TRANS: N
(in) M: 1
(in) N: 3
(in) ALPHA: 1.14752e-41
(in) LDA: 2
(in) INCX: 1
(in) BETA: 1
(in) INCY: 1
(in) A: 0 0 0 0 0 0
(in) X: nan 0 0
(in) Y: 0
```

sgemv / sger

$$\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

$$\mathbf{A} := \alpha\mathbf{x}\mathbf{y}^T + \mathbf{A}$$

EXCVATE found exception-handling failures caused by **compiler optimizations that change control flow** in the face of comparisons involving NaN

```
(in) TRANS: N
(in) M: 1
(in) N: 3
(in) ALPHA: 1.14752e-41
(in) LDA: 2
(in) INCX: 1
(in) BETA: 1
(in) INCY: 1
(in) A: 0 0 0 0 0 0
(in) X: nan 0 0
(in) Y: 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

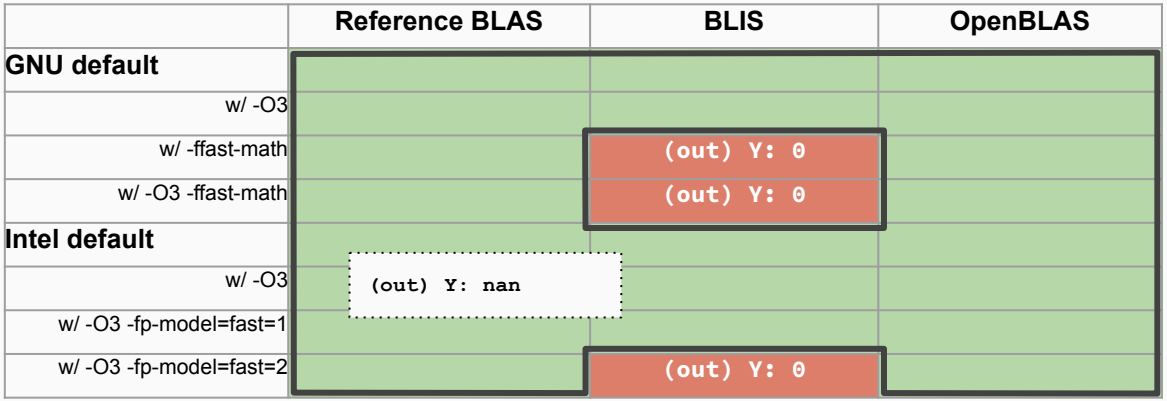
sgemv / sger

$$\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

$$\mathbf{A} := \alpha\mathbf{x}\mathbf{y}^T + \mathbf{A}$$

EXCVATE found exception-handling failures caused by **compiler optimizations that change control flow** in the face of comparisons involving NaN

```
(in) TRANS: N
(in) M: 1
(in) N: 3
(in) ALPHA: 1.14752e-41
(in) LDA: 2
(in) INCX: 1
(in) BETA: 1
(in) INCY: 1
(in) A: 0 0 0 0 0 0
(in) X: nan 0 0
(in) Y: 0
```



sgemv / sger

$$\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

$$\mathbf{A} := \alpha\mathbf{x}\mathbf{y}^T + \mathbf{A}$$

EXCVATE found exception-handling failures caused by **compiler optimizations that change control flow** in the face of comparisons involving NaN

```
(in) M: 2
(in) N: 1
(in) ALPHA: 1.14752e-41
(in) INCX: 1
(in) INCY: 1
(in) LDA: 3
(in) X: 0 0
(in) Y: nan
(in) A: 0 0 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math	(out) A: 0 0 0	(out) A: 0 0 0	
w/ -O3 -ffast-math	(out) A: 0 0 0	(out) A: 0 0 0	
Intel default			
w/ -O3			(out) A: nan nan 0
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2	(out) A: 0 0 0	(out) A: 0 0 0	

`srotmg / srotm`

Generate and apply a modified Givens rotation, respectively

EXCVATE found multiple different exception-handling failures **necessitating clearer documentation and even possible deprecation**

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

`srotmg / srotm`

Generate and apply a modified Givens rotation, respectively

EXCVATE found multiple different exception-handling failures **necessitating clearer documentation and even possible deprecation**

```
(in) N: 1
(in) INCX: 1
(in) INCY: 1
(in) SPARAM: nan 0 0 0 0
(in) SX: 0
(in) SY: 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

`srotmg / srotm`

Generate and apply a modified Givens rotation, respectively

EXCVATE found multiple different exception-handling failures **necessitating clearer documentation and even possible deprecation**

```
(in) N: 1
(in) INCX: 1
(in) INCY: 1
(in) SPARAM: nan 0 0 0 0
(in) SX: 0
(in) SY: 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math		`(out) SX: 0` `(out) SY: 0`	
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

`srotmg / srotm`

Generate and apply a modified Givens rotation, respectively

EXCVATE found multiple different exception-handling failures **necessitating clearer documentation and even possible deprecation**

```
(in) SD1: inf
(in) SD2: -1.99976
(in) SX1: nan
(in) SY1: -inf
(in) SPARAM: 0 0 0 0 0
```

```
(in) SD1: 1.17549e-38
(in) SD2: -1.4013e-45
(in) SX1: nan
(in) SY1: -1.66667
(in) SPARAM: 0 0 0 0 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

`srotmg / srotm`

Generate and apply a modified Givens rotation, respectively

EXCVATE found multiple different exception-handling failures **necessitating clearer documentation and even possible deprecation**

```
(in) SD1: inf
(in) SD2: -1.99976
(in) SX1: nan
(in) SY1: -inf
(in) SPARAM: 0 0 0 0 0
```

```
(in) SD1: 1.17549e-38
(in) SD2: -1.4013e-45
(in) SX1: nan
(in) SY1: -1.66667
(in) SPARAM: 0 0 0 0 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

```
(out) SD1: 0
(out) SD2: 0
(out) SX1: 0
(out) SPARAM: -1 0 0 0 0 0
```

`srotmg / srotm`

Generate and apply a modified Givens rotation, respectively

EXCVATE found multiple different exception-handling failures **necessitating clearer documentation and even possible deprecation**

```
(in) SD1: inf
(in) SD2: -1.99976
(in) SX1: nan
(in) SY1: -inf
(in) SPARAM: 0 0 0 0 0
```

```
(in) SD1: 1.17549e-38
(in) SD2: -1.4013e-45
(in) SX1: nan
(in) SY1: -1.66667
(in) SPARAM: 0 0 0 0 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

```
(out) SD1: 0
(out) SD2: 0
(out) SX1: 0
(out) SPARAM: -1 0 0 0 0 0
```

UNDOCUMENTED ERROR CODE

`srotmg / srotm`

Generate and apply a modified Givens rotation, respectively

EXCVATE found multiple different exception-handling failures **necessitating clearer documentation and even possible deprecation**

```
(in) SD1: inf
(in) SD2: -1.23382e-05
(in) SX1: -1.81899e-12
(in) SY1: -2.21834e-39
(in) SPARAM: 0 0 0 0 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

`srotmg / srotm`

Generate and apply a modified Givens rotation, respectively

EXCVATE found multiple different exception-handling failures **necessitating clearer documentation and even possible deprecation**

```
(in) SD1: inf
(in) SD2: -1.23382e-05
(in) SX1: -1.81899e-12
(in) SY1: -2.21834e-39
(in) SPARAM: 0 0 0 0 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2		**Undocumented Error Code**	**NaNs in output**

`srotmg / srotm`

Generate and apply a modified Givens rotation, respectively

EXCVATE found multiple different exception-handling failures **necessitating clearer documentation and even possible deprecation**

```
(in) SD1: inf
(in) SD2: -1.23382e-05
(in) SX1: -1.81899e-12
(in) SY1: -2.21834e-39
(in) SPARAM: 0 0 0 0 0
```

	Reference BLAS	BLIS	OpenBLAS
GNU default	INFINITE LOOP		
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2		Undocumented Error Code	NaNs in output

sgbmv

$$\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

EXCVATE found exception handling failures in all tuples due to the handling of **implicit zeros in the banded matrix representation.**

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

sgbmv

$$\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

EXCVATE found exception handling failures in all tuples due to the handling of **implicit zeros in the banded matrix representation.**

```
(in) TRANS: N
(in) M: 2
(in) N: 5
(in) KL: 0
(in) KU: 0
(in) ALPHA: 1
(in) LDA: 2
(in) INCX: -1
(in) BETA: 0
(in) INCY: 1
(in) A: 4.26326e-14 -1 -1
-1 -1 -1 -1 -1 -1 -1
(in) X: nan -1 -1 -1 0
(in) Y: 0 -1
```

$$\begin{bmatrix} 4.26326 \times 10^{-14} & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \\ -1 \\ -1 \\ NaN \end{bmatrix}$$

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

sgbmv

$$\mathbf{y} := \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$$

EXCVATE found exception handling failures in all tuples due to the handling of **implicit zeros in the banded matrix representation.**

```
(in) TRANS: N
(in) M: 2
(in) N: 5
(in) KL: 0
(in) KU: 0
(in) ALPHA: 1
(in) LDA: 2
(in) INCX: -1
(in) BETA: 0
(in) INCY: 1
(in) A: 4.26326e-14 -1 -1
-1 -1 -1 -1 -1 -1 -1
(in) X: nan -1 -1 -1 0
(in) Y: 0 -1
```

$$\begin{bmatrix} 4.26326 \times 10^{-14} & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \\ -1 \\ -1 \\ NaN \end{bmatrix}$$

	Reference BLAS	BLIS	OpenBLAS
GNU default			
w/ -O3			
w/ -ffast-math			
w/ -O3 -ffast-math			
Intel default			
w/ -O3			
w/ -O3 -fp-model=fast=1			
w/ -O3 -fp-model=fast=2			

```
(out) Y: 0 1
```

We have…

…introduced the problem of testing exception handling

- And why current input-generation tools are a poor fit for the problem

…described a novel approach to this problem

- Targeting binary executables using exception spoofing and constraint solving
- Implemented in the prototype tool EXCVATE

…demonstrated our approach on the BLAS

- Tested across multiple implementations, compilers, compiler optimizations
- Found exception-handling failures in 5/26 functions

Source code and data available at https://github.com/ucd-plse/EXCVATE

UC DAVIS
UNIVERSITY OF CALIFORNIA