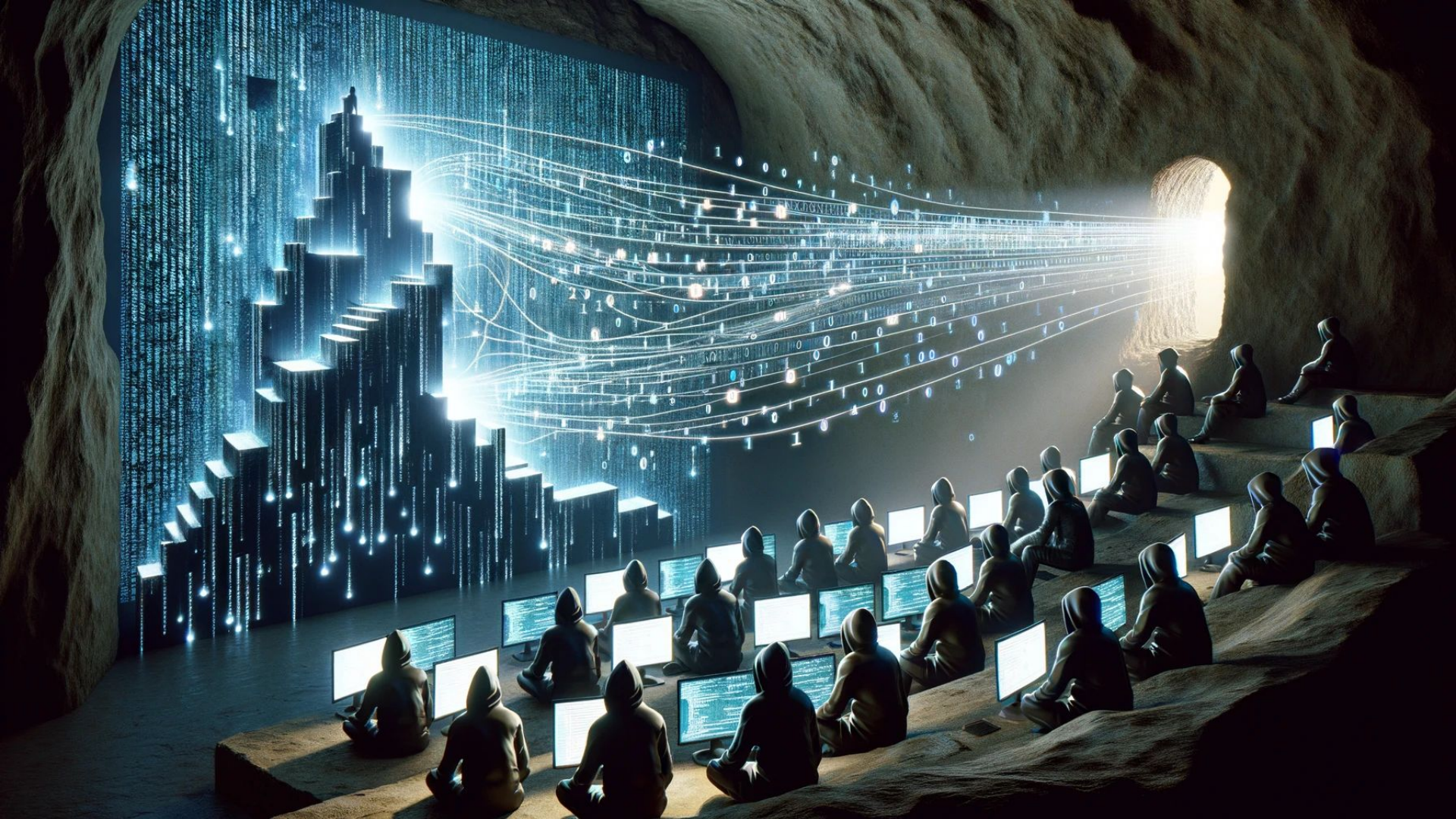
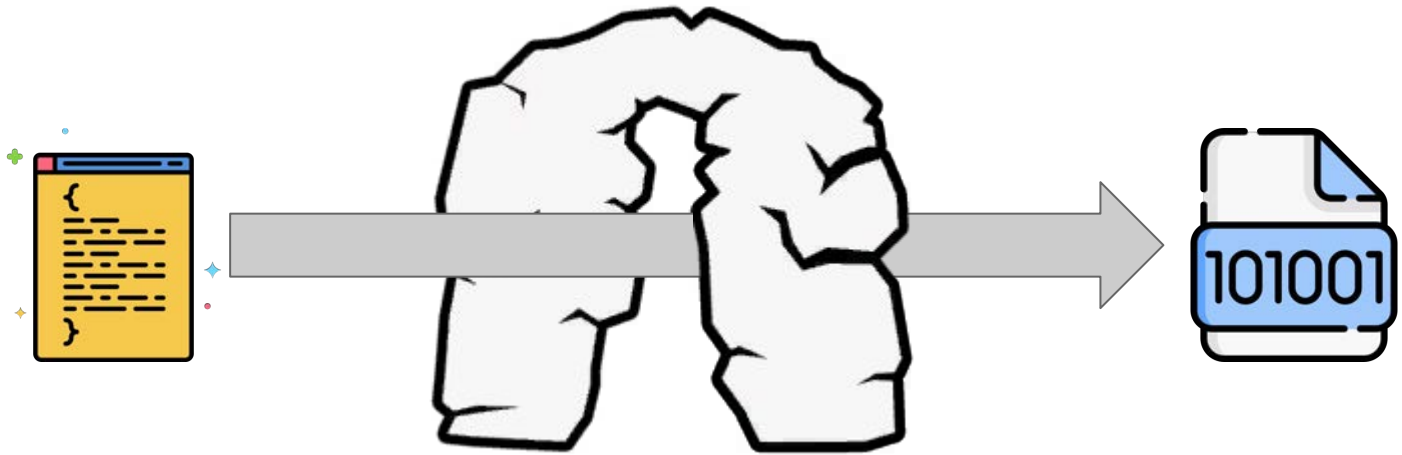
NSA SoS
ASU Kickoff

Yan Shoshitaishvili and Adam Doupé
January 11, 2024

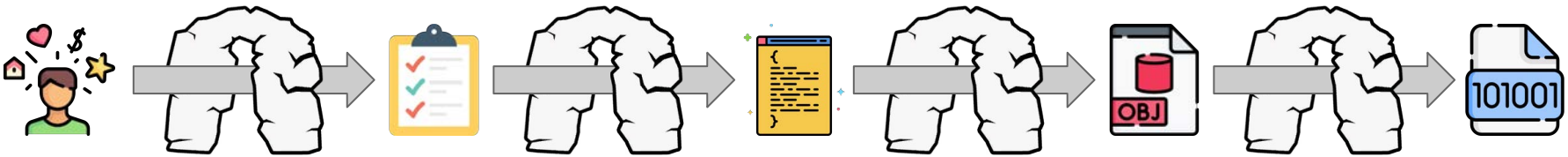
Leveraging Machine Learning for **Binary Software Understanding**



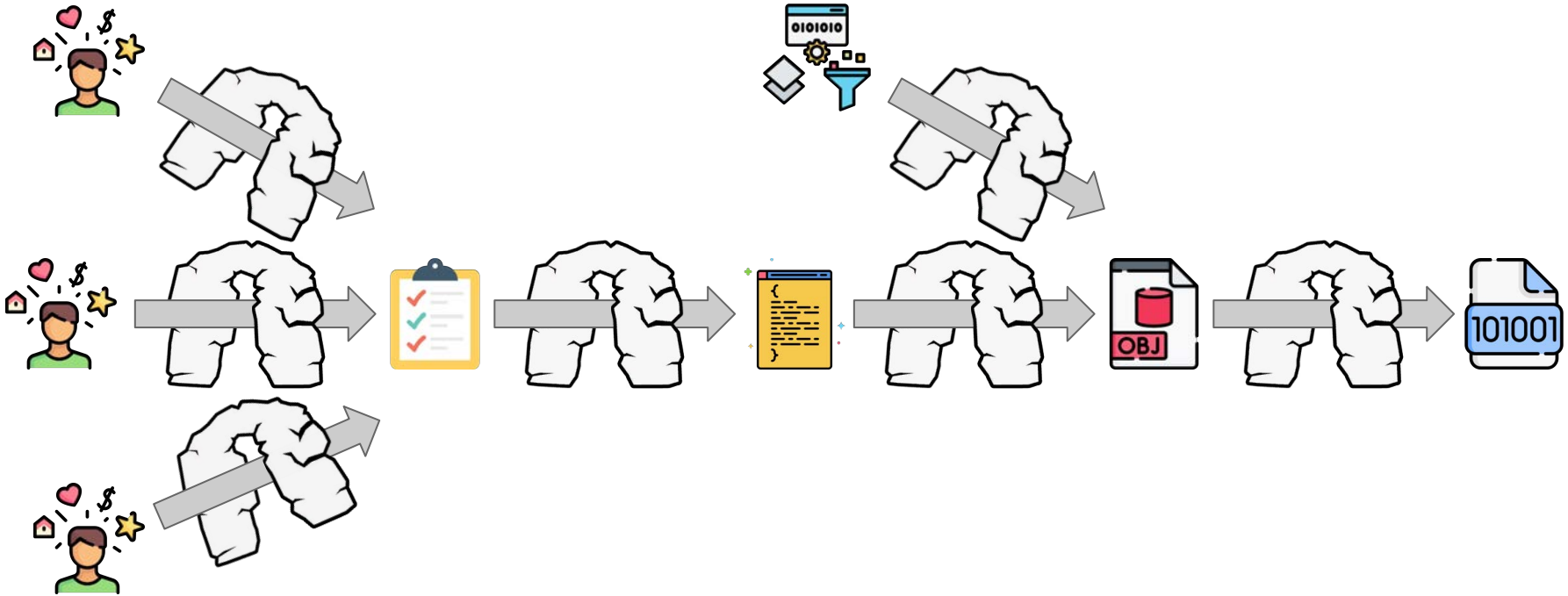
Prevailing Wisdom



Reality?



Reality?

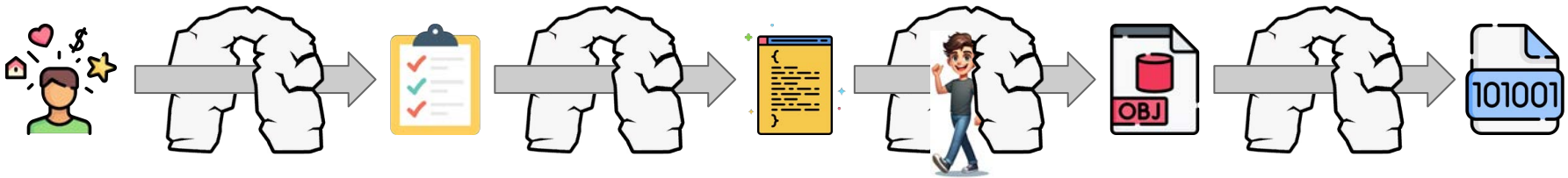


Our intuition: different types of lost information necessitate different information reconstruction approaches!

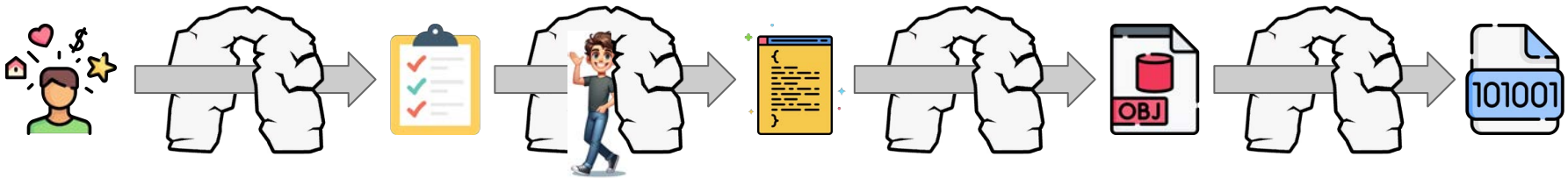
Leveraging Machine Learning for **Binary Software Understanding**

Our mascot, AdamDe ("De" is short for Decompiler)

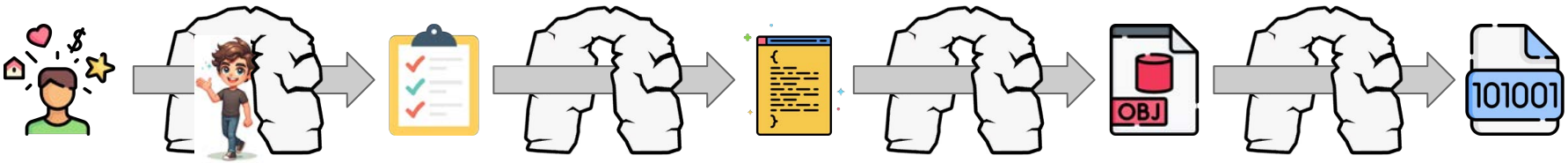
ASU Task 1



ASU Task 2



ASU Task 3



ASU Task 1

Achieving Semantically-Equivalent Decompilation

Fair question: "Do we really need ML?"

The Structuring Problem

```
// Source
void foo(int a, int b, int c)
{
  if (a && b) {
      puts("first print");
  }

  puts("second print");

  if (b || c) {
      puts("third print");
  }
}
```

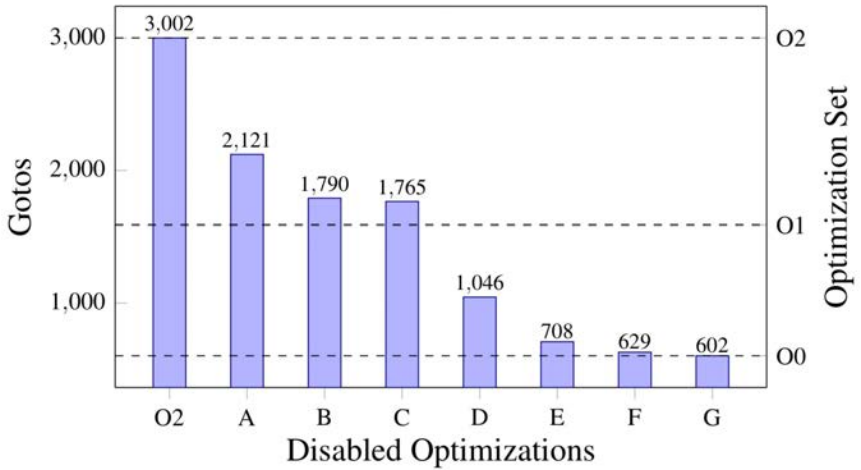

```
// HexRays 7.7
void foo(int a, int b, int c)
{
  if (a && b) {
      puts("first print");
      puts("second print");
      goto third_print;
  }

  puts("second print");

  if (b || c) {
      third_print:
      puts("third print");
  }
}
```

Intuition: Bad (Spurious) GOTOs Have a Root Cause!

GOTOs are introduced by optimizations… WHICH ONES?

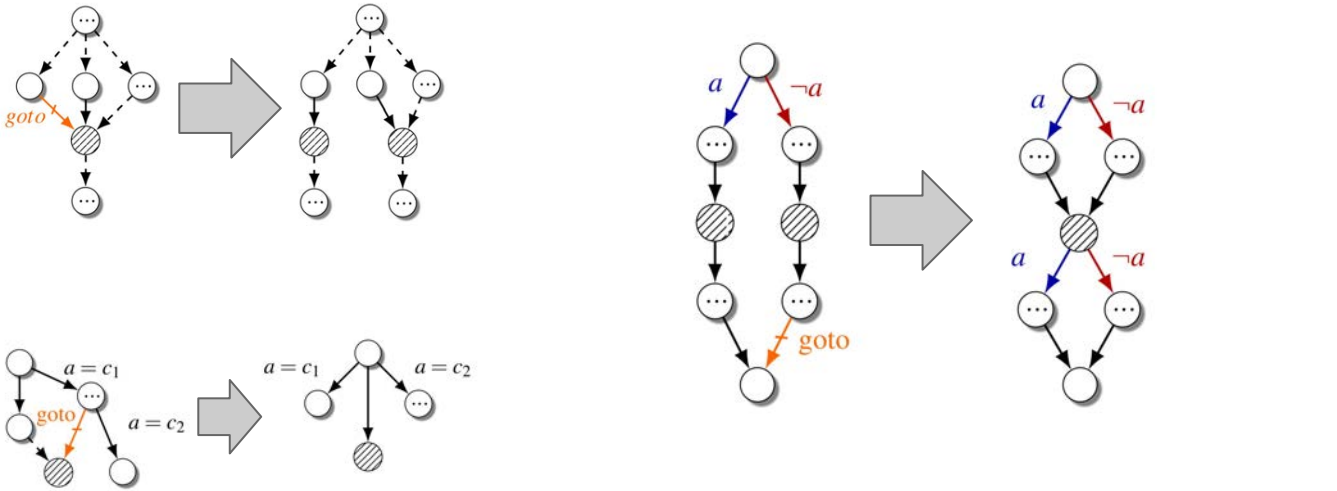
We actually looked at the compilers, and found 9 types:

A. Jump Threading
B. Common Subexpression Elimination
C. Switch Conversion
D. Cross Jumping
E. Software Thread Cache Reordering
F. Loop Header Optimization
G. Builtin Inlining
H. Switch Lowering
I. Nonreturning Functions Transformations



Solution: Precise De-optimization!

SAILR works through *iterative deoptimization* to remove only the spurious GOTOs...



```c
int schedule_job(int needs_next, int fast_job, int mode)
{
    if (needs_next && fast_job) {
        complete_job();
        if (mode == EARLY_EXIT)
            goto cleanup;

        next_job();
    }

    refresh_jobs();
    if (fast_job)
        fast_unlock();

cleanup:
    complete_job();
    log_workers();
    return job_status(fast_job);
}
```

Original (derived from Linux Kernel)

```c
long long schedule_job(unsigned int a0,
    unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT == a2)
            goto LABEL_4012eb;
        next_job();
    }
    refresh_jobs();


    if (a1)
        fast_unlock();


LABEL_4012eb:
    complete_job();
    log_workers();
    return job_status(a1);
}
```

SAILR

Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation

Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao,
Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, Ruoyu Wang
Arizona State University
{zbasque,atipriya,wfgibbs,judeo,derronm,tbao,doupe,yans,fishw}@asu.edu

Abstract

Contrary to prevailing wisdom, we argue that the measure of binary decompiler success is not to eliminate all gotos or reduce the complexity of the decompiled code but to get as close as possible to the original source code. Many gotos exist in the original source code (the Linux kernel version 6.1 contains 3,754) and, therefore, should be preserved during decompilation, and only *spurious gotos* should be removed.

Fundamentally, decompilers insert spurious gotos in decompilation because structuring algorithms fail to recover C-style structures from binary code. Through a quantitative study, we find that the root cause of spurious gotos is compiler-induced optimizations that occur at all optimization levels (17% in non-optimized compilation). Therefore, we believe that to achieve high-quality decompilation, decompilers must be *compiler-aware* to mirror (and remove) the goto-inducing optimizations.

In this paper, we present a novel structuring algorithm called SAILR that mirrors the compilation pipeline of GCC

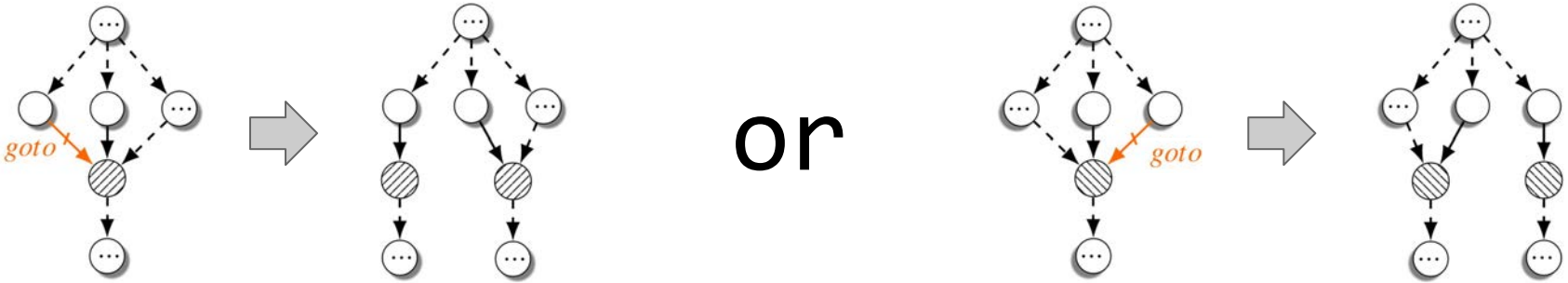
malware analysis [19, 20, 33, 43], and vulnerability discovery and mitigation [32, 36].

Many applications of decompilation require *high-quality* decompiled source code. One important criterion of high-quality source code is meaningful control flow structures, such as `if-else` statements, `while` loops, and `do-while` loops in the C language. Unfortunately, the semantics of these control flow structures are lost during compilation, replaced by simple binary-level control flow transfers such as `jmp` instructions.

Decompilers leverage control flow *structuring* algorithms that analyze low-level constructs in the compiled binary and attempt to recover the high-level control flow. If compilers simply translated C code to assembly and pieced the assembly together with `jmp` instructions, the resulting code would be easily *structurable*, and decompilers would be able to produce high-quality decompiled source code. However, modern compilers optimize and distort code structures during compilation, making the result *unstructurable* and preventing current structuring algorithms from recovering the high-level control flow structure.

ML Opportunity #1

SAILR is *iterative*... but what if there are multiple options?



Making the wrong choice in this iteration impacts *later* interactions...
... same as chess or go?
... **alphaSAIL**!

Next Steps #2

SAILR is great... but IDA outperforms us on some functions.

Turns out IDA has a **LOT** of special-cased decompilation "fixups".

```
bool __fastcall read_char(int *c)
{
  char v1; // r12
  FILE *i; // rdi
  int *v4; // rax
  bool v5; // bl
  bool file; // al
  int v7; // eax

  v1 = 1;
  *c = -1;
  for ( i = in_stream; in_stream; v1 &= file && v5 )
  {
    v7 = fgetc(i);
    *c = v7;
    if ( v7 != -1 )
      break;
    v4 = _errno_location();
    v5 = check_and_close(*v4);
    file = open_next_file();
    i = in_stream;
  }
  return v1;
}
```

```
int read_char(unsigned int *a0)
{
    unsigned int v1;  // r12d
    unsigned int *v2;  // rdi
    unsigned long long v3;  // rax

    v1 = 1;
    *(a0) = -1;
    v2 = in_stream;
    if (in_stream)
    {
        while (true)
        {
            (unsigned int)v3 = fgetc(v2);
            *(v2) = v3;
            if ((unsigned int)v3 != -1)
                break;
            v2 = in_stream;
            v1 &= check_and_close(*(__errno_location())) & open_next_file();
            if (!in_stream)
                break;
        }
    }
    return v1;
}
```

Questions abound!

- What are they?
- What is their actual impact?
- Can we analyze them to *learn* generalized techniques?

Promising opportunities for ML-augmented decompilation

First: Variable/Function Names.
Our early ML-driven success: recovering names of variables!

Prior Work

VarBERT is not the first work using ML to predict variable names. Two earlier ones:

DIRE (2019).
Applies a transformer NN on decompiled code (with decompiler-generated variable name placeholders) to recover meaningful variable names.

DIRTY (2022).
Applies a transformer NN on decompiled code (with decompiler-generated variable name placeholders) to recover meaningful variable names.

VarBERT (2023).
Applies a transformer NN on decompiled code (with decompiler-generated variable name placeholders) to recover meaningful variable names.

Understanding the Dataset

Decompilation ML is *very* easy to mess up.

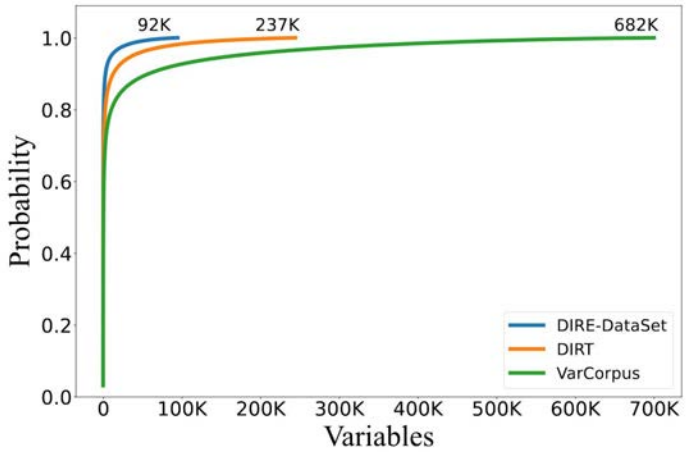
Duplication runs rampant.

Building a proper dataset without duplication is difficult...
For example, the average coreutils binary shares >60% of its code with every other coreutils binary!

Ground truth is elusive.

The most common variable name in DIRTY's ground truth is **v1**.
This is because decompilers, used to generate the ground truth dataset, *insert spurious variables*!

	Functions	DIRE	DIRTY
Overall	Total	1,214,930	1,872,420
	Duplicates	817,298 (67.2%)	1,015,768 (54.2%)
	Deduplicated	397,632	856,652
Train	Total	1,011,054	1,668,544
	Duplicates	683,814 (67.6%)	950,990 (56.9%)
	Deduplicated	327,240	717,554
Test	Total	124,179	203,876
	Duplicates	53,787 (43.3%)	64,778 (31.7%)
	Deduplicated	70,392	139,098
Overlap	Total	99,317	133531
	De-duplicated	46,316	71,967



Our Approach

A Two-Step Training Process.

VarBERT first *pretrains* to reason about *human-written* source code, and then *finetunes* for two applications:

1. Differentiating *spurious* from *human-written* variables in decompiled code.
2. Recovering human-written names for human-written variables.

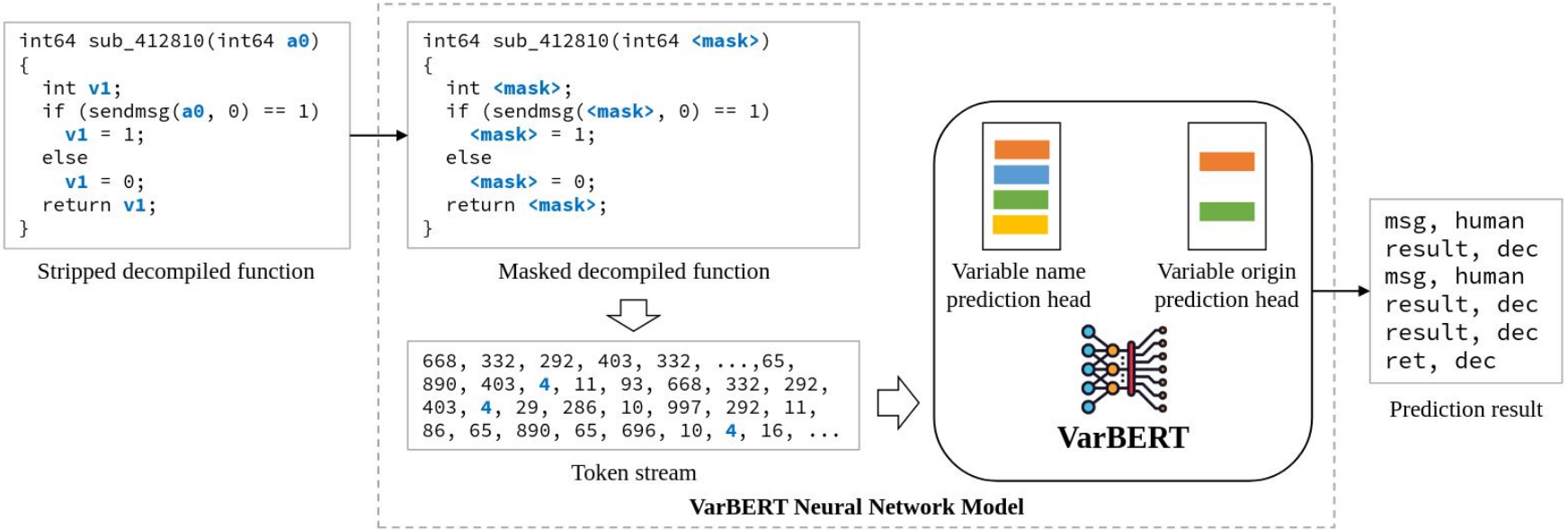
New Datasets!

VarBERT ships with datasets to address limitations of prior work:

1. Human Source Code dataset of >5M functions to support pretraining.
2. Large, aggressively deduplicated (at the function level) datasets of many binaries at different optimization levels.

Data Set	C.O.	Unique Variables	Functions	Binaries
HSC	N/A	3,561,537	5,235,792	N/A
VarCorpus (IDA)	O0	682,461	2,657,046	26,280
	O1	234,417	722,942	16,815
	O2	201,525	579,606	15,893
	O3	198,297	578,156	15,427
VarCorpus (Ghidra)	O0	521,668	2,066,871	20,433
	O1	179,016	856,608	16,647
	O2	218,633	763,053	17,939
	O3	193,712	628,384	13,770
DIRE-DataSet [36, RQ4]*	O0	92,082	1,259,935	164,632
DIRE-DataSet-Dedup	O0	92,082	463,238	N/A
DIRT [10, Table 11]*	O0	237,928	2,075,762	75,656
DIRT-Dedup	O0	237,928	995,418	N/A

And, finally, a transformer-based variable name recovery!



Success!

Model	Split	Top-1 Accuracy
VARBERT	Function	50.70
	Binary	37.17
DIRTY	Function	38.00[6]
	Binary	32.65[6]
DIRE	Function	35.94

Success, even in failure...

	Correct Predictions	Top-3 Incorrect Predictions		
buffer	buffer (63.55%)	buf (8.95%)	UNK (5.35%)	data (2.43%)
len	len (73.57%)	UNK (5.65%)	size (2.80%)	n (1.84%)
tmp1	tmp1 (48.56%)	tmp2 (12.83%)	tmp0 (11.55%)	UNK (5.56%)
srcsize	srcsize (42.86%)	len (28.57%)	length (14.29%)	size (7.14%)
substring_n	substring_n (50.00%)	substring1 (25.00%)	substring (25.00%)	-

"Len or index or count, anything but v1": Predicting Variable Names in Decompilation Output with Transfer Learning

Kuntal Kumar Pal*, Ati Priya Bajaj*, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura,
Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran,
Yan Shoshitaishvili, Adam Doupé, Chitta Baral, Ruoyu Wang
Arizona State University
{*kkpal, atipriya, pbanerj6, dutcher, mutsumi, zbasque, hgupta35,*
ssawan13, uananthe, yans, doupe, chitta, fishw}*@asu.edu*

***Abstract*—Binary reverse engineering is an arduous and tedious task performed by skilled and expensive human analysts. Information about the source code is irrevocably lost in the compilation process. While modern decompilers attempt to generate C-style source code from a binary, they cannot recover lost variable names. Prior works have explored machine learning techniques for predicting variable names in decompiled code. However, the state-of-the-art systems, DIRE and DIRTY, generalize poorly to functions in the testing set that are not included in the training set—31.8% for DIRE on DIRTY's data set and 36.9% for DIRTY on DIRTY's data set.**

In this paper, we present VARBERT, a Bidirectional Encoder Representations from Transformers (BERT) to predict meaningful variable names in decompilation output. An advantage of VARBERT is that we can pre-train on human source

(relying only on registers, memory, and branch statements), so the compiled output does not need this information.

This phenomenon of compilation-induced information loss makes it more difficult for human analysts to understand binary programs ("binaries") than to understand source code [61], despite the fact that a compiler-generated binary encodes the same logic as the corresponding source code. To aid humans in such understanding, and support a number of downstream security tasks, researchers have developed a number of *de*compilation techniques, which take as input binary code, recover the lost semantic information from the binary, and derive roughly equivalent source code (or pseudocode, which generally is in an approximate version of C). State-of-the-art decompilers, e.g., IDA Pro's Hex-Rays decompiler [27], Ghidra [13], and Binary Ninja [43], are

Promising opportunities for ML-augmented decompilation

First: Variable/Function Names.
Our early ML-driven success: recovering names of variables!

Next: *Type* Inference with ML.

Challenges in ML Type Inference

VarBERT works because other tokens remain identical during variable renaming.

```
unsigned __int64 qemu_clock_enable(
    __int64 uc, char enabled) {
  char status;
  unsigned __int64 v4;
  v4 = __readfsqword(Number);

  status = *(uc + Number);
  *(uc + Number) = enabled;
  if (enabled && status != Number) {
   qemu_rearm_alarm_timer(alarm_timer);
  }

  return __readfsqword(Number) ^ v4;
}
```

```
unsigned __int64 qemu_clock_enable(
    __int64 clock, char enable) {
  char old;
  unsigned __int64 v4;
  v4 = __readfsqword(Number);

  old = *(clock + Number);
  *(clock + Number) = enable;
  if (enable && old != Number) {
   qemu_rearm_alarm_timer(alarm_timer);
  }

  return __readfsqword(Number) ^ v4;
}
```

This is *not* the same with types!

```
void qemu_clock_enable(
    QEMUClock *clock, bool enabled) {



  bool old = clock->enabled;
  clock->enabled = enabled;
  if (enabled && !old) {
    qemu_clock_notify(clock);
  }

}
```

```
unsigned __int64 qemu_clock_enable(
    __int64 uc, char enabled) {
  char status;
  unsigned __int64 v4;
  v4 = __readfsqword(Number);

  status = *(uc + Number);
  *(uc + Number) = enabled;
  if (enabled && status != Number) {
   qemu_rearm_alarm_timer(alarm_timer);
  }

  return __readfsqword(Number) ^ v4;
}
```

A shifting structure impacts the performance of text-based transformers when training on non-trivial type information (e.g., beyond int/float/char)...

Our Approach: BITYR

We cannot rely on decompiler output (too variable with different types).

But we can use the intermediate analysis results!

BITYR uses statically-recovered read/write patterns of variables (recovered during decompilation) to query a Graph Neural Network for...

1. Is the variable a member of a struct, or a just a primitive type?
2. If the variable is a struct member, what is its type?

Early Promise!

Preliminary results are promising, beating both ML and non-ML techniques.

Note: previous ML techniques can only predict previously-seen types!

Arch./Opt.	Solution	Genre	Precision	Struct Precision
X64/O0	StateFormer	ML-based type **prediction** on binary code	53.9%	Unsupported
	DIRTY	ML-based type **prediction** on decompiled code	55.8%	34.1%
	Osprey	Non-ML Type **inference**	71.8%	29.5%
	Bityr	ML-based type **inference**	82.8%	68.0%

A Note About Open Science!

Thank You!

Yan Shoshitaishvili
yans@asu.edu
https://yancomm.net
@Zardus@defcon.social

Adam Doupé
doupe@asu.edu
@adamd@defcon.social

https://sefcom.asu.edu
https://angr.io
https://pwn.college

```c
int schedule_job(int needs_next, int fast_job, int mode)
{
    if (needs_next && fast_job) {
        complete_job();
        if (mode == EARLY_EXIT)
            goto cleanup;

        next_job();
    }

    refresh_jobs();
    if (fast_job)
        fast_unlock();

cleanup:
    complete_job();
    log_workers();
    return job_status(fast_job);
}
```

Original (derived from Linux Kernel)

```c
long long schedule_job(unsigned int a0,
    unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT != a2)
        {
            next_job();
            refresh_jobs();
        }
    }

    if (!a0 || !a1)
        refresh_jobs();
    if (a1 && (!a0 || EARLY_EXIT != a2))
        fast_unlock();

    complete_job();
    log_workers();
    return job_status(a1);
}
```

DREAM

```c
int schedule_job(int needs_next, int fast_job, int mode)
{
    if (needs_next && fast_job) {
        complete_job();
        if (mode == EARLY_EXIT)
            goto cleanup;

        next_job();
    }

    refresh_jobs();
    if (fast_job)
        fast_unlock();

cleanup:
    complete_job();
    log_workers();
    return job_status(fast_job);
}
```

Original (derived from Linux Kernel)

```c
long long schedule_job(unsigned int a0,
    unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT == a2)
            goto LABEL_4012eb;
        next_job();
        refresh_jobs();
        goto LABEL_4012d3;
    }
    refresh_jobs();
    if (!a1)
        goto LABEL_4012eb;
LABEL_4012d3:
    fast_unlock();
LABEL_4012eb:
    complete_job();
    log_workers();
    return job_status(a1);
}
```

Phoenix

Intuition: There are Good (Developer-Intended) GOTOs

No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations

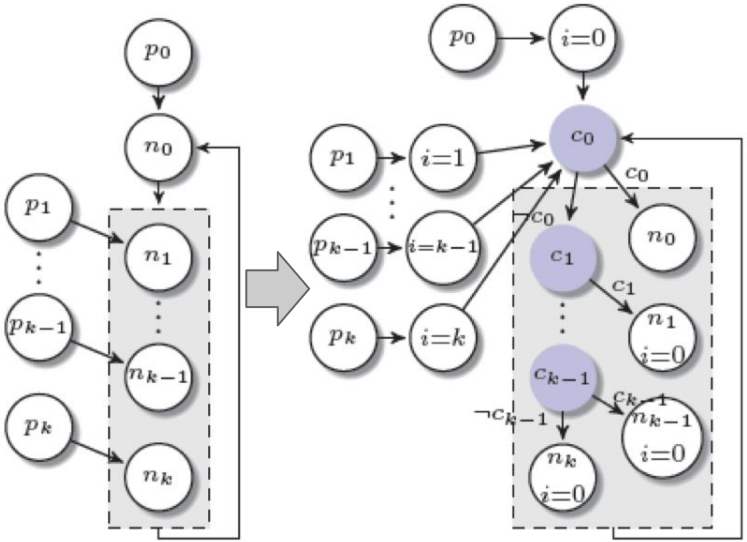
Khaled Yakdan*, Sebastian Eschweiler†, Elmar Gerhards-Padilla†, Matthew Smith*
*University of Bonn, Germany
{yakdan, smith}@cs.uni-bonn.de
†Fraunhofer FKIE, Germany
{sebastian.eschweiler, elmar.gerhards-padilla}@fkie.fraunhofer.de

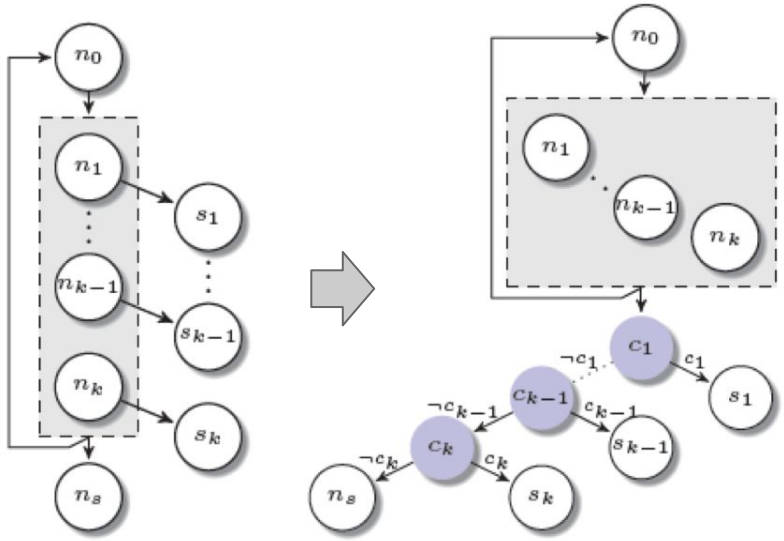
***Abstract*—Decompilation is important for many security applications; it facilitates the tedious task of manual malware reverse engineering and enables the use of source-based security tools on binary code. This includes tools to find vulnerabilities, discover bugs, and perform taint tracking. Recovering high-level control constructs is essential for decompilation in order to produce structured code that is suitable for human analysts and source-based program analysis techniques. State-of-the-art decompilers rely on structural analysis, a pattern-matching approach over the control flow graph, to recover control constructs from binary code. Whenever no match is found, they generate `goto` statements and thus produce unstructured decompiled output. Those statements are problematic because they make decompiled code harder to understand and less suitable for program analysis.**

In this paper, we present DREAM, the first decompiler to offer a `goto`-free output. DREAM uses a novel *pattern-independent* control-flow structuring algorithm that can recover all control constructs in binary programs and produce structured decompiled code without any `goto` statement. We also present *semantics-preserving transformations* that can transform unstructured control flow graphs into structured graphs. We demonstrate the correctness of our algorithms and show that we outperform

effective countermeasures and mitigation strategies requires a thorough understanding of functionality and actions performed by the malware. Although many automated malware analysis techniques have been developed, security analysts often have to resort to manual reverse engineering, which is difficult and time-consuming. Decompilers that can reliably generate high-level code are very important tools in the fight against malware: they speed up the reverse engineering process by enabling malware analysts to reason about the high-level form of code instead of its low-level assembly form.

Decompilation is not only beneficial for manual analysis, but also enables the application of a wealth of source-based security techniques in cases where only binary code is available. This includes techniques to discover bugs [5], apply taint tracking [10], or find vulnerabilities such as RICH [7], KINT [38], Chucky [42], Dowser [24], and the property graph approach [41]. These techniques benefit from the high-level abstractions available in source code and therefore are faster and more efficient than their binary-based counterparts. For example, the average runtime overhead for the source-


p_0
n_0
p_1
n_1
p_{k-1}
n_{k-1}
p_k
n_k
p_0
$i=0$
c_0
p_1
$i=1$
c_0
p_{k-1}
$i=k-1$
c_0
n_0
c_1
p_k
$i=k$
c_1
n_1
$i=0$
c_{k-1}
c_{k-1}
$\neg c_{k-1}$
n_{k-1}
$i=0$
n_k
$i=0$


n_0
n_1
s_1
n_{k-1}
s_{k-1}
n_k
s_k
n_s
n_0
n_1
n_{k-1}
n_k
c_1
$\neg c_1$
c_1
s_1
c_{k-1}
$\neg c_{k-1}$
c_{k-1}
s_{k-1}
c_k
$\neg c_k$
c_k
n_s
s_k

Cautionary Tale: Metrics

Metrics for decompilation quality that we evaluated with SAILR:

- # Gotos
- # Calls (action duplication)
- # Booleans (conditional duplication)
- Cyclomatic Complexity
- CFG Edit Distance

Each metric is gameable on its own.

(Clearly impactful for ML)

Gaming GOTOs

DREAM proposes "no more gotos" in decompiled code.

But that doesn't meant the code is recognizable...

```c
int schedule_job(int needs_next, int fast_job, int mode)
{
    if (needs_next && fast_job) {
        complete_job();
        if (mode == EARLY_EXIT)
            goto cleanup;

        next_job();
    }

    refresh_jobs();
    if (fast_job)
        fast_unlock();

cleanup:
    complete_job();
    log_workers();
    return job_status(fast_job);
}
```

Original (derived from Linux Kernel)

```c
long long schedule_job(unsigned int a0,
    unsigned int a1, unsigned int a2)
{
    if (a0 && a1)
    {
        complete_job();
        if (EARLY_EXIT != a2)
        {
            next_job();
            refresh_jobs();
        }
    }

    if (!a0 || !a1)
        refresh_jobs();
    if (a1 && (!a0 || EARLY_EXIT != a2))
        fast_unlock();

    complete_job();
    log_workers();
    return job_status(a1);
}
```

DREAM

Gaming non-GOTOs

GOTOs enable arbitrary structures in the decompiled CFG...

... allowing for a CFGed of 0
... allowing for a Cyclomatic Complexity of 0
... allowing for zero duplication of function calls

The only way to quantify decompilation quality is GOTOs plus another metric.

	Source	SAILR	IDA	Ghidra	DREAM
GOTOs	1,367	2,673	6,115	6,575	0
CFGed	0	166,468	165,583	187,509	388,231
Bools	6,180	3,980	4,279	4,850	43,661

Sum of metrics of 7,355 functions across 26 popular Debian packages.

Example

```
void qemu_clock_enable(
     QEMUClock *clock, bool enabled) {



  bool old = clock->enabled;
  clock->enabled = enabled;
  if (enabled && !old) {
    qemu_clock_notify(clock);
  }


}
```

(a) Original source code.