BTC: Beyond the C
Retargetable Decompilation using Neural Machine Translation

Iman Hosseini, Brendan Dolan-Gavitt

NYU

Decompilation

Why?

To analyze malware

Patch vulnerable software

Audit closed-source libraries

Previously:

- Lift native code to some Intermediate Representation
    - This IR is decoupled from the underlying architecture now
- Apply data-flow and type analysis
- Reason about types, construct control flow graph
- And finally: construct high level code from the CFG

Magic ingredient is C-specific heuristics baked in

Modeled on C

Prone to emitting GOTOs (Spaghetti, anyone?)

Decompilation: Classics

Decompilation: Classics

- Complex, costly (to develop) software
- Based on C
- Not based on the premise of generating human-friendly code
  - Traded with ‘correctness’
- “But I want to reverse this Go or Rust malware” ☹
  - Motivation for **Retargetability**
- These examples have all taken many engineer-years to develop and making new ones for other languages is a large endeavor: not much of the previous C-specific patterns and insights can be reused
  - Rust was released a decade ago -> today it is used in malware

Decompilation

Lets decompile a Go function…

```go
func foo(x int) {
    for i := 0; i < x; i++ {
        fmt.Println("hi")
    }
}
```

compiled unstripped and unoptimized

<main.foo>:

```
mov    %fs:0xfffffffffffffff8,%rcx

cmp    0x10(%rcx),%rsp
jbe    48cf2e <main.foo+0x9e>
sub    $0x60,%rsp
mov    %rbp,0x58(%rsp)
lea    0x58(%rsp),%rbp
xor    %eax,%eax
jmp    48cf1a <main.foo+0x8a>
mov    %rax,0x40(%rsp)
xorps  %xmm0,%xmm0
movups %xmm0,0x48(%rsp)
lea    0x11317(%rip),%rax
mov    %rax,0x48(%rsp)
lea    0x4c0ab(%rip),%rcx
mov    %rcx,0x50(%rsp)
mov    0xd02cf(%rip),%rdx
lea    0x4d6b8(%rip),%rbx
....
```

GHIDRA decompiler

```c
void main.foo(void)

{
  ulong *puVar1;
  long lVar2;
  long in_FS_OFFSET;
  long param_7;

  puVar1 = (ulong *)(*(long *)(in_FS_OFFSET + 0xfffffff8) + 0x10);
  if ((undefined *)*puVar1 <= register0x00000020 &&
      (undefined *)register0x00000020 != (undefined *)*puVar1) {
    lVar2 = 0;
    while (lVar2 < param_7) {
      fmt.Fprintln();
      lVar2 = lVar2 + 1;
    }
    return;
  }
  runtime.morestack_noctxt();
  main.foo();
  return;
}
```

Traditional methods are *bad* at decompiling non-C…

Neural Decompilation

What if we applied NMT methods to decompilation?

- 2018 Katz et. al applied RNNs to this problem
    - Modified Clang
    - Limited to < 88 C source, < 112 binary tokens
    - 200,000 snippet pairs (short snippets -> attribution problem)
- 2019 Fu et. al introduced Coda which adds additional error correcting stage
    - Instruction-type awareness
    - Works with ASTs
    - Still RNN-based

Neural Decompilation

- Prior work is only limited to C
- Datasets limited and lacking (i.e. Coda is tested on Hacker’s Delight loop-free programs, and very narrow synthetic data using ‘math.h’ far from Real-World)
- Not retargetable!

Challenges To Retargetability

Compiler Integration (i.e. Clang pass)

Parsing/Lexing required (asm or source)

Requiring awareness of assembly instruction types/ semantics

Working with Abstract Syntax Trees

What if…

We treated code, both assembly and source, as text?
No more compiler integration
No longer requiring parsing
Goodbye to ASTs!

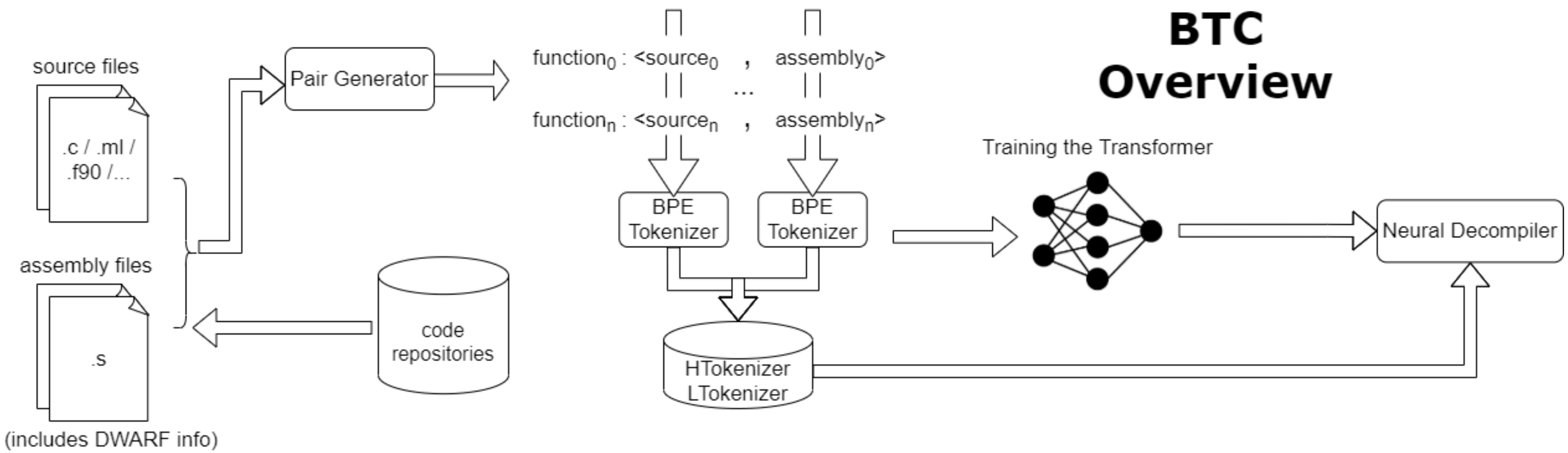
Minimize language and target-specific knowledge & get the most out of the least domain-specific knowledge

NMT, NLP has progressed a lot recently…
Using what NMT has to offer and seeing how far it gets us.

Contributions

- Assessing viability of discarding language-specific knowledge by treating code as plain text
- Demonstrating retargetability by applying our system to 4 programming languages, 3 of which never been the subject of research in decompilation before.
- Provide evaluation of the models
- We will make our training corpora, trained models and code available to enable further research
    - There is little prior examples of such data
    - Including 6 million C functions extracted from 50K+ Debian packages
    - 800K+ Ocaml functions from OPAM repositories
    - A vscode extension to view a decompilation model in action

BTC Architecture



The 4 Languages

C – ~50 years old and still going, close to HW

Go – relatively new language, recently used in ransomware (among other uses)

OCaml – functional, used in various fields from finance (Jane Street) to automated analysis (FB Infer)

Fortran – Old with roots in scientific software, widely used in HPC now and in the past (lots of legacy)

Data Acquisition

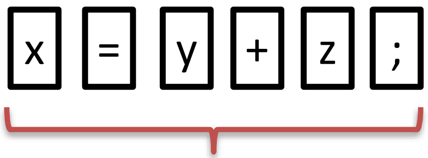
snippets of tokenized code

Katz et. al: programs that "compile with minimal modification" from an assortment of Fedora projects [recall max-seq-len of buckets: 5, 9, 17, 88]

snippets of parsed code [graph]

Coda: synthetic sets composed of math expressions, argument less function calls and math epressions + function calls (only from math.h) and a "real" set which includes Hackers Delight loop-free programs

Also used in other prior work: Euler project/ programming competition programs



6 token in Katz's model
Won't fit in 1st bucket (limited to maximum of 5)

Data Acquisition

100% non-GMO dataset

Each of our samples is a **complete** function

C [10K]: competitive programming (Euler project etc.), interview questions (leetcode)

Go [10K]: competitive programming (Euler project etc.), interview questions (leetcode) and implementations of general algorithms and utility programs

Fortran [10K]: computational code (Couette flow solver, ODE solvers etc.) which is the main use of Fortran in the wild

OCaml [30K]: used packages from OPAM package manager and picked a set of core packages and BAP [a binary analysis tool] with all it’s dependencies

How does it scale if we had more data?

So we gathered larger, MUCH larger corpora for C and OCaml…

Tokenization

BPE: Byte Level Encoding, subword-based
Middle-ground between word-based and character-based
Found to be more suitable than character-level
Generic solution used in NMT: oblivious to language/ any specifics of the domain
Used Huggingface implementation

Model Implementation

Fairseq: Facebook AI implementation of Seq2Seq models [including Transformer]

HARDWARE

Trained over NYU Greene
| Nvidia V100 GPU, 32 GB VRAM | Intel Xeon Platinum 8268 CPU | 369 GB RAM |
C-Data collection [6M dataset] over MESS lab server
| 2x AMD EPYC 7542 | 512 GB |

Evaluation

Average edit distance: terminology same as in Katz, find edit-distance between prediction and ground truth, normalize it by length and average over samples. Asks the question:
a human auditor would take a decompilation, and fix it to get the ground truth, what is the % of change he needs to do?

BTC results

TABLE III: Summary of evaluation results for C, Go, Fortran and OCaml

language	avg. edit dist.	training time (s)	translation speed (function/s)	final loss	number of model params
C	0.60	6165	17.8	2.030	53,481,472
Go	0.74	11024	14.74	2.438	55,517,184
Fortran	0.63	9523	13.4	1.207	54,255,616
OCaml	0.77	10041	27.7	2.540	76,730,368

Evaluation

No prior result has been established on Go, OCaml or Fortran

Let's compare with Katz et. al results on C

Katz et. al

	avg. edit dist.
C Source	0.70
Bucket 0	0.65
Bucket 1	0.67
Bucket 2	0.72
Bucket 3	0.75

Max. Bin. Len.	Max. C Source Len.
11	5
22	9
47	17
112	88

BTC

language	avg. edit dist.
C	0.60
Go	0.74
Fortran	0.63
OCaml	0.77

Language	Max Source Length	Max Asm Length
C	271	1776
Go	254	6350
Fortran	398	5408
Ocaml	192	5939

GROUND TRUTH

Fortran

```
subroutine en_her_02_xiu_size ( n , o
 ↪  )
 implicit none
 integer ( kind = 4 ) n
 integer ( kind = 4 ) o
 o = n + 1
 return
end
```

OCaml

```
let fmt_path f x = fprintf f " STR "
 ↪  fmt_path_aux x ;;
```

Go

```
sum := 0
for _ , val := range arr {
sum += val
}
return sum
```

C

```
char* p = "STR";
while ( scanf ("STR", &a )== 1 && a )
 ↪  puts ( p );
return 0;
```

PREDICTION

Fortran

```
subroutine i4_determinant ( n , value
 ↪  )
 implicit none
 integer ( kind = 4 ) n
 integer ( kind = 4 ) value
 value = n * n - 1
 return
end
```

OCaml

```
let pp ppf x = Format . fprintf ppf "
 ↪  STR " ( to_string x )
```

Go

```
sum := 0
for _ , v := range nums {
sum += v
}
return sum
```

C

```
while (scanf ("STR", &a) != EOF) if (a
 ↪  == 0)
printf ("STR");
else
printf ("STR");
return 0;
```

Complementation rather than competition

ND research is far from production-grade decompilers

For C: good traditional decompilers that *are* production-ready

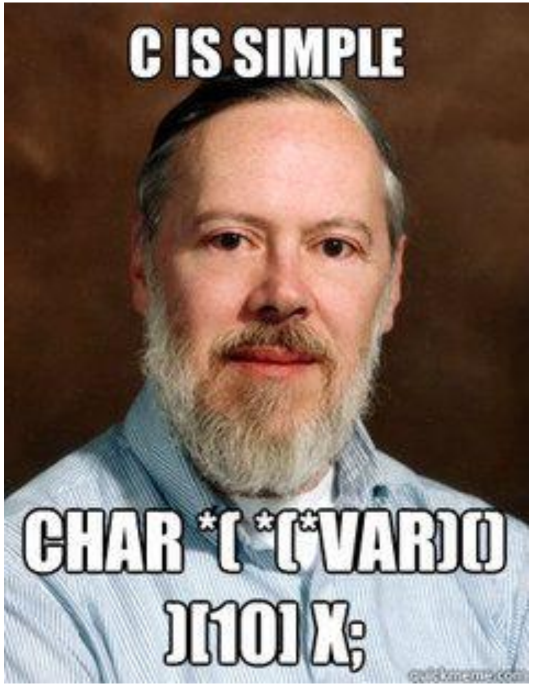
ND can't compete with them in C

But if we go *Beyond-The-C* it's different

Our approach complements Traditional Decompilers

As a tool beside them for reversing non-C languages

Limitations



ND still has a long way to go

- Interpretability
- Provable semantic correctness
- Long sequence length
    - A general limitation of NMT
    - Will get better with advances in ML
- Little open-sourced work/data, no demonstration of a retargetable approach (until now)
    - We’re sharing our data/code

Future Work

- BTC's main advantage is simplicity
- This approach can be used in conjunction with other methods
  - With Coda or Evolving Exact Decompilation as Sketch gen
  - Anonymization, canonicalization other techniques
- Interactive UX

Conclusion

- ND is promising
  - With the right design choices
  - And retargetability
- Little domain knowledge *can* go a long way
- There is much to explore, if we go Beyond-The-C

[THE END]