ACSAC 2022
December 5-9, 2022 · Austin, Texas, USA

Boosting Neural Networks to Decompile Optimized Binaries

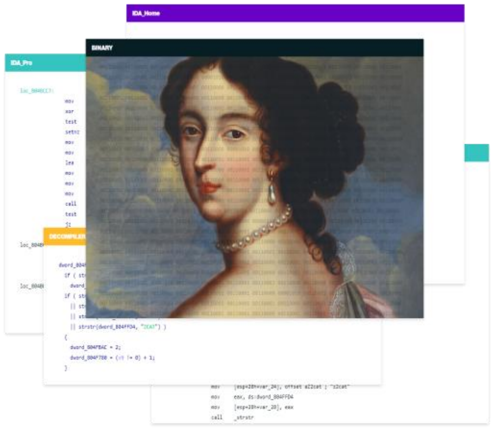
Ying Cao[1,2], Ruigang Liang[1,2], Kai Chen[1,2,3], Peiwei Hu[1,2]
[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences
[2]School of CyberSecurity, University of Chinese Academy of Sciences
[3]Beijing Academy of Artificial Intelligence

December 9, 2022

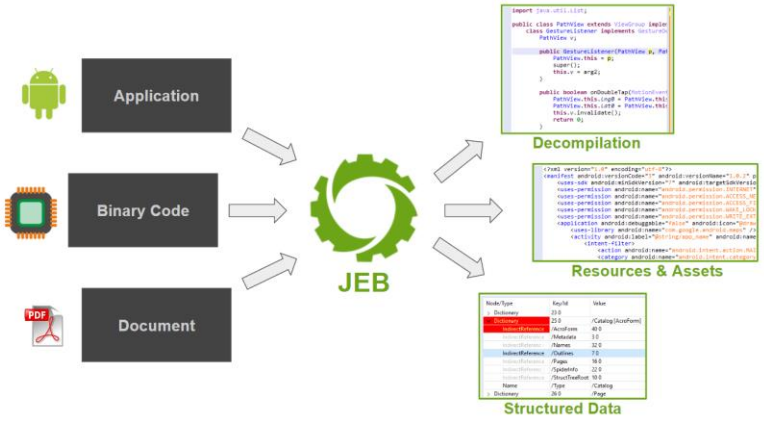
中国科学院大学
University of Chinese Academy of Sciences

中国科学院信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

Popular Decompilers



Hex-Rays



JEB

Ghidra

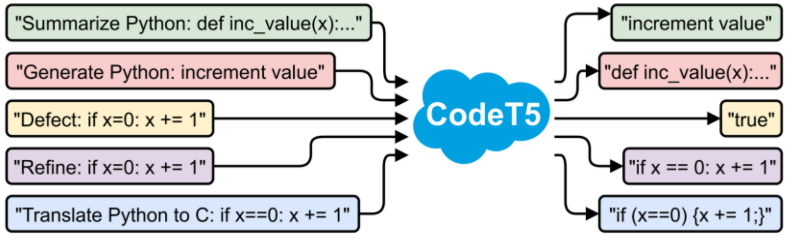
RetDec

These decompilers are all rule-based.

ACSAC 2022
December 5-9, 2022 · Austin, Texas, USA

AI for Code

AI for Source

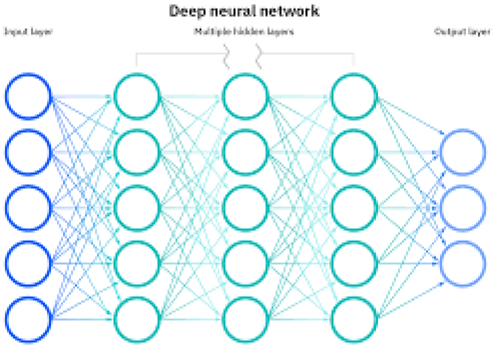


tabnine

GitHub Copilot

AI for Binary

DEEPBITS TECHNOLOGY



DEBIN

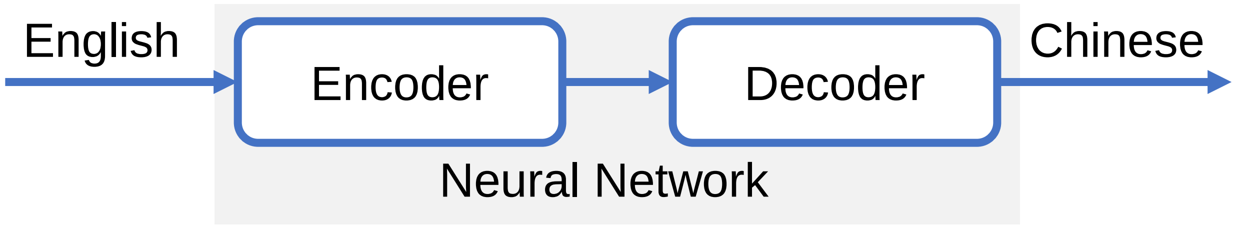
CUMLSec/XDA

This makes it possible to build a neural-based decompiler!

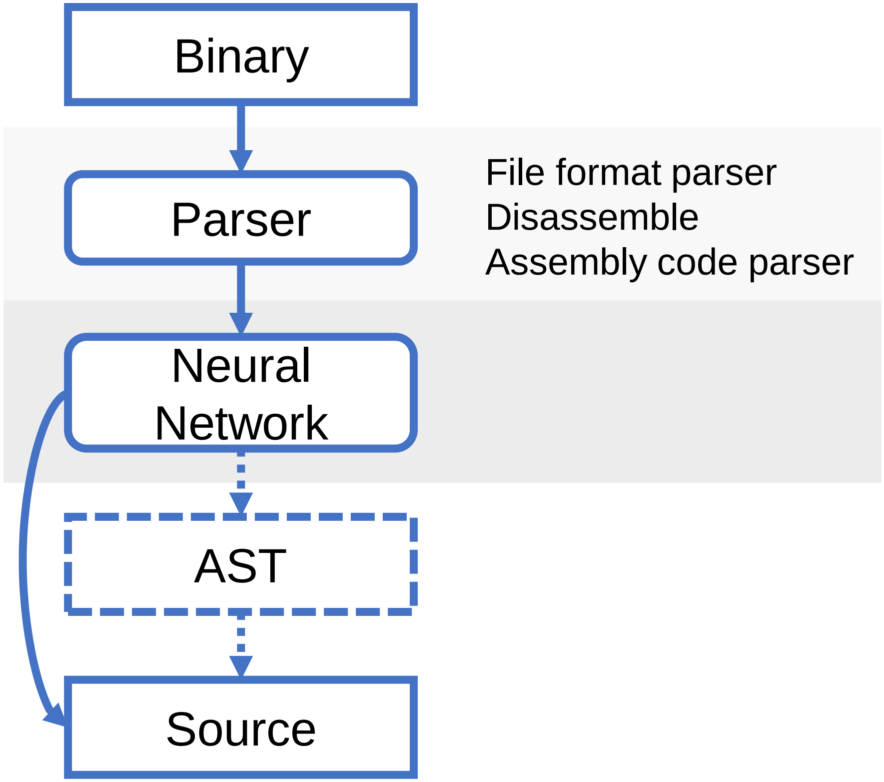
https://www.ibm.com/cloud/learn/neural-networks

Neural-based Decompilers

- Neural Machine Translation



- Existing Work (end-to-end)
  - Using Recurrent Neural Networks for Decompilation (SANER'18)
  - Coda (NeurIPS'19)
  - Neutron (Cybersecurity'21)



Neural-based Decompilers

None of these works can appropriately handle the decompilation of optimized code.

Challenges

C1: Statements of high-level programming language (HPL) are often significantly refactored during compiler optimization.

```c
int func2(int p0)
{
 int var0;
 scanf("%d", &var0);
 int var1 = rand();
 int var2 = -123;
 var1 = var0 + p0 * var1;
 var1 = var1 / var0;
 var1 = var1 / -123;
 return 0;
}
```

?

gcc –O2

```asm
func2(int):
        sub     rsp, 24
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax
        lea     rsi, [rsp+12]
        call    __isoc99_scanf
        call    rand
        xor     eax, eax
        add     rsp, 24
        ret
```

Challenges

C2: Splitting low-level programming language (LPL) and HPL into code fragments with correct correspondence is a nontrivial task.

```
func1(int):
        push    rbx
        xor     eax, eax
        mov     ebx, edi
        mov     edi, OFFSET FLAT:.LC0
    ...
        mov     edi, eax
        call    func1(int)
        add     rsp, 48
        xor     eax, eax
        pop     rbx
        ret
```

Scheme1:
Function/Basic Block

Challenges

C2: Splitting low-level programming language (LPL) and HPL into code fragments with correct correspondence is a nontrivial task.

```
func1(int):
        push    rbx
        xor     eax, eax
        mov     ebx, edi
        mov     edi, OFFSET FLAT:.LC0
        ...
        mov     edi, eax
        call    func1(int)
        add     rsp, 48
        xor     eax, eax
        pop     rbx
        ret
```

heme

Functi e Block

Challenges

C2: Splitting low-level programming language (LPL) and HPL into code fragments with correct correspondence is a nontrivial task.

```
func1(int):
        push    rbx
        xor     eax, eax
        mov     ebx, edi
        mov     edi, OFFSET FLAT:.LC0
        ...
        mov     edi, eax
        call    func1(int)
        add     rsp, 48
        xor     eax, eax
        pop     rbx
        ret
```

~~…heme…~~
~~Functi… …e Block~~

```
 int var2 = -123;
 var1 = var0 + p0 * var1;
 var1 = var1 / var0;
 var1 = var1 / -123;
 return 0;
}
    11c1:     48 8b 44 24 08          mov    0x8(%rsp),%rax
    11c6:     64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
    11cd:     00 00
    11cf:     75 0a                   jne    11db <func2+0x48>
    11d1:     b8 00 00 00 00          mov    $0x0,%eax
    11d6:     48 83 c4 18             add    $0x18,%rsp
    11da:     c3                      ret
    11db:     e8 90 fe ff ff          call   1070 <__stack_chk_fail@plt>
```

?

Scheme2:
Debug Information

Challenges

C2: Splitting low-level programming language (LPL) and HPL into code fragments with correct correspondence is a nontrivial task.

```
func1(int):
        push    rbx
        xor     eax, eax
        mov     ebx, edi
        mov     edi, OFFSET FLAT:.LC0
        ...
        mov     edi, eax
        call    func1(int)
        add     rsp, 48
        xor     eax, eax
        pop     rbx
        ret
```

Functi… Block

```
  int var2 = -123;
  var1 = var0 + p0 * var1;
  var1 = var1 / var0;
  var1 = var1 / -123;
  return 0;
}
    11c1:       48 8b 44 24 08          mov    0x8(%rsp),%rax
    11c6:       64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
    11cd:       00 00
    11cf:       75 0a                   jne    11db <func2+0x48>
    11d1:       b8 00 00 00 00          mov    $0x0,%eax
    11d6:       48 83 c4 18             add    $0x18,%rsp
    11da:       c3                      ret
    11db:       e8 90 fe ff ff          call   1070 <__stack_chk_fail@plt>
```

?

Debu… ation

Challenges

C1: Statements of HPL are often significantly refactored during compiler optimization.

C2: Splitting low-level programming language (LPL) and HPL into code fragments with correct correspondence is a nontrivial task.

It is difficult to build an end-to-end model between HPL and LPL without a well-labeled dataset.

Contribution

- Design a novel neural decompilation technique that can handle compiler-optimized code.
- Design a basic block splitting scheme named *optimal translation unit (OTU).*
- We release our dataset and the NN parameters on GitHub. https://github.com/zijiancogito/neur-dp-data.git

Observation

C1: Statements of HPL are often significantly refactored during compiler optimization.

O1: The structural differences between IR and LPL are much more minor than those between HPL and LPL.

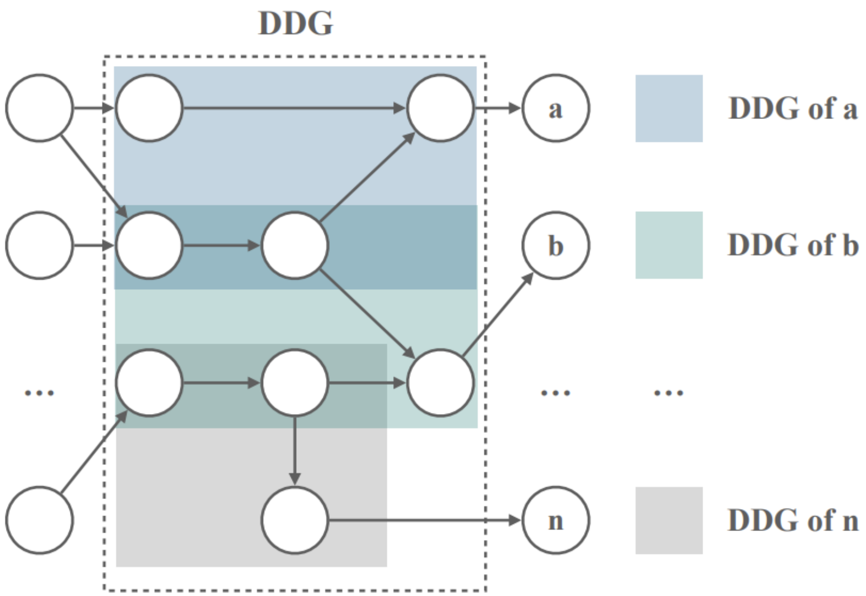
int var0 = f_scanf_nop();	%2 = call i32 @f_scanf_nop()	call 1150 <f_scanf_nop>
int var1 = f_rand();	%3 = call i32 @f_rand()	call 1170 <f_rand>
int var2 = -123;	ret i32 0	xor %eax,%eax
var1 = var0 + p0 * var1;		pop %rcx
var1 = var1 / var0;		ret
var1 = var1 / -123;		
return 0;		
HPL	IR	LPL

Observation

C2: Splitting low-level programming language (LPL) and HPL into code fragments with correct correspondence is a nontrivial task.

O2 :

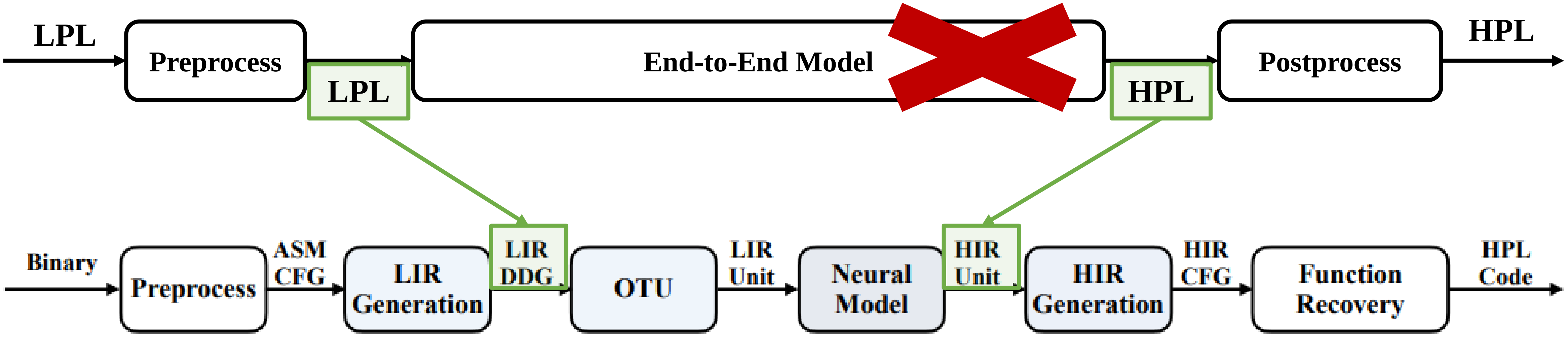
- Most optimization strategies do not change the output of a basic block.
- The output of a basic block usually contains multiple variables whose DDGs often overlap.



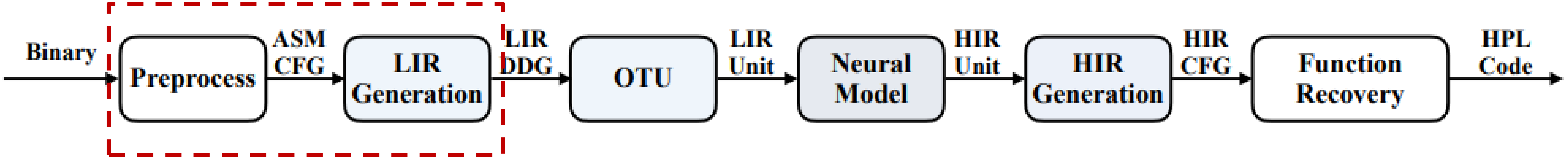
Compilers usually optimize the overlapped and independent parts respectively.

Architecture of NeurDP

NeurDP uses a neural network model to translate LPL into an optimized IR instead of training an end-to-end model for LPL and HPL.



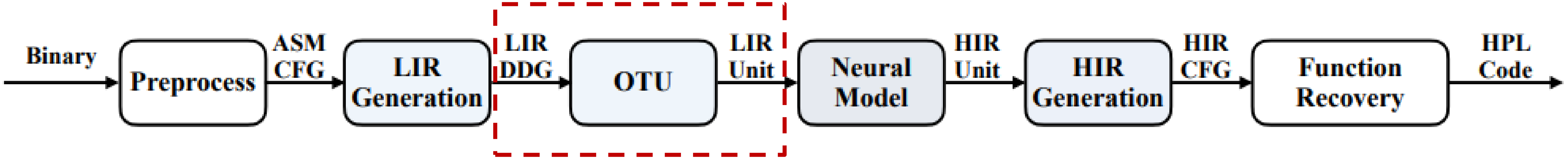
Architecture of NeurDP



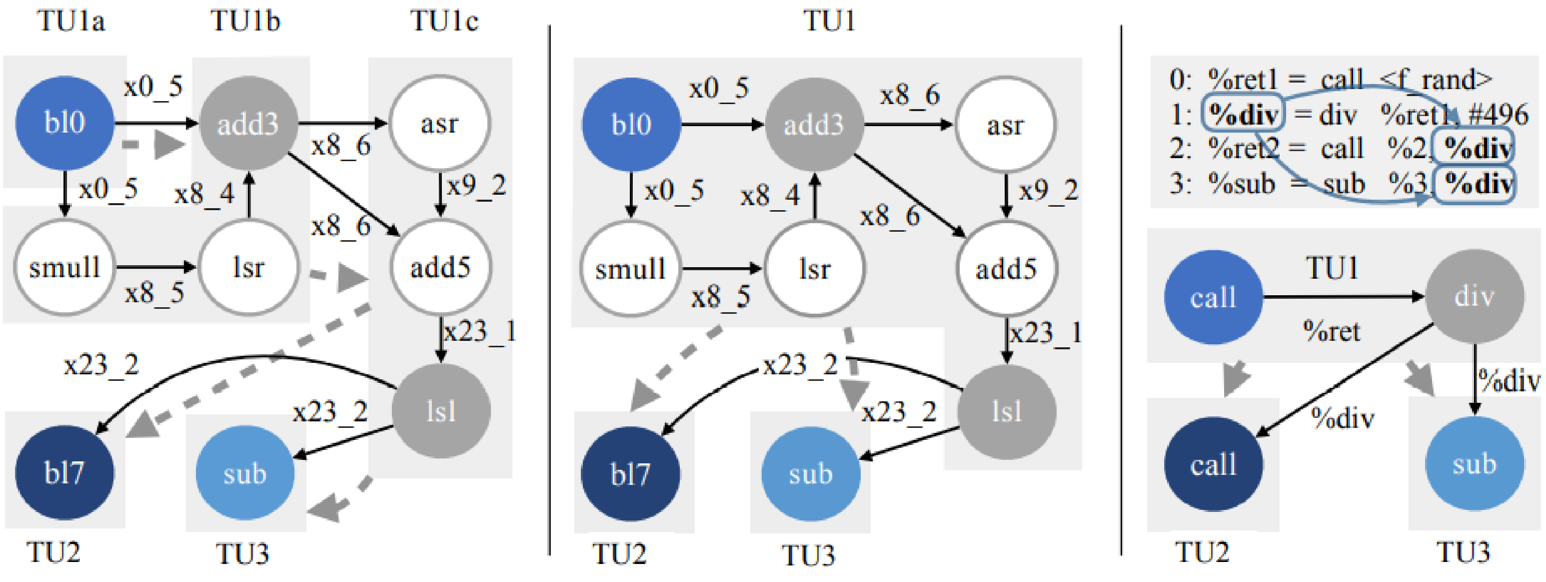
- **Disassemble**
  - Identify the code sections and get all functions.
  - Get the control flow graph (CFG) of each function. (Angr, SP'16)
- **Analysis**
  - To make the model learn better, we perform static single assignment analysis on the LPL and transform it into LIR.
  - Get the DDG of LIR.

ACSAC 2022
December 5-9, 2022 · Austin, Texas, USA

Architecture of NeurDP

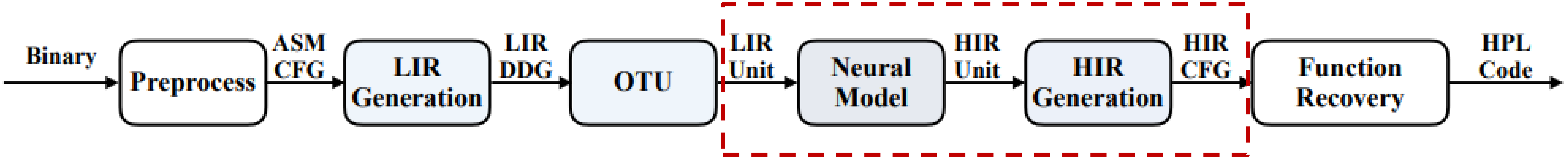


- **OTU**
  - Find the **overlapped** (TU1) and **independent** (TU2 and TU3) regions in DDG (C2)
  - Match the units of LIR and HIR. (HIR is optimized IR)

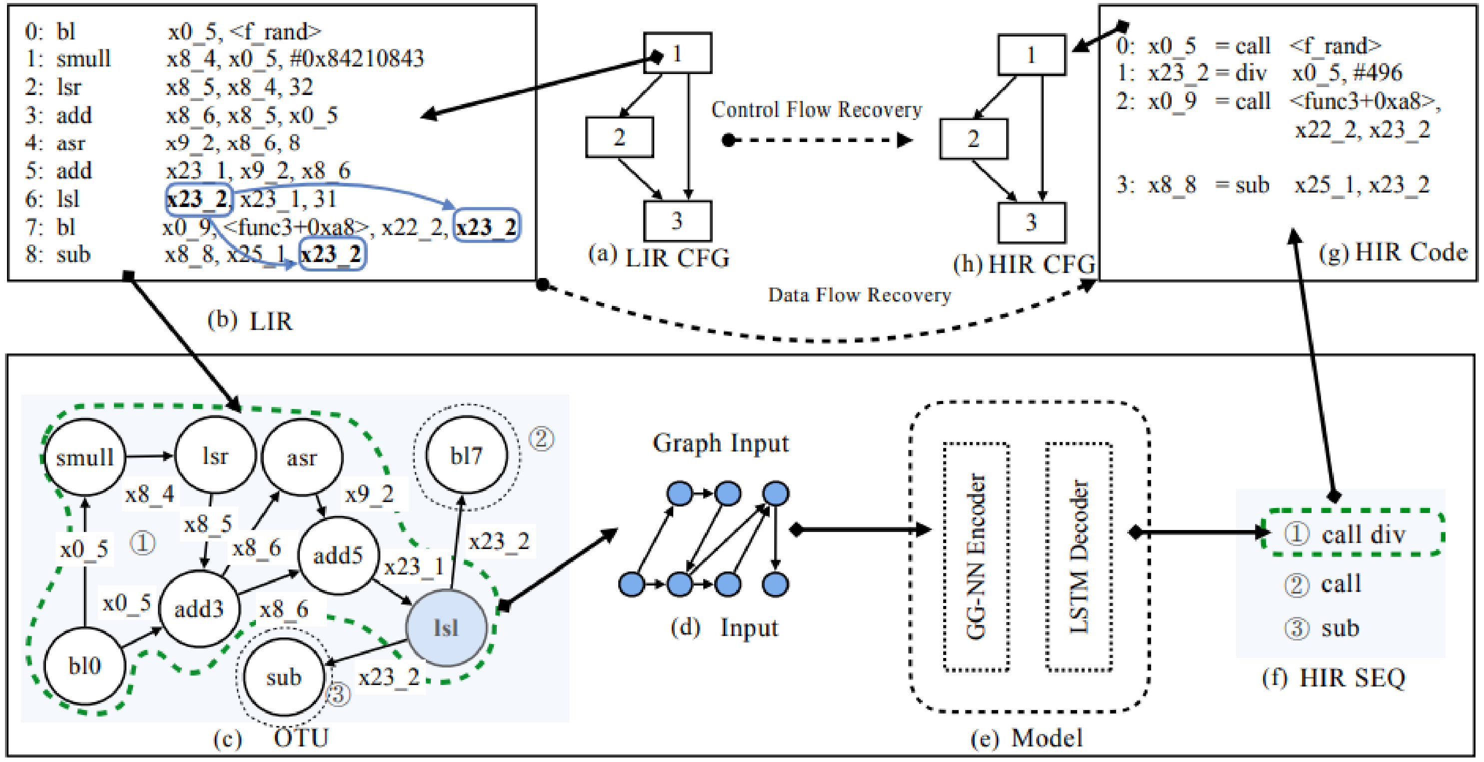


(a) DDG_l (b) DDG_l after combined (c) DDG_h

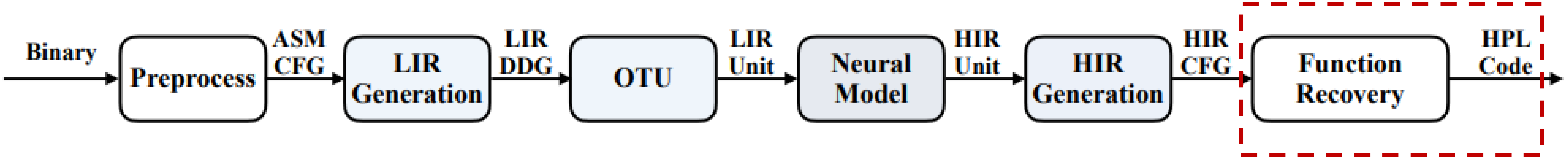
Architecture of NeurDP



- **Model**
  - Input: LIR unit
  - Output: HIR unit
- **HIR Generation**
  - Recover operands of HIR.
- **Build CFG of HIR**



Architecture of NeurDP



- **Function recovery**
  - Translate HIR statements to HPL statements by rules.
  - Control structure recovery (DREAM, NDSS'15)
  - Function signatures recovery (EKLAVYA, Usenix'17)

Finally, we can get a complete HPL function.

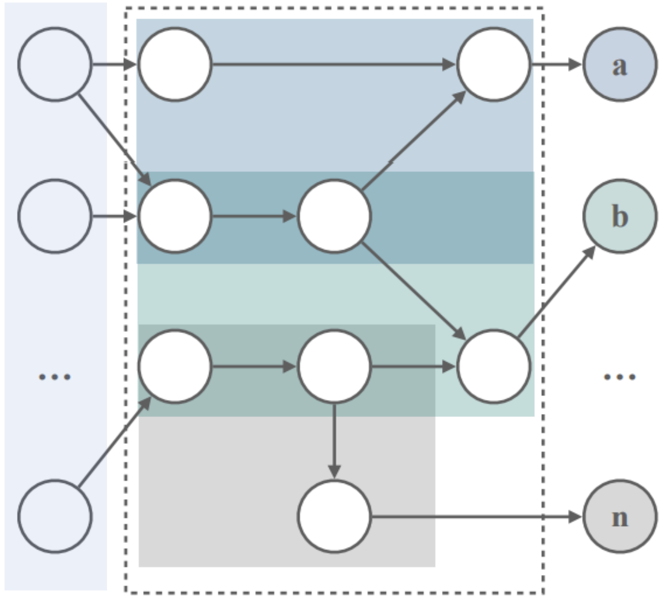
Evaluation

- Training:
  - **Dataset:** 20,000 functions consisting of arithmetic and calling statements
  - **Compiler:** clang10.0 with optimization levels from O0 to O3
  - **OUT units:** 242,000 pairs, 220,000 for training and 22,000 for validation.
- Evaluating:
  - **Dataset:** 1,000 source files, 4,000 ELFs (O0-O3), 20,000 functions

Accuracy

- Metric

IN_B Basic Block



$$OUT_B = F(IN_B),$$
$$\text{where } F = \{f_a, f_b, \dots, f_n\}$$

$$OUT_a = f_a(IN_B)^1$$

$$OUT_b = f_b(IN_B)$$

$$OUT_n = f_n(IN_B)$$

$$Acc_B = \frac{Count_{correct}(f_i^{HPL})}{Count(f_i^{LPL})}, \text{where}$$

$$f_i^{HPL} \text{ is correct if } f_i^{HPL} = f_i^{LPL}$$

Program Accuracy:

$$Acc = \sum_{k=1}^{N} Acc_{B(B_i)}/N,$$
$$\text{where } N = Count(B)$$

[1] f_i is the **symbolic expression** of variable i

Accuracy

- Accuracy of NeurDP
  - This table shows the accuracy of **NeurDP** for O0-O3 optimization levels.
  - Our method has good robustness to the stripped binaries.

Strip option	Compiler optimization level			
	O0	O1	O2	O3
no	0.95	1	0.9	0.9
debug	0.95	0.92	0.9	0.9
all	0.95	0.92	0.9	0.9

no: remain all symbolic information.
debug: strip debug information.
all: strip all symbolic information.

Accuracy

- Compare with SOTA
  - Coda and Neutron cannot cope with the optimized code very well.

	Compiler optimization level			
	O0	O1	O2	O3
Neutron	87.78%	35.45%	32.79%	32.81%
Coda	67.2%-89.2%*	-	-	-
RetDec	29%	25%	4%	-
NeurDP	**95%**	**94.67%**	**90%**	**90%**

*Program accuracy 67.2%-89.2% of Coda is from Table 2 in [16] .

Accuracy

- Analysis of Rule-based Decompiler (RetDec)
  - RetDec does not perform as well as neural-based decompilers.
  - Many errors are caused by the fact that the development of some features is not yet complete.

Building a rule-based decompiler is not easy!

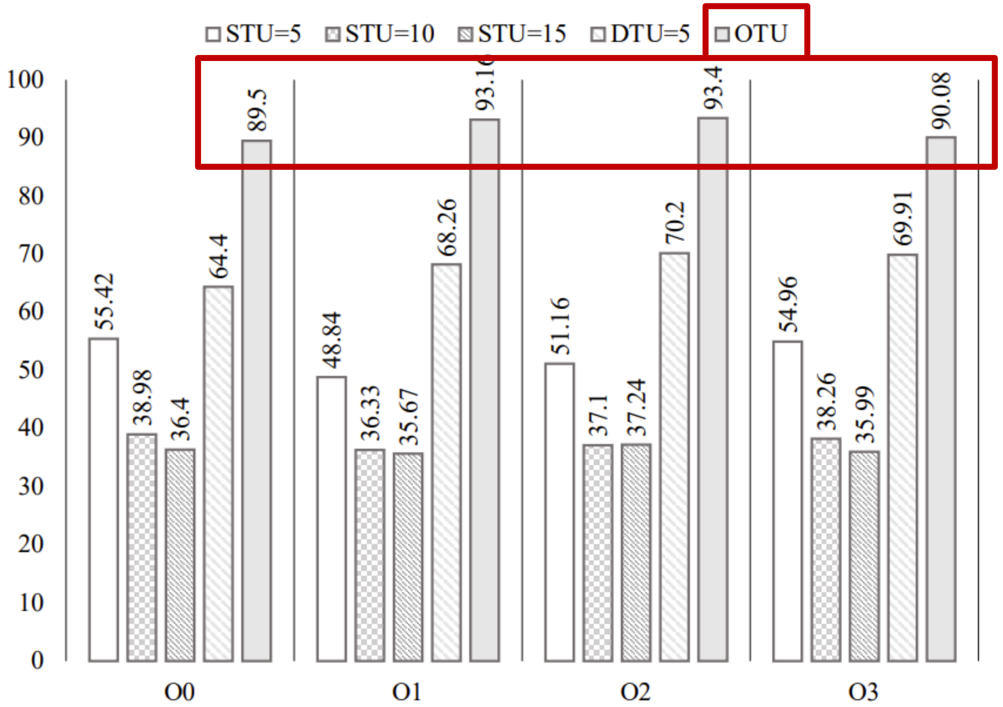
Accuracy

- Compare with Different Neural Networks
  - The results show that using HIR as the translation target of the model is highly generalizable for optimized code.

	Compiler optimization level			
	O0	O1	O2	O3
Transformer-SRC	52.93%	38.22%	39.84%	33.37%
Transformer-AST	75.79%	45.62%	44.60%	32.98%
Transformer-IR	77.18%	75.66%	74.49%	73.94%
LSTM-IR	85.40%	86.61%	85.63%	85.95%
GRU-IR	26.56%	21.93%	25.85%	24.19%
NeurDP	**89.50%**	**93.16%**	**93.40%**	**90.08%**

Accuracy

- Impact under Different Translation Unit
  - **OTU** can maximize the automation of obtaining LIR and HIR pairs with the correct correspondence for training.



Summary

- Conclusion https://github.com/zijiancogito/neur-dp-data.git
  - We propose and implement a neural decompilation framework named NeurDP. **We design a splitting scheme OTU suitable to form a dataset for building a better model.** The evaluation results show that NeurDP achieves better accuracy for optimized code than SOTA neural decompilers.
- Limitation & Future Work
  - Building NeurDP still needs much development effort. We plan to include more types of statements. Explore techniques for improving the decompilation quality of NeurDP.

Thank you!

Q&A