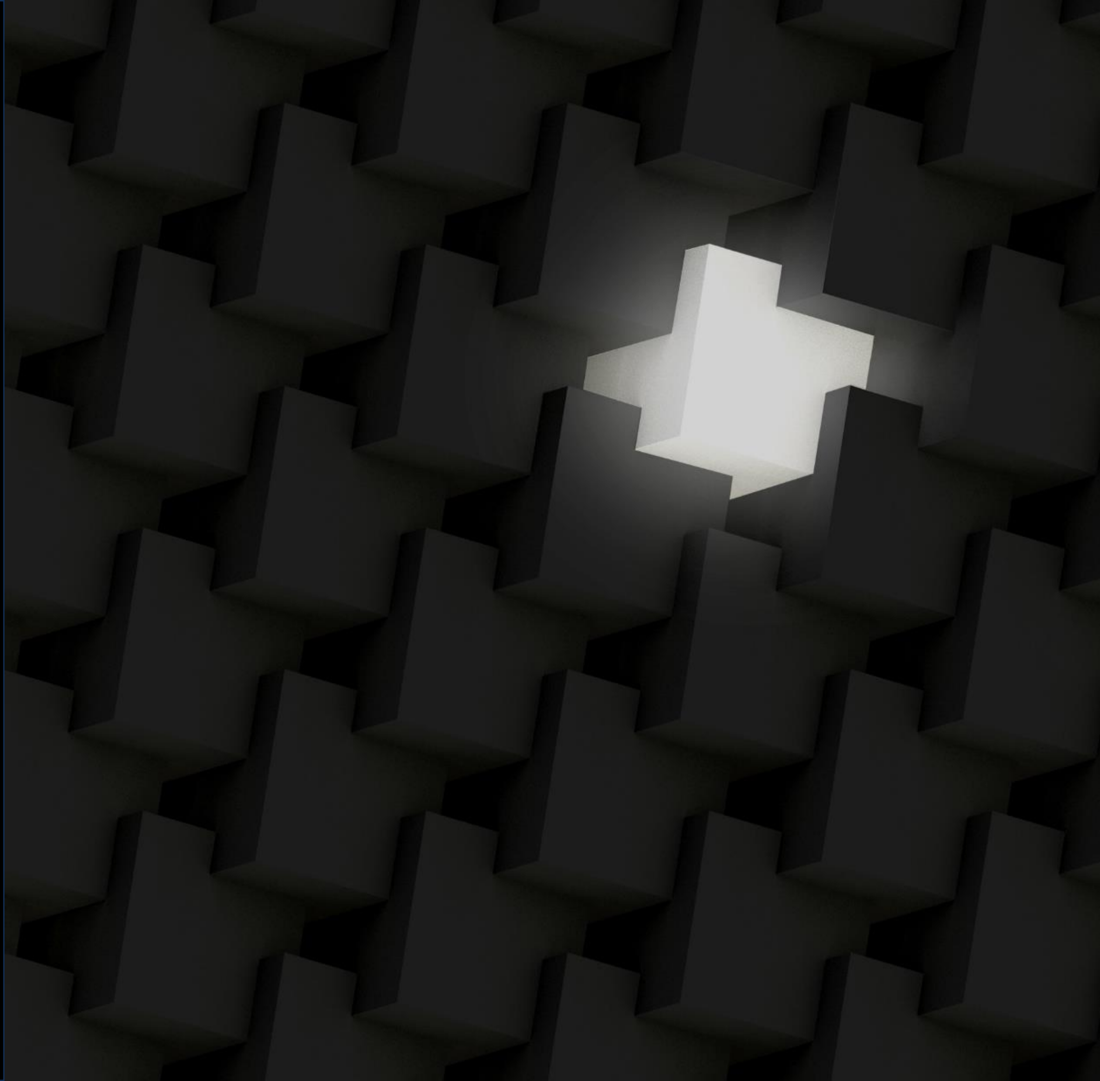
Carnegie Mellon University
Software Engineering Institute

RESEARCH REVIEW 2020

Advancing Cyber Operator Tradecraft through Automated Static Binary Analysis

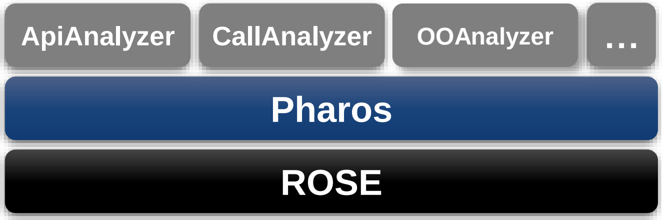
Cory Cohen & Dr. Edward Schwartz



This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM20-0906

Executable Code Analysis Team at CERT



Building tools to solve DoD program analysis challenges!

- Historically focused on malware reverse engineering (RE)
- Focused on software assurance & vulnerability discovery

Pharos is a static binary analysis framework that

- Extends the LLNL ROSE compiler infrastructure (http://rosecompiler.org), DOE sensitive to DoD needs

Also working extensively in NSA's Ghidra RE platform

Tools are focused on **making a difference** in operational tradecraft

- Analyzing malware design
- Performing advanced static emulation
- Recovering data types
- Performing control flow analyses
- Defeating obfuscations

The Pharos Static Binary Analysis Framework

Pharos includes

- File format parsing
- Disassembler
- Function partitioner
- Instruction semantics
- Emulation framework
- Usage-definition chains
- XSB Prolog integration
- Variable type analysis
- API parameter database
- Call parameter analysis

Built on top of ROSE

- Close partnership with LLNL
- Highly extensible
- BSD Licensed
- Implemented as C++ Library

Pharos Framework is publicly available on GitHub at

https://github.com/cmu-sei/pharos

Analyst Tools Built in the Pharos Framework

OO Analyzer

Detects object oriented constructs, resolves virtual function calls

Impact: Greatly reduces the malware analysis effort required for deep understanding of malware capabilities

Call Analyzer

Reports constant parameters to calls in binary executables

Impact: Permits analyst to identify parameters to important operating system API calls to detect undesired behaviors in software

FN2Yara

Automatically generates YARA signatures

Impact: Promotes high-quality signatures to detect similarity in malware families, which can be converted to Snort signatures for use in network defense

FN2Hash

Generates function hashes to identify functions in malware files

Impact: Reduces analyst time spent doing repetitive tasks, automates identification of functions of interest in malware

Malware Design Matcher

Detects high-level design abstractions in malware files

Impact: Automated identification of key abstractions in known families, permits human analysts to record abstract knowledge precisely

Api Analyzer

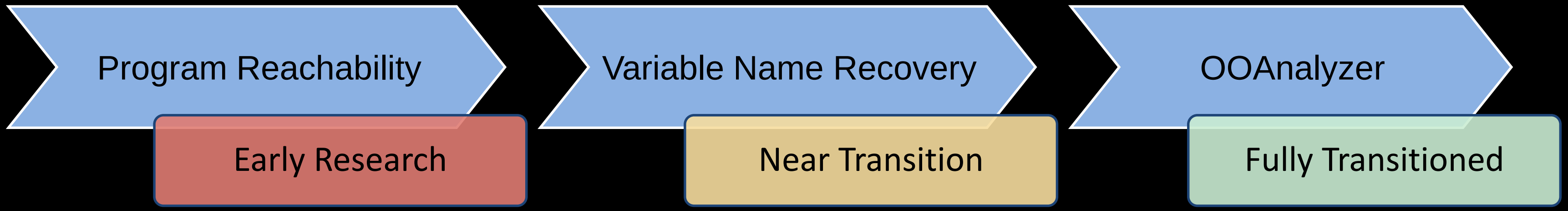
Detects patterns of API calls representing malicious behaviors

Impact: Focuses analyst attention on important aspects of code via automated analysis, detects unexpected patterns for software assurance

Agenda

Today we're going to discuss three examples of how we're advancing cyber operator tradecraft through automated static binary analysis:

- Program Reachability for Vulnerability and Malware Analysis
- Recovering Meaningful Variable Names in Decompiled Code
- Improvements to Object-Oriented Construct Recovery Using OOAnalyzer



RESEARCH REVIEW 2020

Advancing Cyber Operator Tradecraft through Automated Static Binary Analysis

Program Reachability for Vulnerability and Malware Analysis

Problem: Highly skilled Department of Defense (DoD) malware and vulnerability analysts currently spend significant amounts of time manually coercing specific portions of executable code to run.

Solution: Automate the analysis of binary code, choosing program inputs that will trigger specific behavior to reduce the time that DoD cyber personnel spend performing complex software analysis.

Approach: Use model checking techniques to identify these inputs and generate a simplified executable free of complex and convoluted dependencies that can be analyzed by existing code analysis tools.

Path Finder Design Overview

```
cmp ecx, 76h
jge  exit

cmp ecx, 42h
jnz  normal

call check
cmp eax, 34h
jnz normal

call malware

call normal

inc ecx
jmp loop
```

Analyst

Under what conditions does the malicious code execute?

Z3 SMT Constraint Problem → Satisfiable: ecx = 42h check() = 34h → Rewritten Executable Always Runs Code

Evaluating Multiple Approaches/Implementations

Pharos Function Summaries

Completely remove or greatly simplify functions that are not important to improve performance.

Weakest Precondition

Analyze function input and output states to minimize complexity for solver while being as accurate as possible.

Property Directed Reachability (PDR)

Base analysis on complete symbolic behavior of instructions to increase accuracy.

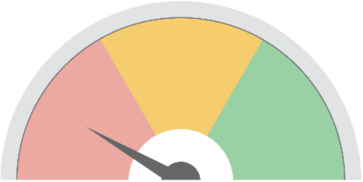
Ghidra + Seahorn

A more source-code centric approach to resolving the problems presented by our early PDR attempts.

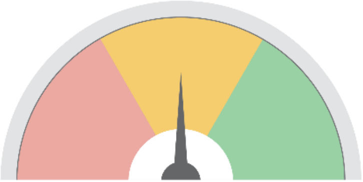
Accuracy? →

← Scalability?

Pharos Function Summaries



Accuracy = Poor



Speed = Fair

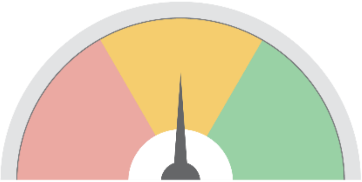
Represent functions using a simplified model of memory and loops that reduces the complexity of the problem sent to the SMT solver.

Very fast when it works correctly!

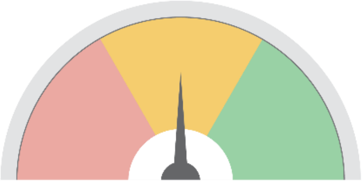
Limitations are becoming more obvious as we test more complex cases and push the limits of the approach.

Accuracy	Speed	
		Memory is represented simply and efficiently (as a scalar map).
		Loops are unrolled, which is unable to prove some paths.
		Great when it works, but limitations are becoming more obvious now.

Weakest Precondition Approach (WP)



Accuracy = Fair



Speed = Fair

Use an intermediate representation (IR) based on the full semantics of the instructions to model the program accurately.

More accurate than Pharos function summary approach and more stable performance than the PDR approach.

But can this approach really beat PDR?

Accuracy	Speed	
		Memory is represented precisely as a single large array.
		Loops are unrolled, which is unable to prove some paths.
		Efficient algorithm generates formulas that are linear in size.

Property Directed Reachability Approach (PDR/IC3)

This PDR approach

- Is related to work from model checking
- Can reason correctly about loops
- Hasn't really been used on executables

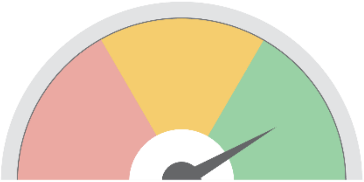
Collaboration with Dr. Arie Gurfinkel

- University of Waterloo
- Expert in Z3 SMT & PDR
- Creator of SPACER PDR Engine

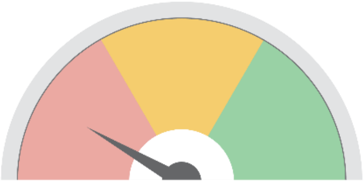
Dr. Arie Gurfinkel
University of Waterloo

We're improving support for bit vectors and arrays.

Property Directed Reachability Approach (PDR)



Accuracy = Good



Speed = Poor

PDR approach is clearly more capable.

However, the performance is highly variable.

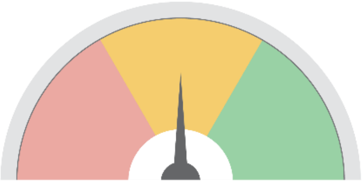
It often gets stuck guessing the bits of a value.

It struggles with proving memory model properties.

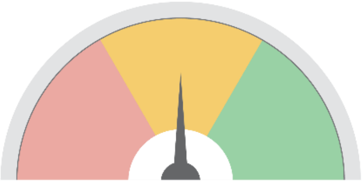
Details of SMT representation seem to matter a lot more than in other approaches.

Accuracy	Speed	
		Memory is represented precisely as a single large array.
		SPACER is able to reason about loops correctly but slowly.

Ghidra & Seahorn Approach



Accuracy = Fair



Speed = Fair

Uses same SPACER based solve engine as PDR.

Ghidra decompiler used to lift program representation in LLVM.

Seahorn (source code analysis) used to answer reachability. This approach known to work fairly well.

Big Question: How accurate is the decompilation?

Accuracy	Speed	
		Each stack frame is represented as a separate memory array.
		SPACER is able to reason about loops correctly but slowly.

Overall Assessment of Approaches (Pass/Fail/Timeout)

Test Case Configuration		Pharos Function Summaries			Weakest Precondition			Property Directed Reachability			Ghidra/Seahorn		
Optimized	Arch	Fail	Tout	Pass	Fail	Tout	Pass	Fail	Tout	Pass	Fail	Tout	Pass
None	32-bit	55	2	34	16	2	73	3	29	59	21	7	63
None	64-bit	47	0	44	15	3	73	2	36	53	28	2	61
Medium	32-bit	40	0	51	9	3	79	1	13	77	12	7	72
Medium	64-bit	53	0	38	9	4	78	1	17	73	21	6	64
High	32-bit	50	0	41	6	2	83	1	12	78	18	7	66
High	64-bit	32	1	58	28	3	60	2	16	73	32	5	54
Total		**257**	**3**	**266**	**83**	**17**	**446**	**10**	**123**	**413**	**132**	**34**	**380**

There were 91 tests in each optimization/architecture configuration.
Red = Worst, Green = Best, Yellow = 2nd place, Gold = 3rd place
Results are not intended to be definitive but to communicate our experience.

There's no one solution that clearly wins!

Summary of Conclusions

Path reachability in binary executables continues to be a very hard problem!

Primary concern in each approach:

- Pharos FS: Not accurate enough.
- Weakest Precondition: Technically the winner, but has known deficiencies.
- SPACER: Timeouts caused by memory layout complexity a serious problem.
- Ghidra + Seahorn: Unclear if lifting can reach required correctness.

But, we have a good test set to continue to monitor the state of the art!

Perhaps dynamic approaches such as concolic execution deserve more attention?

RESEARCH
REVIEW
2020

Advancing Cyber Operator Tradecraft through Automated Static Binary Analysis

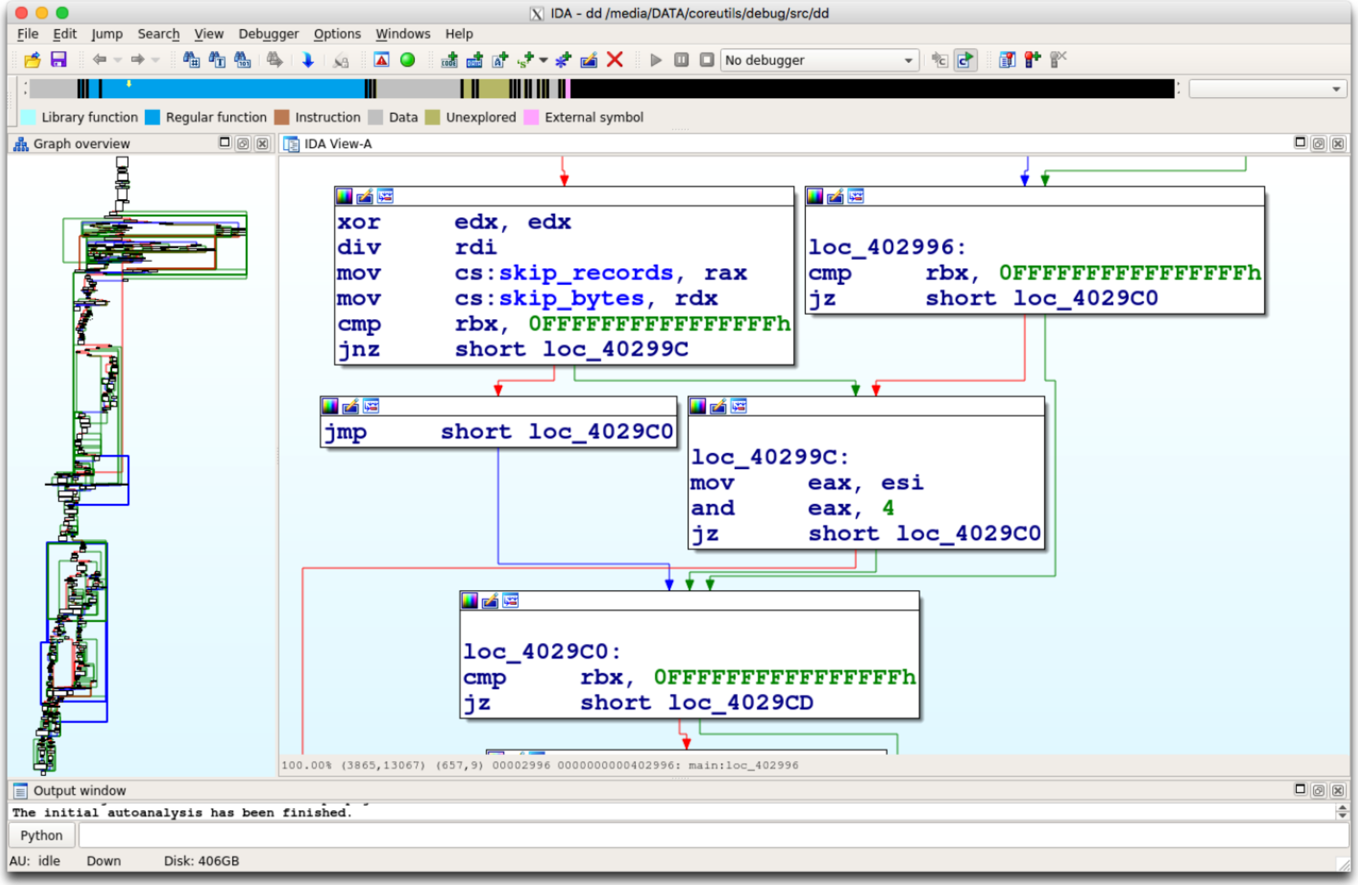
Recovering Meaningful Variable Names in Decompiled Code

Disassembler

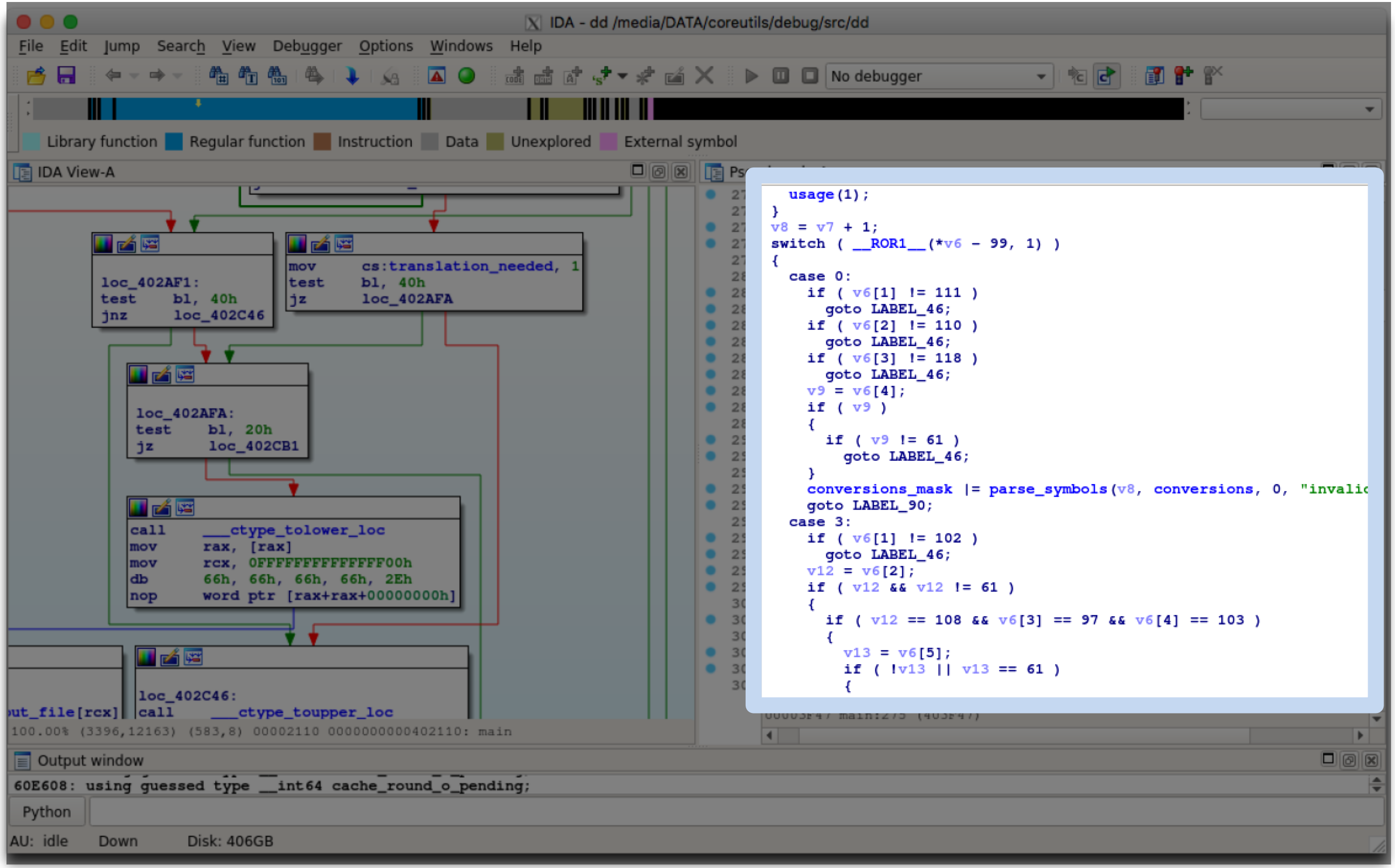
1. jlacomis@gs17931:~/Data/coreutils/debug/src (ssh)

```
  40299c:       89 f0                   mov    %esi,%eax
  40299e:       83 e0 04                and    $0x4,%eax
  4029a1:       74 1d                   je     4029c0 <main+0x8b0>
  4029a3:       31 d2                   xor    %edx,%edx
  4029a5:       48 89 d8                mov    %rbx,%rax
  4029a8:       48 f7 f7                div    %rdi
  4029ab:       48 89 05 be b8 20 00    mov    %rax,0x20b8be(%rip)
  4029b2:       48 89 15 ff ba 20 00    mov    %rdx,0x20baff(%rip)
  4029b9:       4d 85 c0                test   %r8,%r8
  4029bc:       75 14                   jne    4029d2 <main+0x8c2>
  4029be:       eb 31                   jmp    4029f1 <main+0x8e1>
  4029c0:       48 83 fb ff             cmp    $0xffffffffffffffff,%rbx
  4029c4:       74 07                   je     4029cd <main+0x8bd>
  4029c6:       48 89 1d a3 b8 20 00    mov    %rbx,0x20b8a3(%rip)
  4029cd:       4d 85 c0                test   %r8,%r8
  4029d0:       74 1f                   je     4029f1 <main+0x8e1>
  4029d2:       89 c8                   mov    %ecx,%eax
  4029d4:       83 e0 10                and    $0x10,%eax
  4029d7:       74 18                   je     4029f1 <main+0x8e1>
```

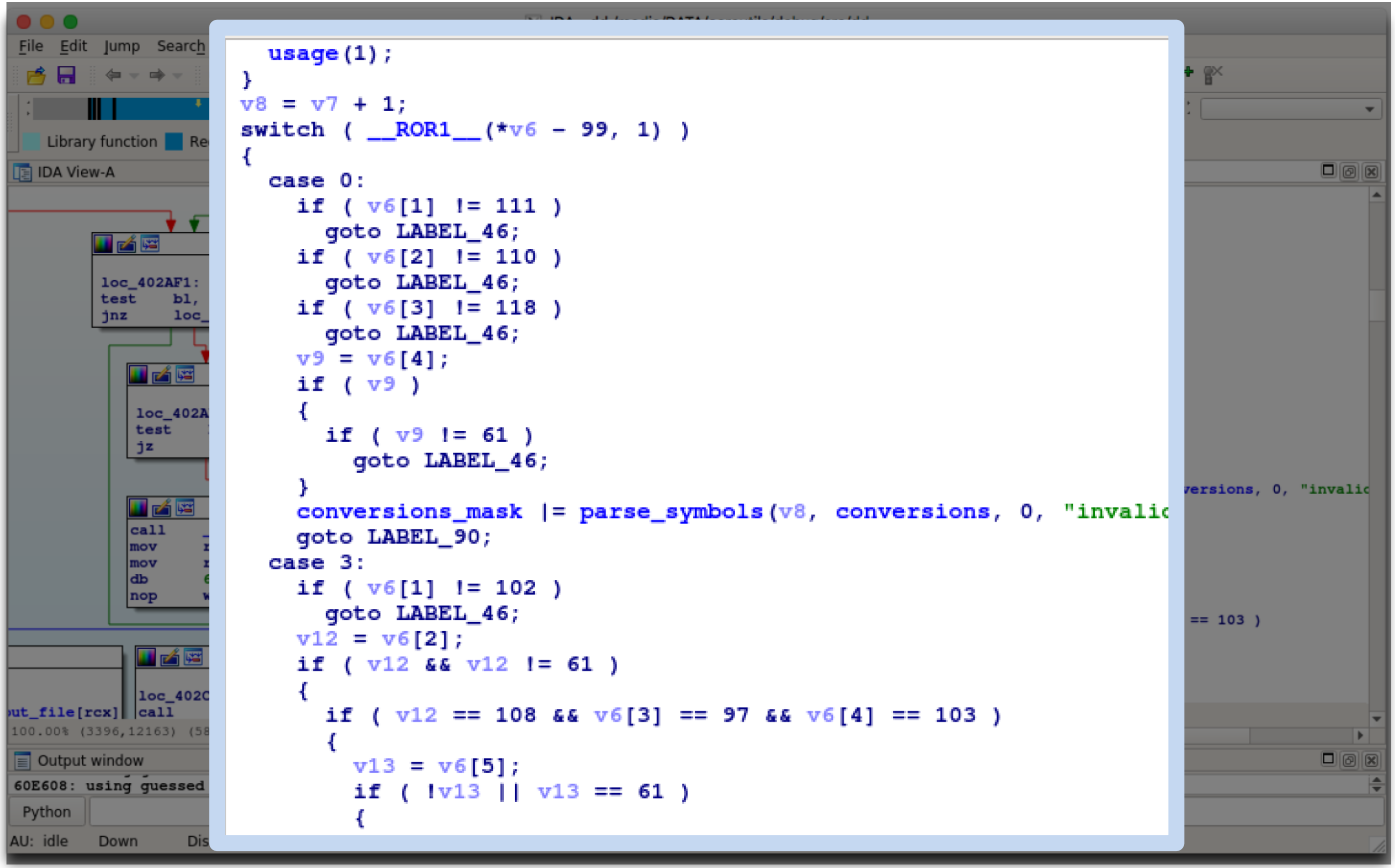
Disassembler



Decompiler


IDA - dd /media/DATA/coreutils/debug/src/dd
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Regular function Instruction Data Unexplored External symbol
IDA View-A
loc_402AF1:
test bl, 40h
jnz loc_402C46
mov cs:translation_needed, 1
test bl, 40h
jz loc_402AFA
loc_402AFA:
test bl, 20h
jz loc_402CB1
call ___ctype_tolower_loc
mov rax, [rax]
mov rcx, 0FFFFFFFFFFFFFF00h
db 66h, 66h, 66h, 66h, 2Eh
nop word ptr [rax+rax+00000000h]
loc_402C46:
call ___ctype_toupper_loc
100.00% (3396,12163) (583,8) 00002110 0000000000402110: main
usage(1);
}
v8 = v7 + 1;
switch (__ROR1__(*v6 - 99, 1))
{
case 0:
if (v6[1] != 111)
goto LABEL_46;
if (v6[2] != 110)
goto LABEL_46;
if (v6[3] != 118)
goto LABEL_46;
v9 = v6[4];
if (v9)
{
if (v9 != 61)
goto LABEL_46;
}
conversions_mask |= parse_symbols(v8, conversions, 0, "invalic
goto LABEL_90;
case 3:
if (v6[1] != 102)
goto LABEL_46;
v12 = v6[2];
if (v12 && v12 != 61)
{
if (v12 == 108 && v6[3] == 97 && v6[4] == 103)
{
v13 = v6[5];
if (!v13 || v13 == 61)
{
Output window
60E608: using guessed type __int64 cache_round_o_pending;
Python
AU: idle Down Disk: 406GB

Decompiler


File Edit Jump Search
Library function
IDA View-A
usage(1);
}
v8 = v7 + 1;
switch (__ROR1__(*v6 - 99, 1))
{
case 0:
if (v6[1] != 111)
goto LABEL_46;
if (v6[2] != 110)
goto LABEL_46;
if (v6[3] != 118)
goto LABEL_46;
v9 = v6[4];
if (v9)
{
if (v9 != 61)
goto LABEL_46;
}
conversions_mask |= parse_symbols(v8, conversions, 0, "invalic
goto LABEL_90;
case 3:
if (v6[1] != 102)
goto LABEL_46;
v12 = v6[2];
if (v12 && v12 != 61)
{
if (v12 == 108 && v6[3] == 97 && v6[4] == 103)
{
v13 = v6[5];
if (!v13 || v13 == 61)
{
Output window
Python
AU: idle Down

The problem:

Decompilers are typically unable to assign meaningful names to variables.

Our Work

Decompiler output → Refactored decompiler output

```
void *file_mmap(int V1, int V2)
{
  void *V3;
  V3 = mmap(0, V2, 1, 2, V1, 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3;
}
```

```
void *file_mmap(int fd, int size)
{
  void *ret;
  ret = mmap(0, size, 1, 2, fd, 0);
  if (ret == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return ret;
}
```

Our Work

Decompiler output → Refactored decompiler output

```
void *file_mmap(int V1, int V2)
{
  void *V3;
  V3 = mmap(0, V2, 1, 2, V1, 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3;
}
```

```
void *file_mmap(int fd, int size)
{
  void *ret;
  ret = mmap(0, size, 1, 2, fd, 0);
  if (ret == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return ret;
}
```

Our Work

Decompiler output → Refactored decompiler output

```
void *file_mmap(int V1, int V2)
{
  void *V3;
  V3 = mmap(0, V2, 1, 2, V1, 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3;
}
```

```
void *file_mmap(int fd, int size)
{
  void *ret;
  ret = mmap(0, size, 1, 2, fd, 0);
  if (ret == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return ret;
}
```

Up to 74%

recovery of original source code names
on an open-source GitHub corpus

Why does it work?

Natural Language

Natural Language

Key Principle: Software is “Natural”

On the Naturalness of Software

Abram Hindle, Earl Barr, Zhendong Su
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
{ajhindle,barr,su}@cs.ucdavis.edu

Mark Gabel
Dept. of Computer Science
The University of Texas at Dallas
Richardson, TX 75080 USA
mark.gabel@utdallas.edu

Prem Devanbu
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
devanbu@cs.ucdavis.edu

***Abstract*—Natural languages like English are rich, complex, and powerful. The highly creative and graceful use of languages like English and Tamil, by masters like Shakespeare and Avvaiyar, can certainly delight and inspire. But in practice, given cognitive constraints and the exigencies of daily life, most human utterances are far simpler and much more repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension.**

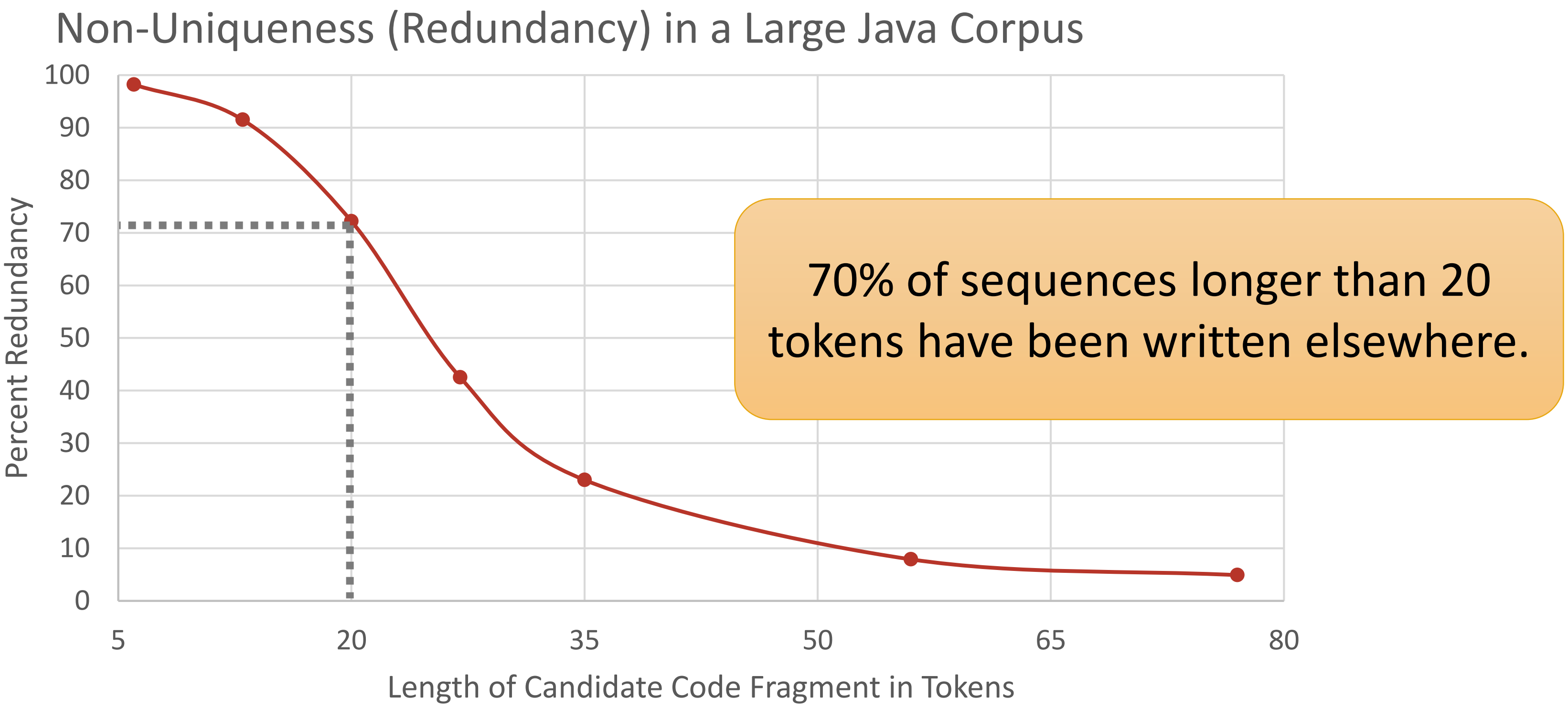
We begin with the conjecture that most software is also natural, in the sense that it is created by humans at work,

efforts in the 1960s. In the ’70s and ’80s, the field was re-animated with ideas from logic and formal semantics, which still proved too cumbersome to perform practical tasks at scale. Both these approaches essentially dealt with NLP from first principles—addressing *language*, in all its rich theoretical glory, rather than examining corpora of actual *utterances*, *i.e.*, what people actually write or say. In the 1980s, a fundamental shift to *corpus-based, statistically rigorous* methods occurred. The availability of large, on-line corpora of natural language text, including “aligned” text with translations in multiple languages,[1] along with the computational muscle (CPU speed,

(Presented at the 2012 International Conference on Software Engineering)
http://earlbarr.com/publications/naturalness.pdf

Software is really repetitive

Gabel & Su, 2010



How can we use this?

Idea

Learn typical variable names in a given context from examples … many, many examples.

If software is repetitive, so are names.

```
int main(int  ?  ,
```

Idea

Learn typical variable names in a given context from examples … many, many examples.

If software is repetitive, so are names.

```
int main(int banana,
```

Idea

Learn typical variable names in a given context from examples … many, many examples.

If software is repetitive, so are names.

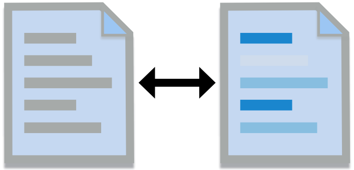
```
int main(int argc,
```

Good news:

We can generate arbitrarily many examples.

GitHub github + Compiler/Decompiler tools

Source code with meaningful names

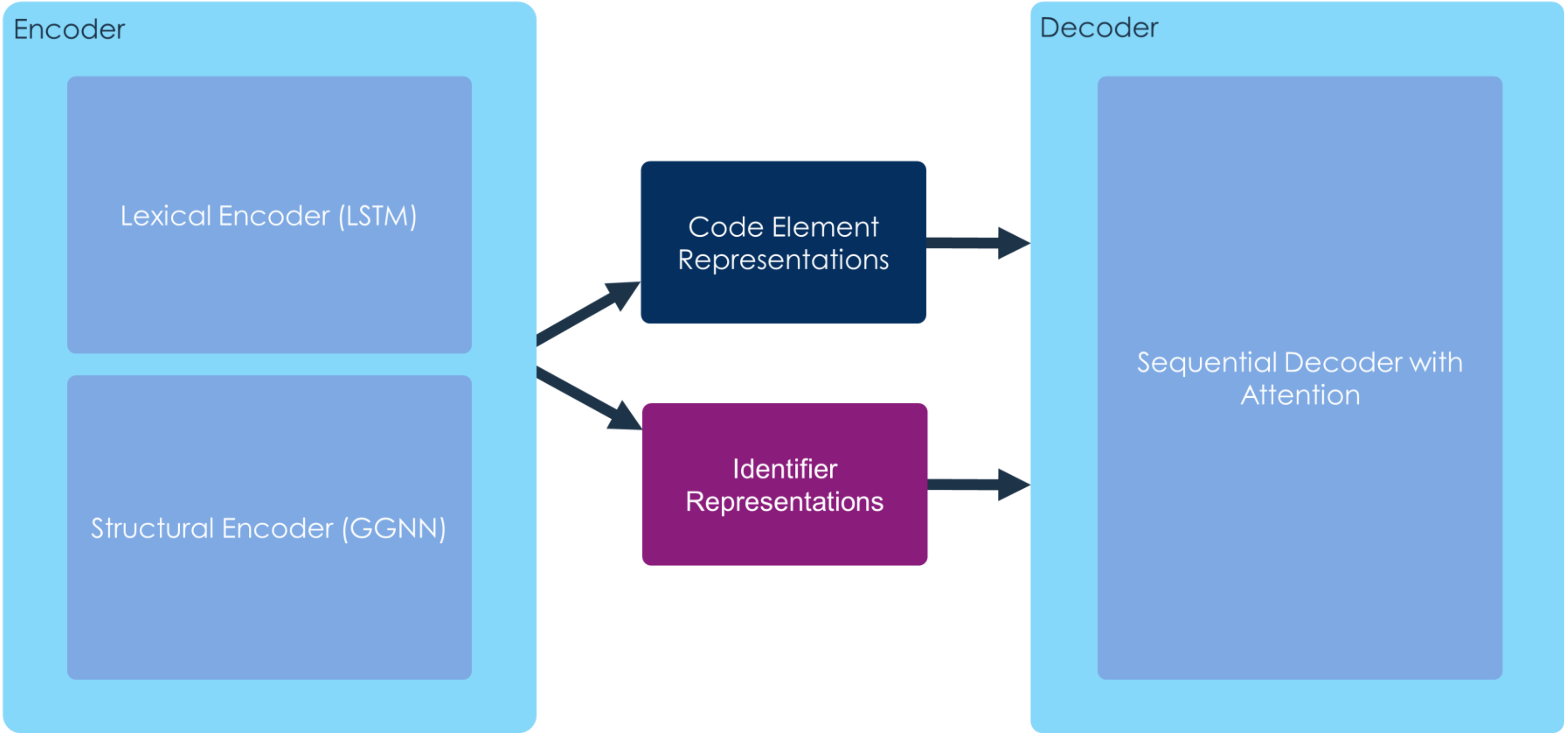


Decompiler output with placeholder names

Github Dataset

- 164,632 unique x86-64 binaries
- 1,259,935 decompiled functions
- Split by binary into test, training, and validation

Neural Network Overview



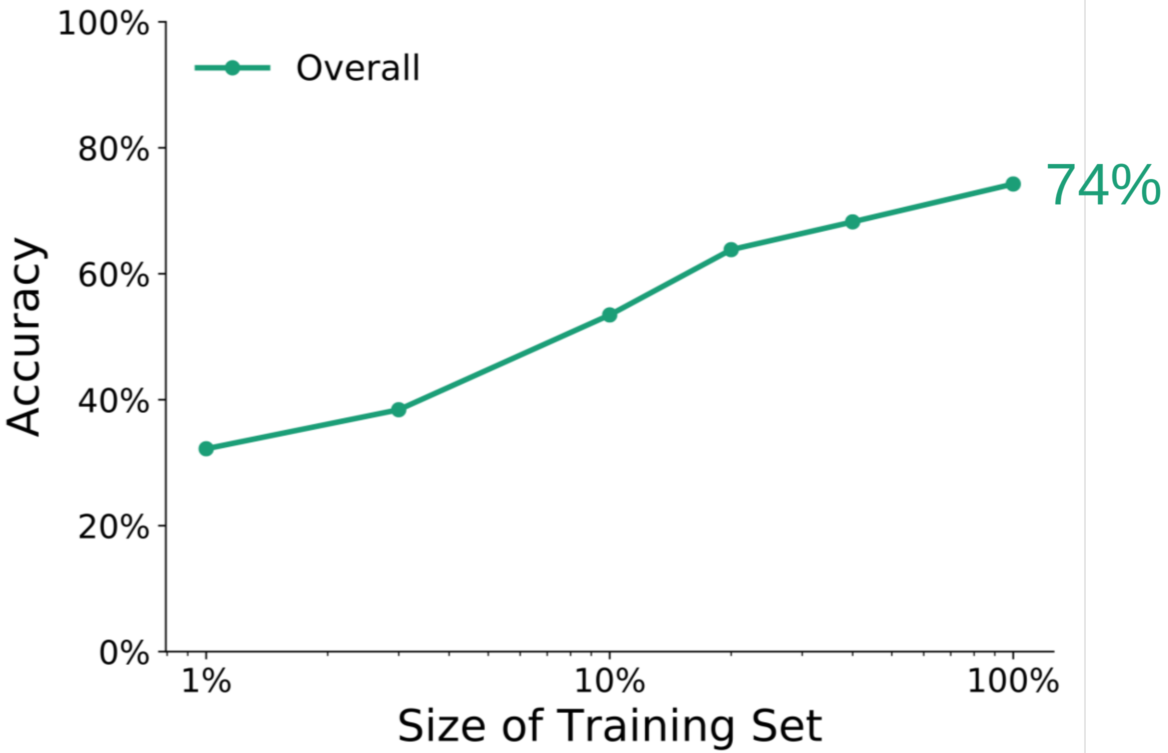
How good are the renamings?

Assumption:

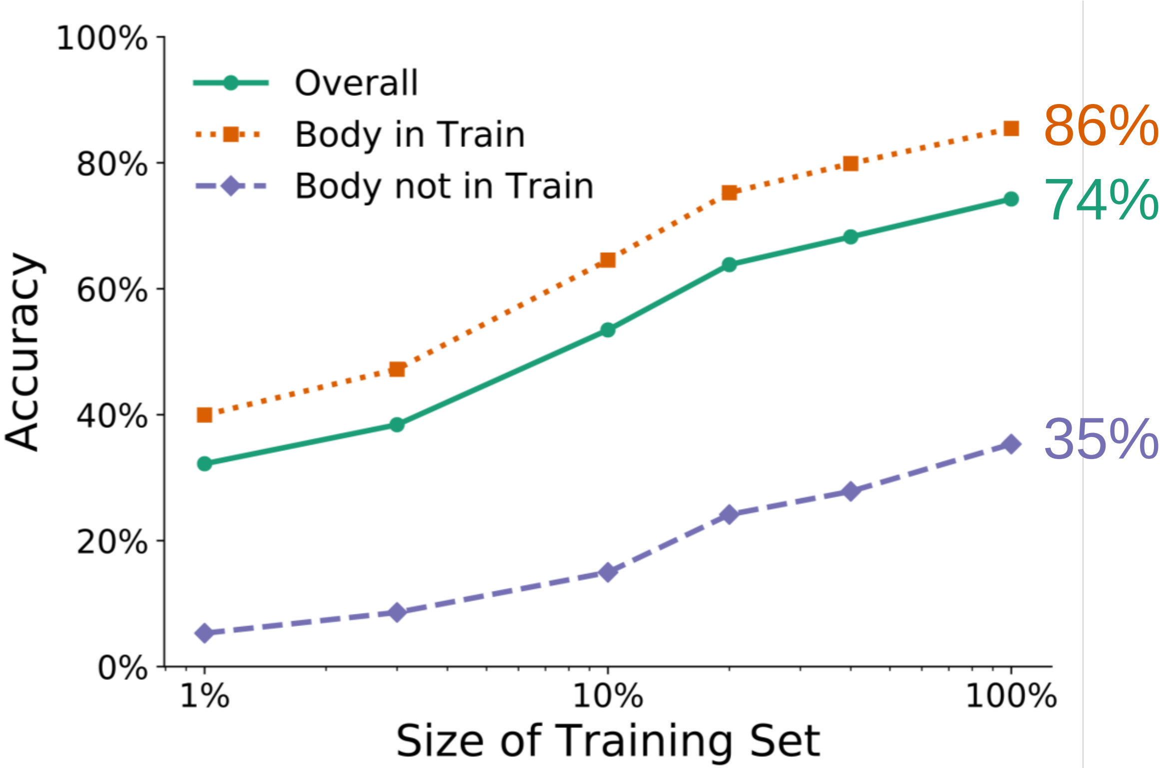
Original (human-written) names are good.

How many can we recover?

The Amount of Training Data Matters



The Uniqueness of Data Matters



Example

```
file *f_open(char **V1, char *V2, int V3) {
  int fd;
  if (!V3)
    return fopen(*V1, V2);
  if (*V2 != 119)
    assert_fail("fopen");
  fd = open(*V1, 577, 384);
  if (fd >= 0)
    return reopen(fd, V2);
  else
    return 0;
}
```

	Developer
V1	filename
V2	mode
V3	is_private

Example

```
file *f_open(char **V1, char *V2, int V3) {
  int fd;
  if (!V3)
    return fopen(*V1, V2);
  if (*V2 != 119)
    assert_fail("fopen");
  fd = open(*V1, 577, 384);
  if (fd >= 0)
    return reopen(fd, V2);
  else
    return 0;
}
```

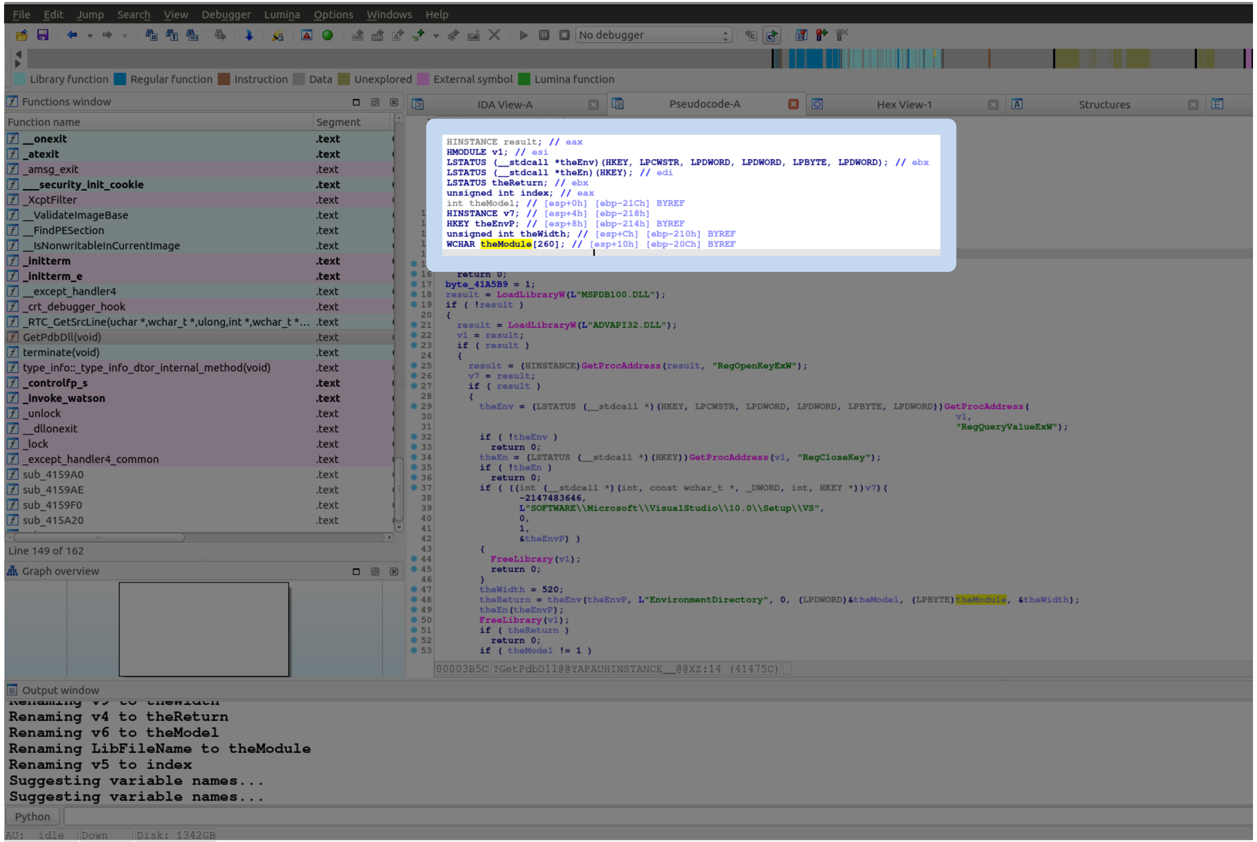
	Developer	Recovered
V1	filename	filename
V2	mode	mode
V3	is_private	create

Transitioning from Research to Practice

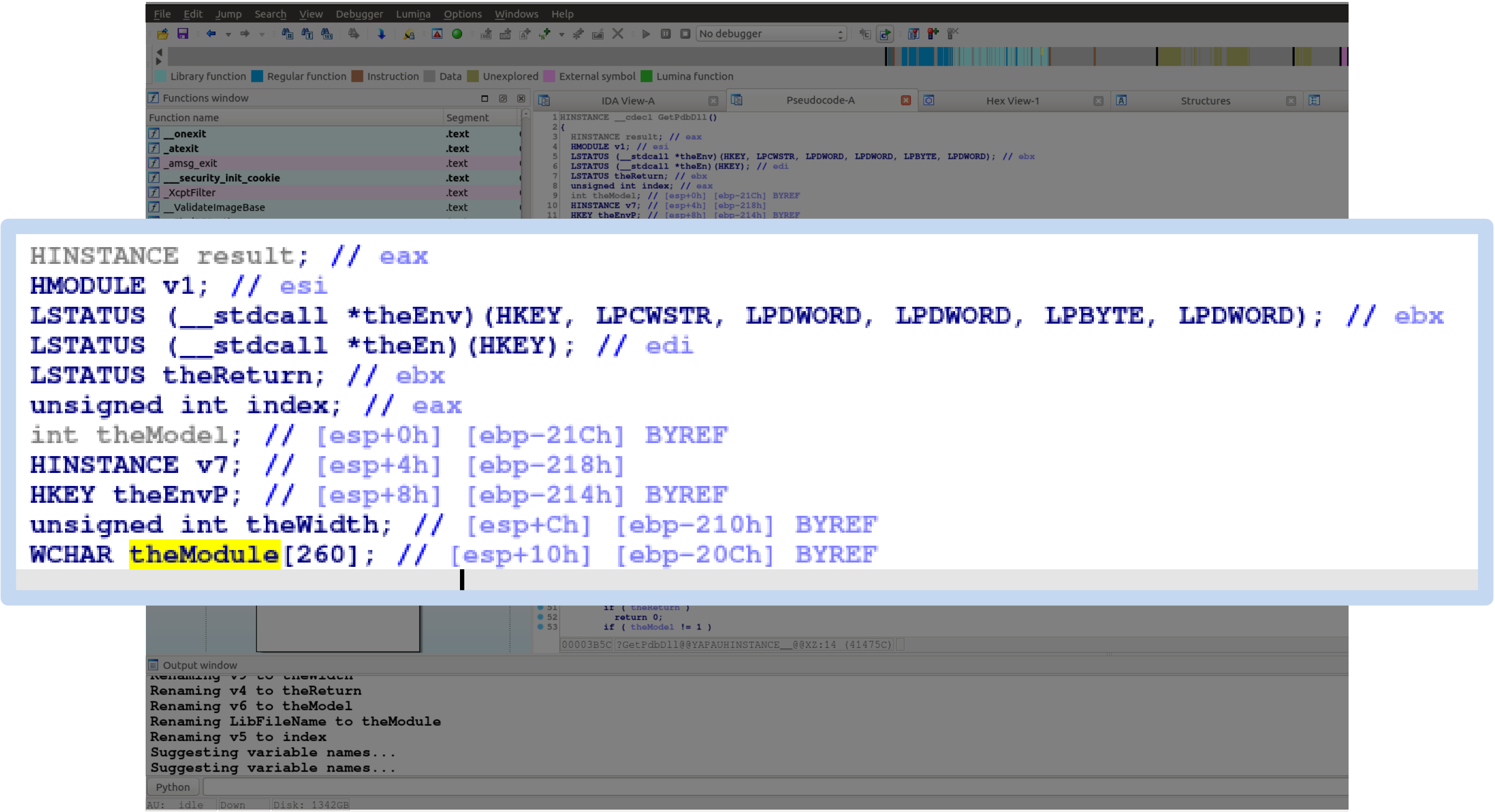
Research was a <u>proof of concept</u>

- Python command line tools that are difficult to use
- Now implemented as a Hex-Rays Plugin for easy use

Transitioning from Research to Practice



Transitioning from Research to Practice



Software is highly structured and predictable. We can leverage this to recover meaningful variable names by studying existing source code.

We can recover up to 74% of variable names.

The uniqueness of the data is very important.

RESEARCH REVIEW 2020

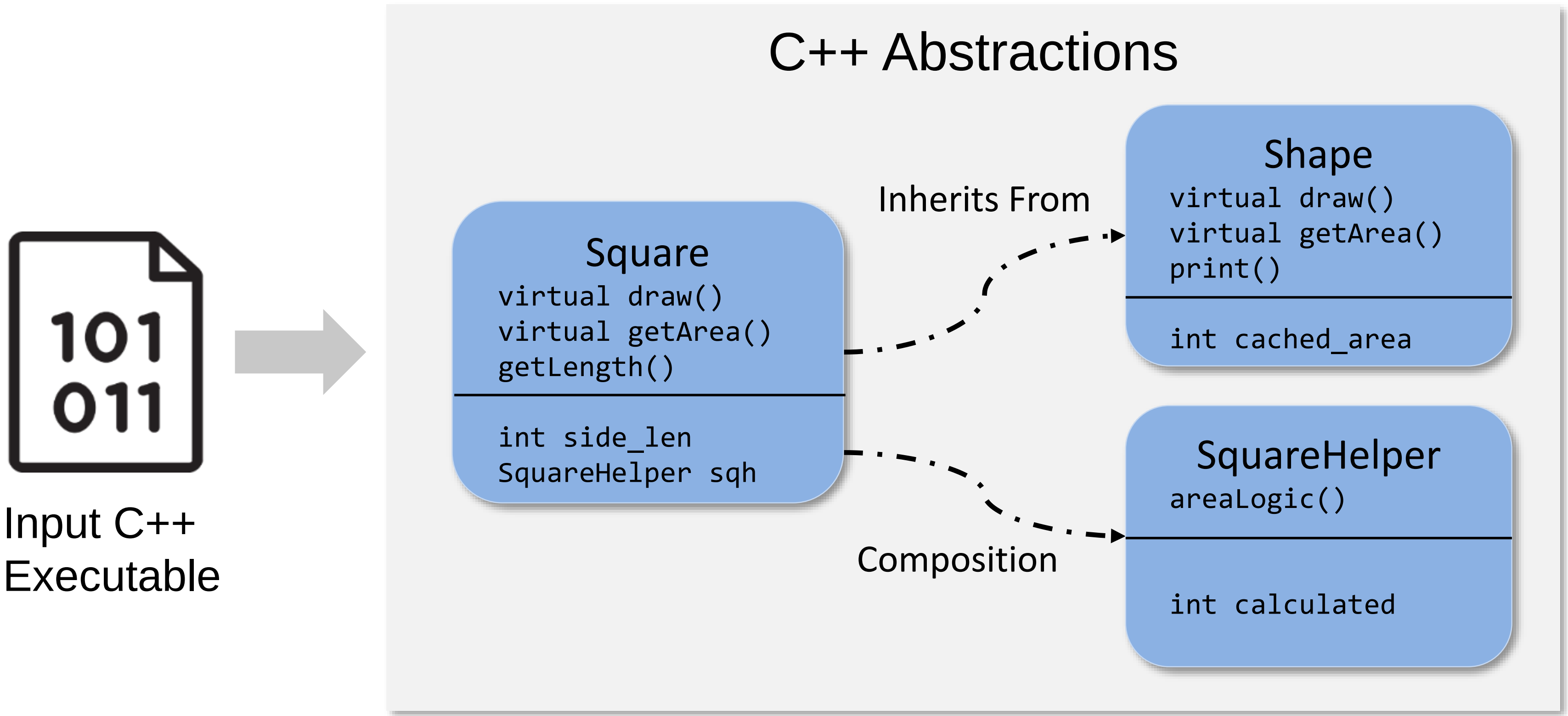
Advancing Cyber Operator Tradecraft through Automated Static Binary Analysis

Improvements to Object-Oriented Construct Recovery Using OOAnalyzer

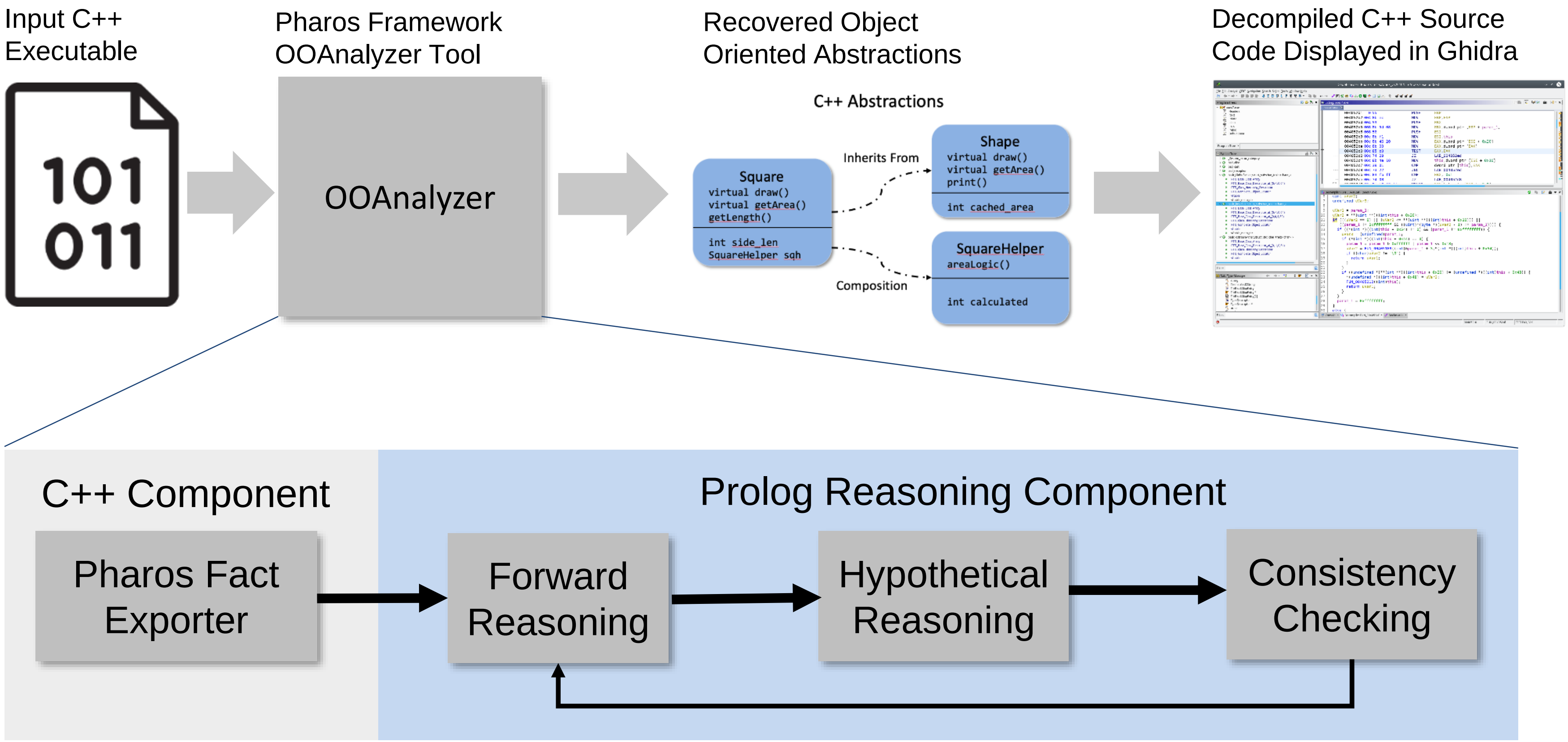
Problem: Object oriented programs have complicated abstractions that are expensive and time consuming to reverse engineer.

Approach: Combine a lightweight program analysis pass with hand written rules in Prolog to automatically recover high-level object oriented constructs.

Object Oriented Abstractions (What Are They?)



OOAnalyzer Design Overview



Why Prolog?

Important information is lost during compilation from source code to executable.

We must make educated guesses and then validate them to find solutions.

New Prolog approach works better than old procedural approach because

- It allows us to backtrack when we make incorrect guesses.
- It expresses compiler behaviors as Prolog rules in a natural format.

Example facts exported to Prolog

- Data and control flow
- Calling convention and parameters

Example Prolog rules

- Only constructors and destructors can update virtual function table pointers.
- Derived classes must be at least large as their base classes.

Fact Exporter

Uses conventional binary analysis to produce <u>initial</u> facts about the program

- Initial facts describe low-level program behaviors

Simple symbolic analysis

- intentionally favors scalability over accuracy
- does not use constraint solvers
- uses a simplified memory model
  - (symbolic memory aliases if memory addresses are equal after simplification)
- is path sensitive up to a threshold

Sufficient because Prolog reasoning system can cope with mistakes

Initial Facts

Initial facts describe low-level program behaviors and form the basis upon which OOAnalyzer's reasoning system operates.

Fact Name	Description
ObjPtrAllocation(I, F, P, S)	Instruction I in function F allocates S bytes of memory for the object pointed to by P.
ObjPtrInvoke(I, F, P, M)	Instruction I in function F calls method M on the object pointed to by P.
ObjPtrOffset(P_1, O, P_2)	Object pointer P_2 points to P_1 + O.
MemberAccess(I, M, O, S)	Instruction I in method M accesses S bytes of memory at offset O from the current object's pointer.
ThisCallMethod(M, P)	Method M receives the object pointed to by P in the ecx register.
NoCallsBefore(M)	No methods are called on any object pointer before method M.
ReturnsSelf(M)	Method M returns the object pointer that was passed as a parameter.
UninitializedReads(M)	Method M reads memory that was not written to by M.
PossibleVFTableEntry(VFT, O, M)	Method M may be at offset O in vftable VFT.

Entity Facts

Entity facts are produced during the reasoning process and describe the high-level model of the program being analyzed.

Fact Name	Description
Method(M)	Method M is an OO method on a class or struct.
Constructor(M)	Method M is an object constructor.
Destructor(M)	Method M is an object destructor.
$Cl_a = Cl_b$	The sets of methods Cl_a and Cl_b both represent methods from the same class. These sets should be combined into a single class.
$Cl_a \leq Cl_b$	Either the sets of methods Cl_a and Cl_b both represent methods from the same class or the methods in Cl_b are inherited from Cl_a.
$M \in Cl$	Method M is defined directly on class Cl.
ClassCallsMethod(Cl, M)	An instance of class Cl calls method M.

Other categories include virtual functions, class relationships, and sizes of classes and tables.

Reasoning Rules

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_n}{C}$$

Forward reasoning

- Unambiguous scenarios
- Interpretation: If P_1, P_2, …, and P_n are satisfied, then C is true
- If inconsistency is reached, P_1, P_2, …, or P_n must not be true

Hypothetical reasoning

- Ambiguous scenarios
- Interpretation: If P_1, P_2, …, and P_n are satisfied, then guess C is true
- If inconsistency is reached, then retract C and assume $\neg C$
- If inconsistency is still reached, P_1, P_2, …, or P_n must not be true

Forward Reasoning

If a method is called on a base class object, it cannot be defined on the derived class.

$$\frac{\begin{array}{c}\mathrm{Constructor}(M_d) \qquad M_d \in Cl_d \\ \mathrm{Constructor}(M_b) \qquad M_b \in Cl_b \\ \mathrm{ClassCallsMethod}(Cl_d, M) \\ \mathrm{ClassCallsMethod}(Cl_b, M) \qquad M_d \neq M_b \\ M \in Cl_m \qquad Cl_d \neq Cl_b \qquad \mathrm{DerivedClass}(Cl_d, Cl_b, \_)\end{array}}{Cl_m \neq Cl_d}$$

Hypothetical Reasoning

If a method is called on a derived class but not a base class,
(first) assume the method is defined on the derived class.

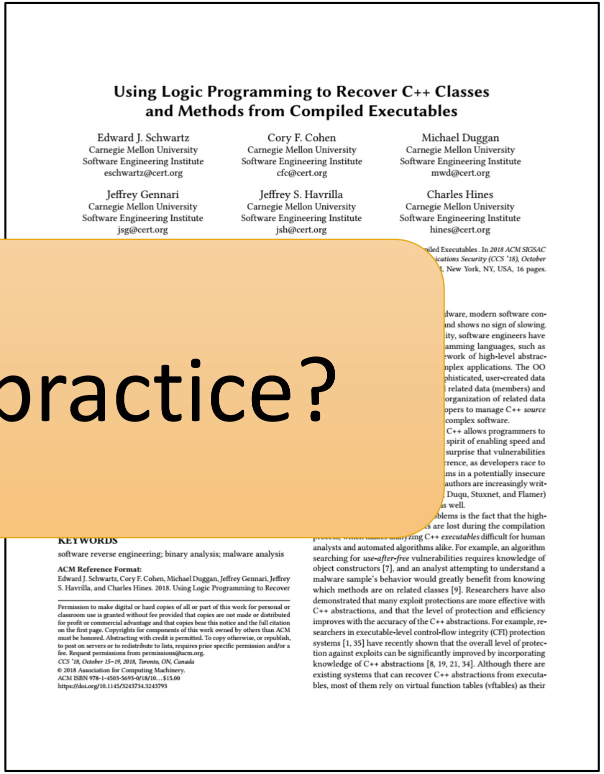
$$\frac{\mathrm{ClassCallsMethod}(\mathrm{Cl}_d, M) \qquad \neg\mathrm{ClassCallsMethod}(\mathrm{Cl}_b, M) \qquad M \in \mathrm{Cl} \qquad \mathrm{DerivedClass}(\mathrm{Cl}_d, \mathrm{Cl}_b, \_)}{\mathrm{Cl}_d = \mathrm{Cl}}$$

OOAnalyzer is the State of the Art in Research

- Unique Prolog-based design
  - Allows human subject knowledge to be easily encoded
  - Back-tracking allows for hypothetical reasoning of properties that cannot be definitely recovered
- Ta
  -

Is it the state of the art in practice?

- Recovers **67-84%** of class abstractions correctly
  - Existing work recovers **<50%** of class abstractions correctly



Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables

Edward J. Schwartz, Carnegie Mellon University, Software Engineering Institute, eschwartz@cert.org
Cory F. Cohen, Carnegie Mellon University, Software Engineering Institute, cfc@cert.org
Michael Duggan, Carnegie Mellon University, Software Engineering Institute, mwd@cert.org
Jeffrey Gennari, Carnegie Mellon University, Software Engineering Institute, jsg@cert.org
Jeffrey S. Havrilla, Carnegie Mellon University, Software Engineering Institute, jsh@cert.org
Charles Hines, Carnegie Mellon University, Software Engineering Institute, hines@cert.org

KEYWORDS

software reverse engineering; binary analysis; malware analysis

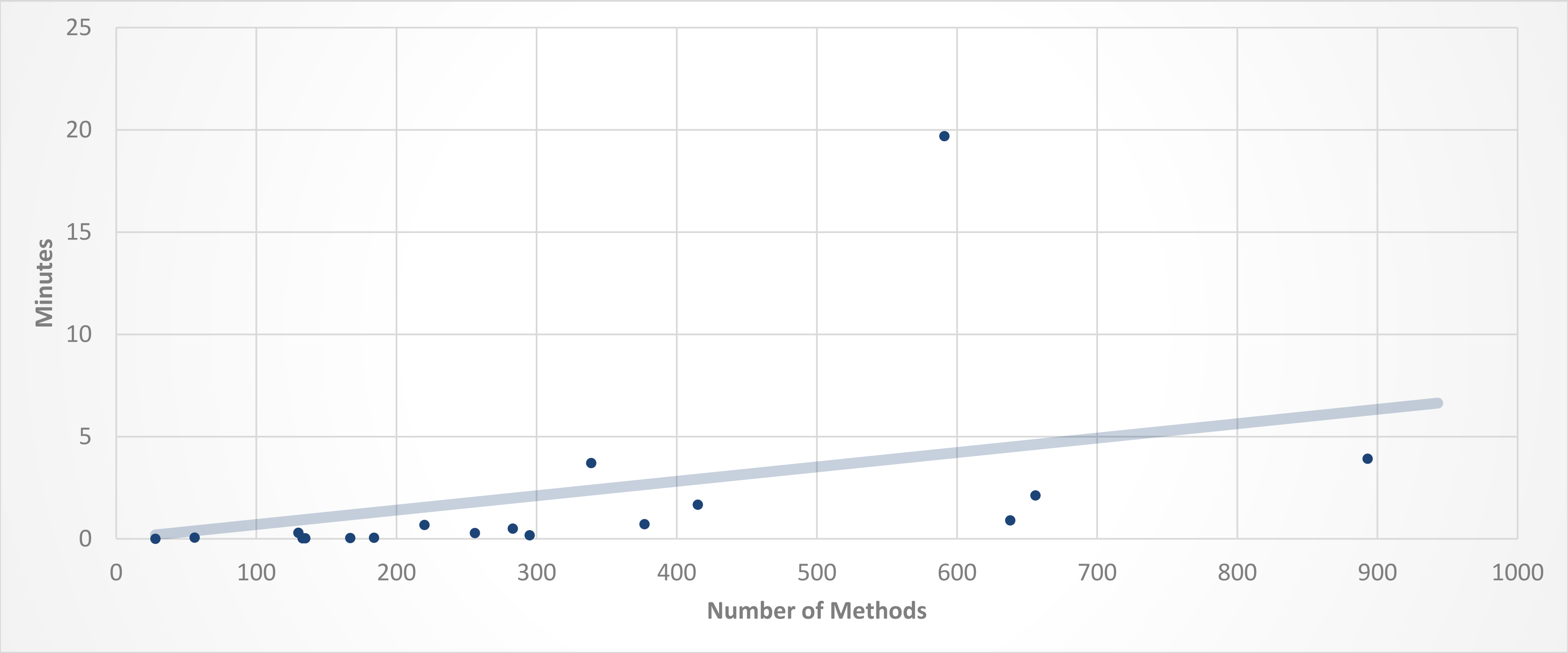
ACM Reference Format:
Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. 2018. Using Logic Programming to Recover

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '18, October 15–19, 2018, Toronto, ON, Canada
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243793

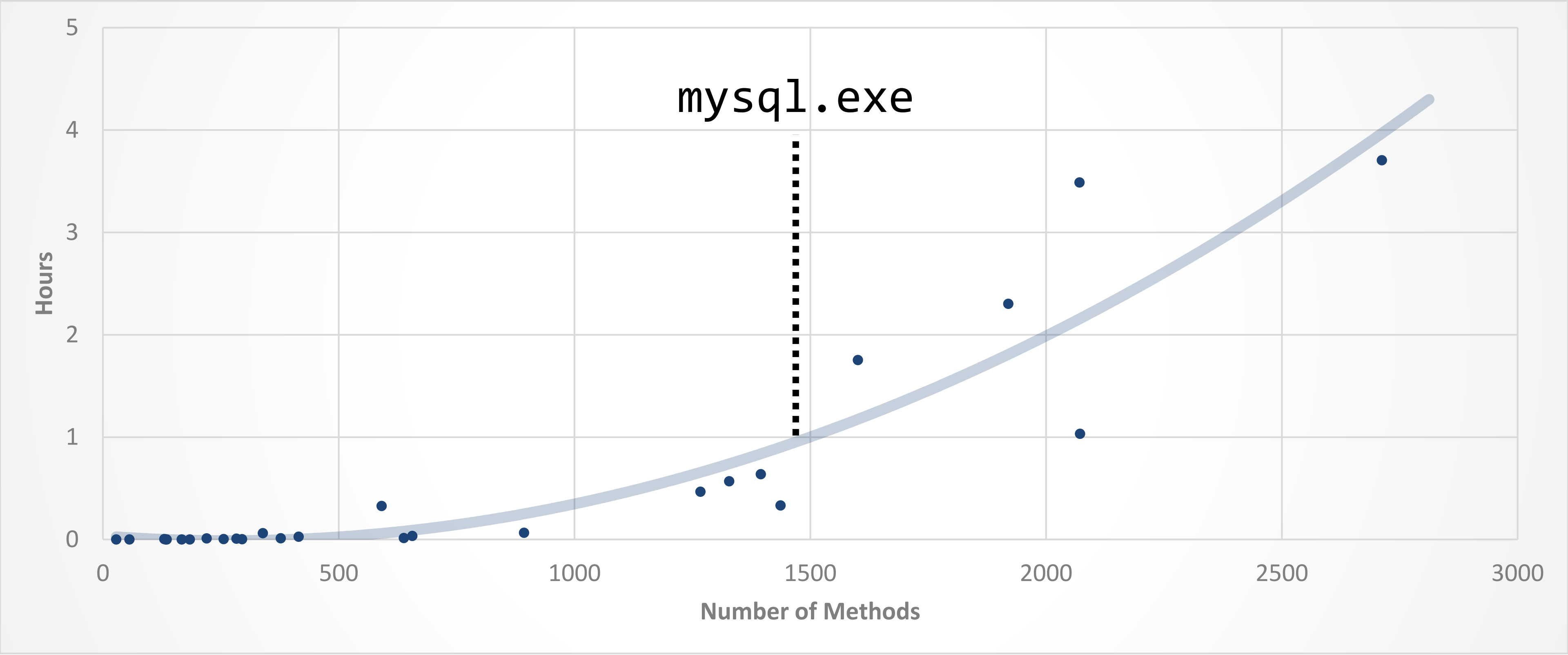
analysts and automated algorithms alike. For example, an algorithm searching for *use-after-free* vulnerabilities requires knowledge of object constructors [7], and an analyst attempting to understand a malware sample's behavior would greatly benefit from knowing which methods are on related classes [9]. Researchers have also demonstrated that many exploit protections are more effective with C++ abstractions, and that the level of protection and efficiency improves with the accuracy of the C++ abstractions. For example, researchers in executable-level control-flow integrity (CFI) protection systems [1, 35] have recently shown that the overall level of protection against exploits can be significantly improved by incorporating knowledge of C++ abstractions [8, 19, 21, 34]. Although there are existing systems that can recover C++ abstractions from executables, most of them rely on virtual function tables (vftables) as their

ACM CCS 2018

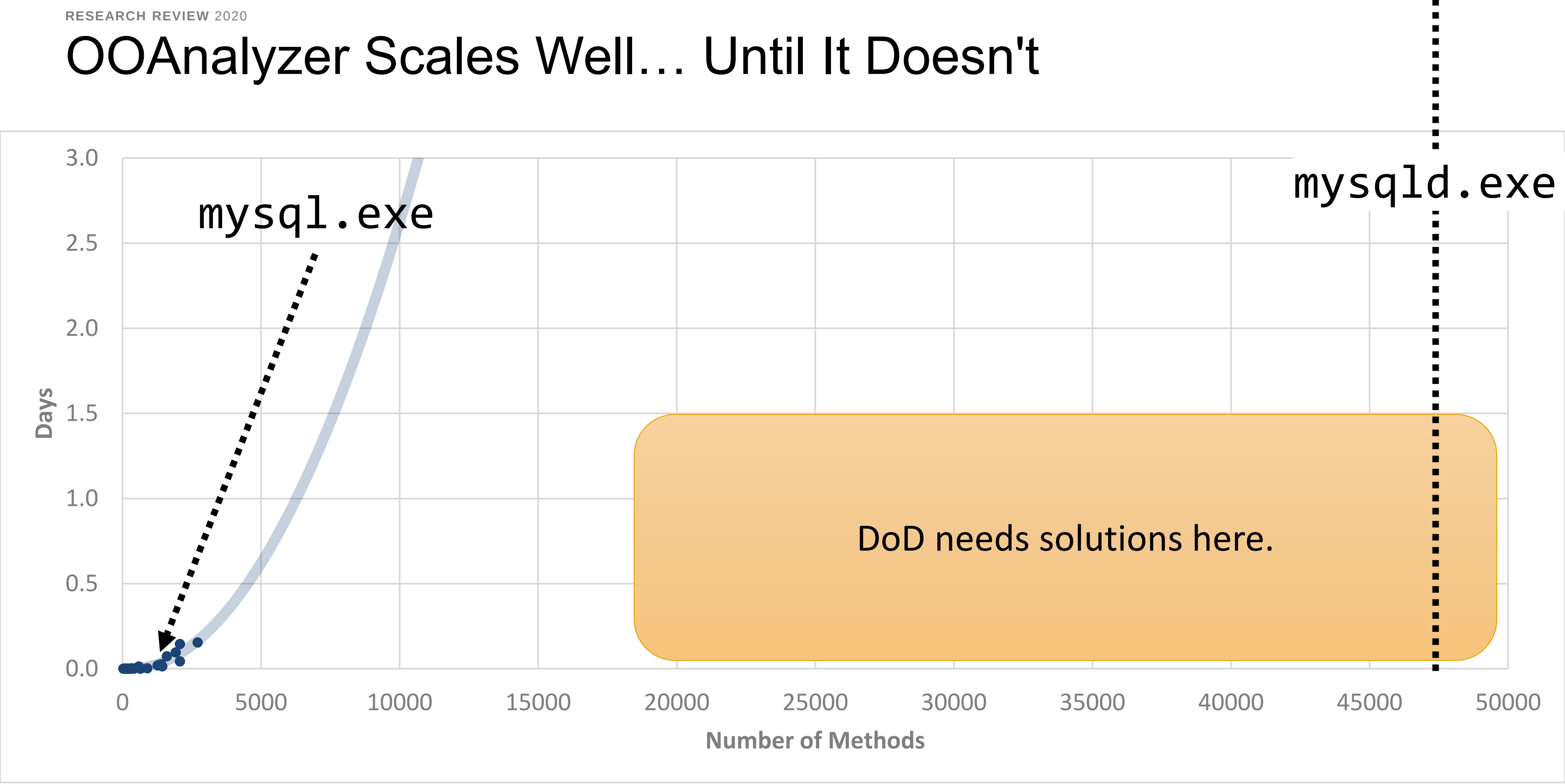
OOAnalyzer Scales Well…



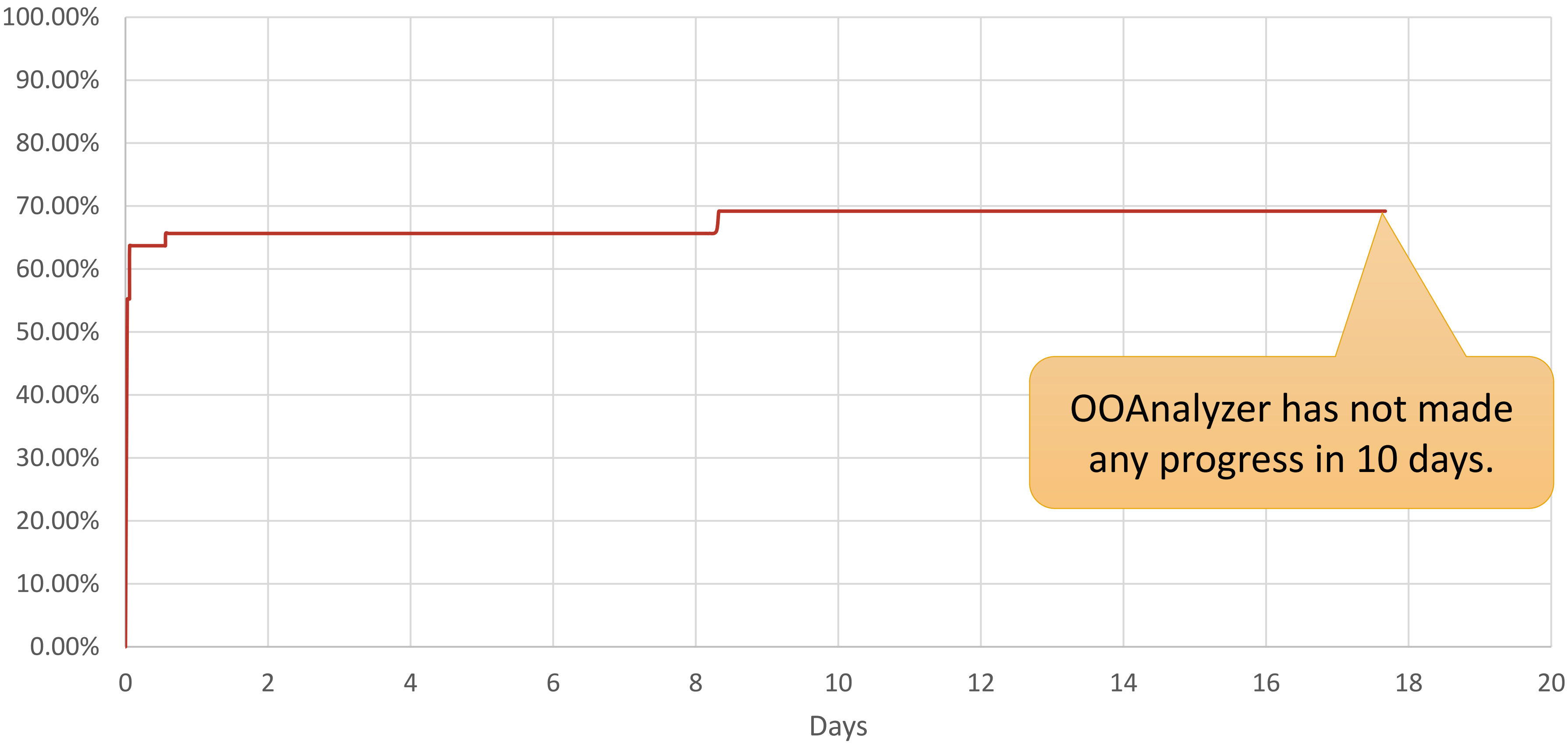
OOAnalyzer Scales Well… Until It Doesn't



OOAnalyzer Scales Well… Until It Doesn't



OOAnalyzer on mysqld.exe



OOAnalyzer was too slow to be used on the programs that the DoD needs it for the most.

Improving Performance

OOAnalyzer relies on incremental tabling

- Memoization for Prolog
  - If P➔Q and P does not change, Q will not change
- Dramatically speeds up performance
- OOAnalyzer originally used XSB Prolog
  - Robust, mature tabling support

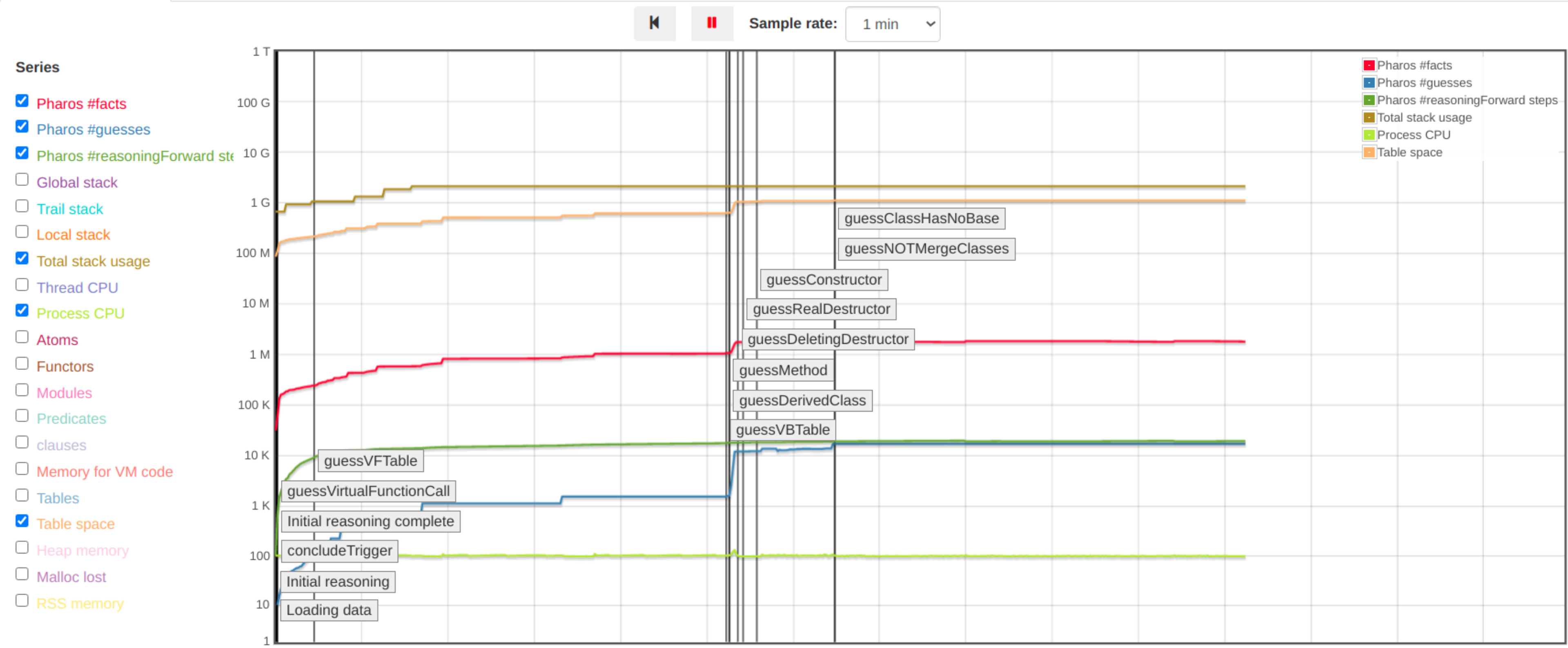
We worked with developers of XSB Prolog to add tabling support to SWI Prolog

- With OOAnalyzer as a test case ☺

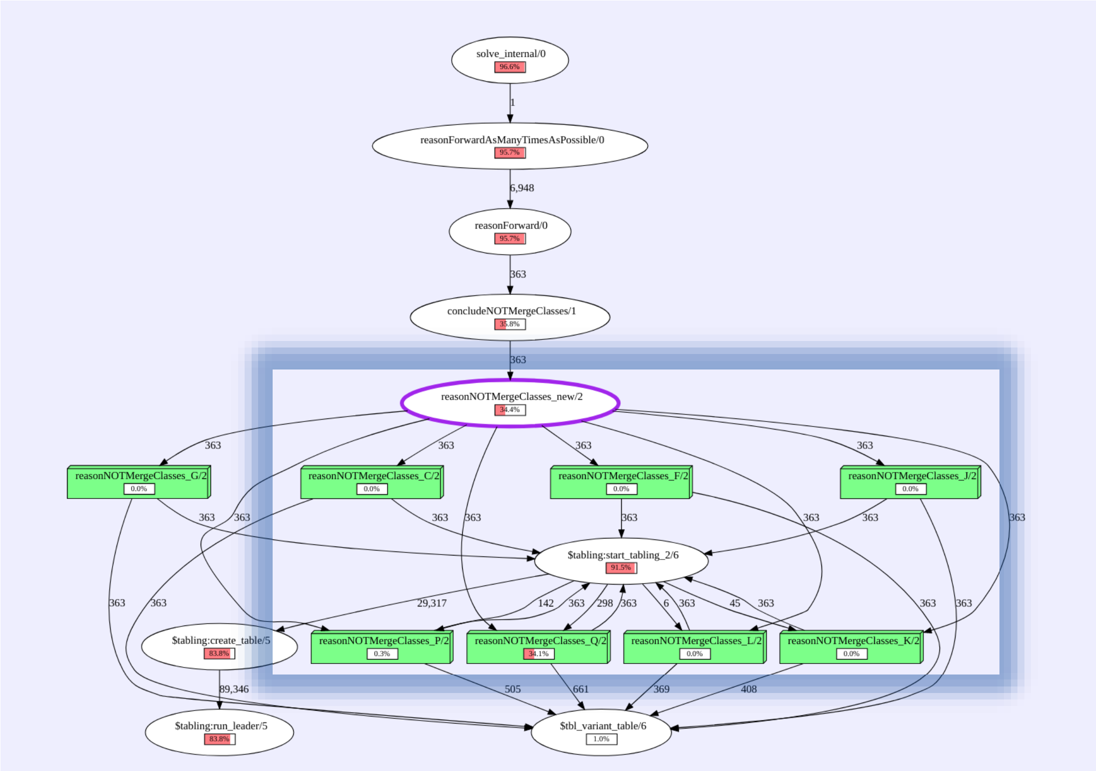
SWI Prolog advantages

- Substantially faster than XSB
- Provides invaluable debugging and profiling tools

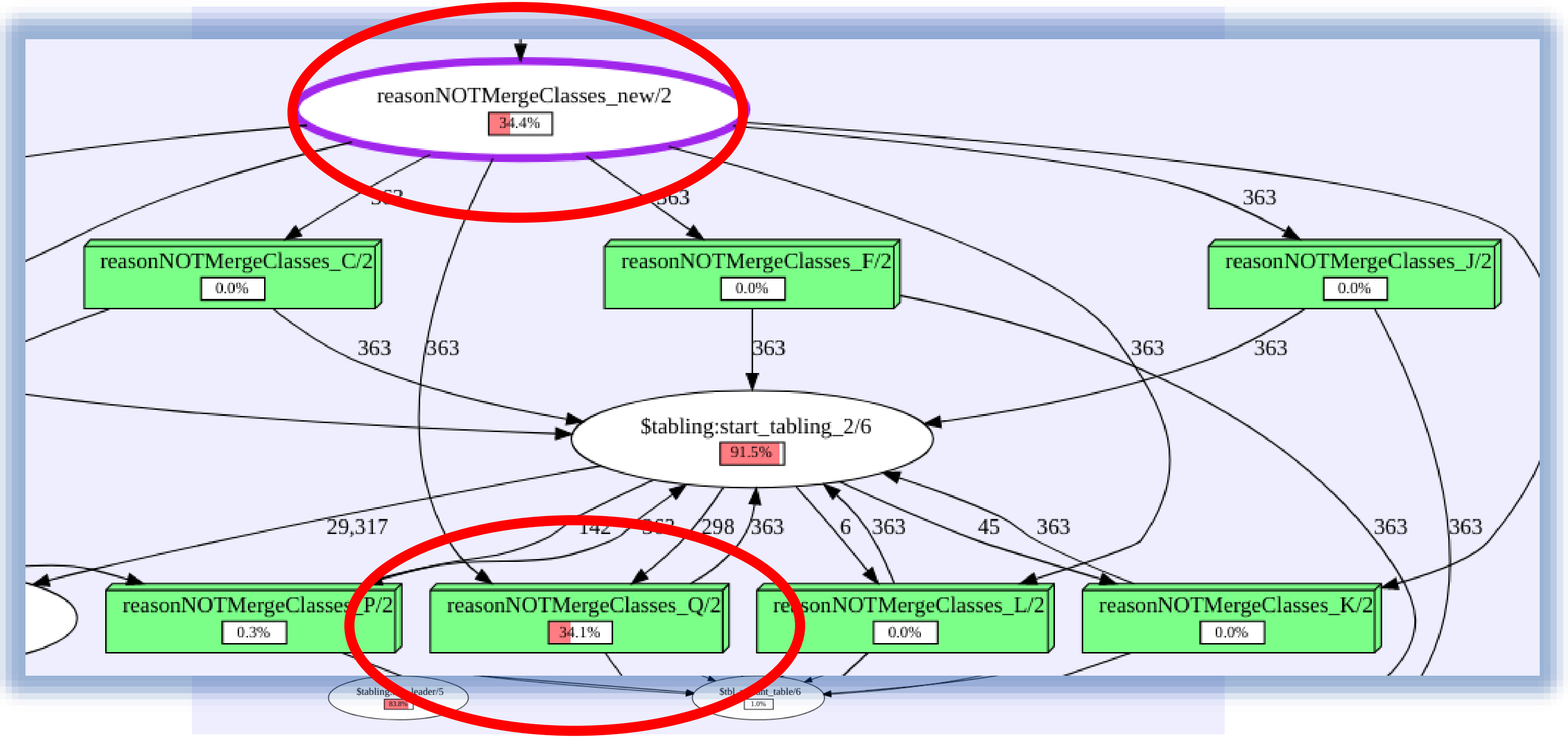
SWI Profiling: Resource Timeline



SWI Detailed Profiling



SWI Detailed Profiling



Fixing Performance Bottlenecks

Some performance problems were caused by simple mistakes.

Some can be fixed by reordering clauses.

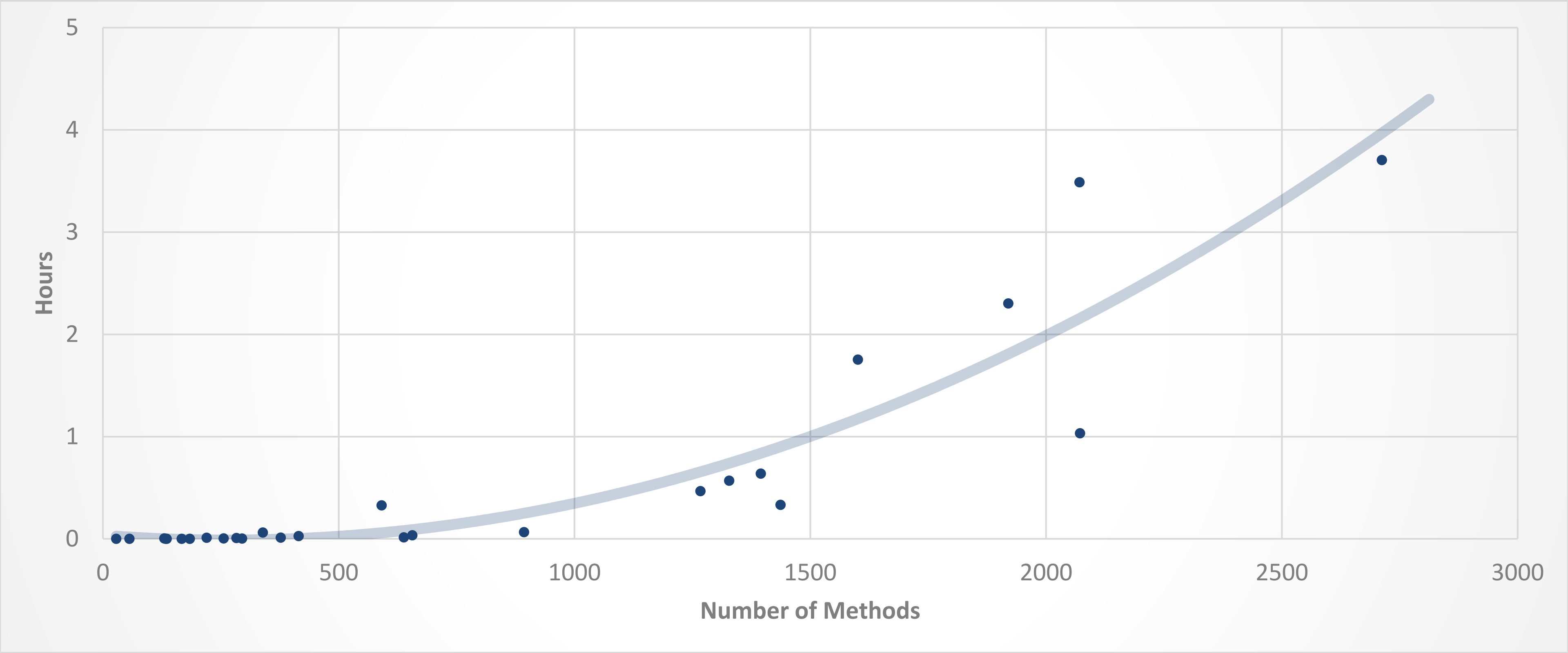
But we also discovered a systemic problem:

- Rules do not need to be recomputed if no dependent fact changes. ☺
- Entire rule needs to be recomputed when a dependent fact changes. ☹
- Some rules are expensive (n^2) to recompute.
  - More facts to consider ➔ More time
  - Becomes slower over time

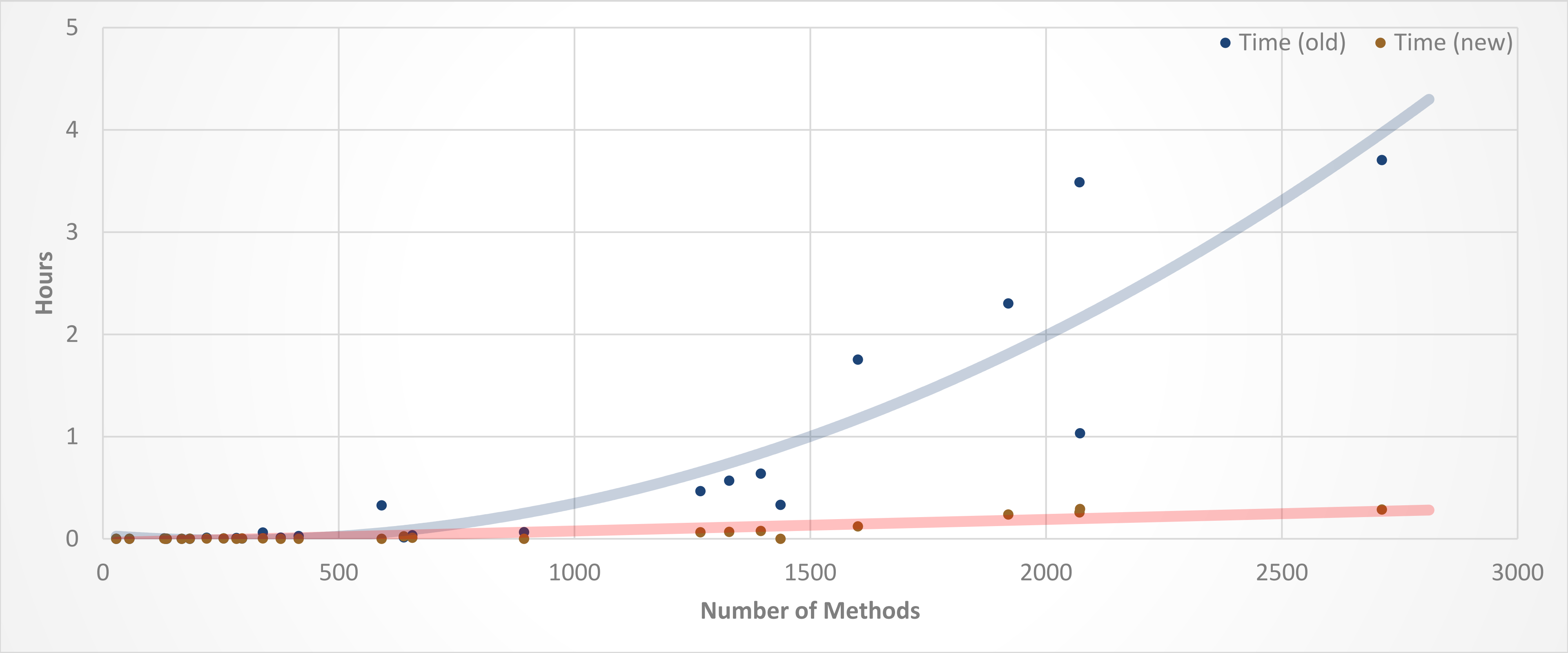
Insight: Most rules in OOAnalyzer are monotonic.

- They only need to be recomputed for "new" facts.
- Inspired development of monotonic tabling in SWI Prolog

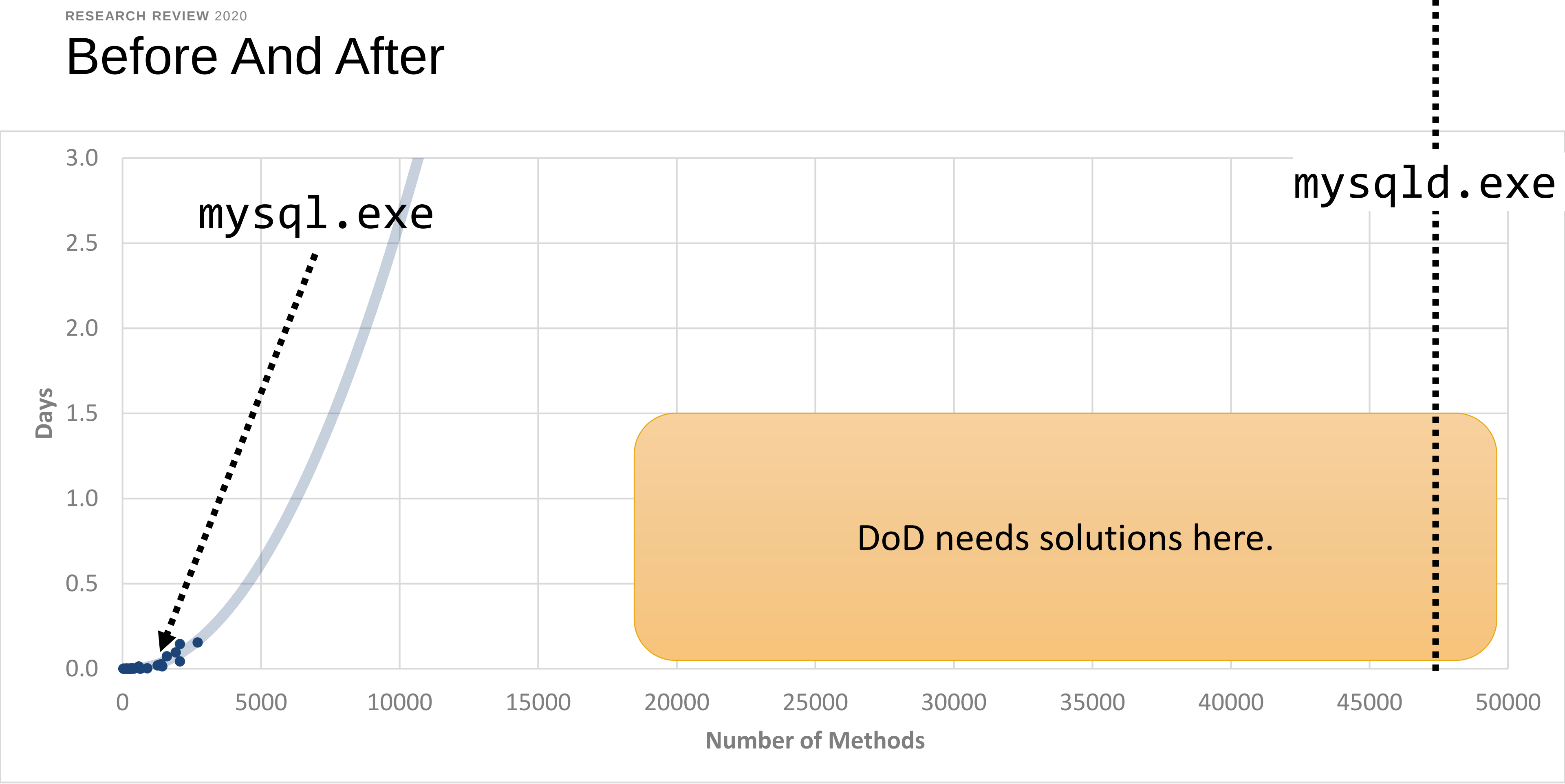
Before and After



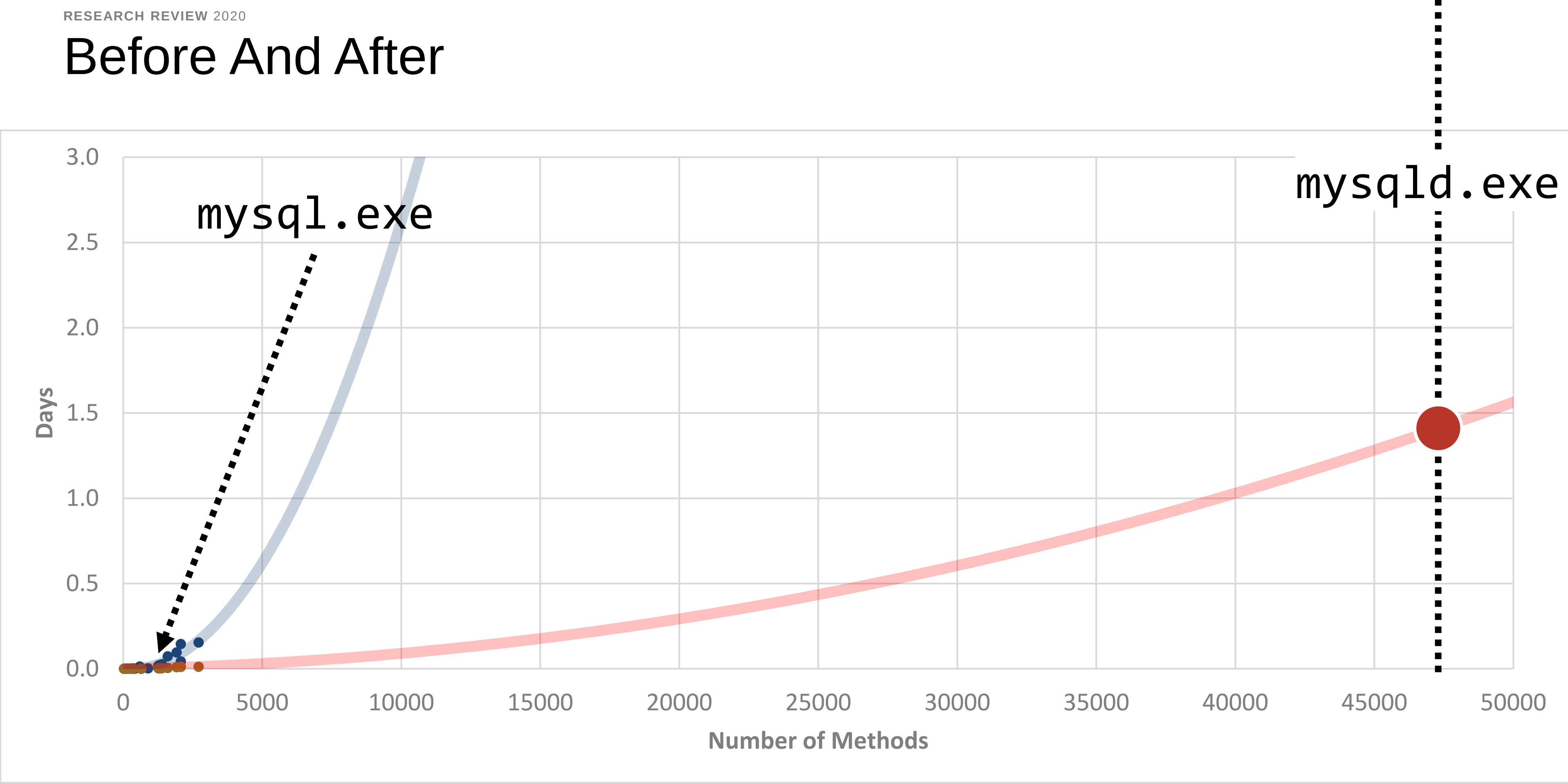
Before and After



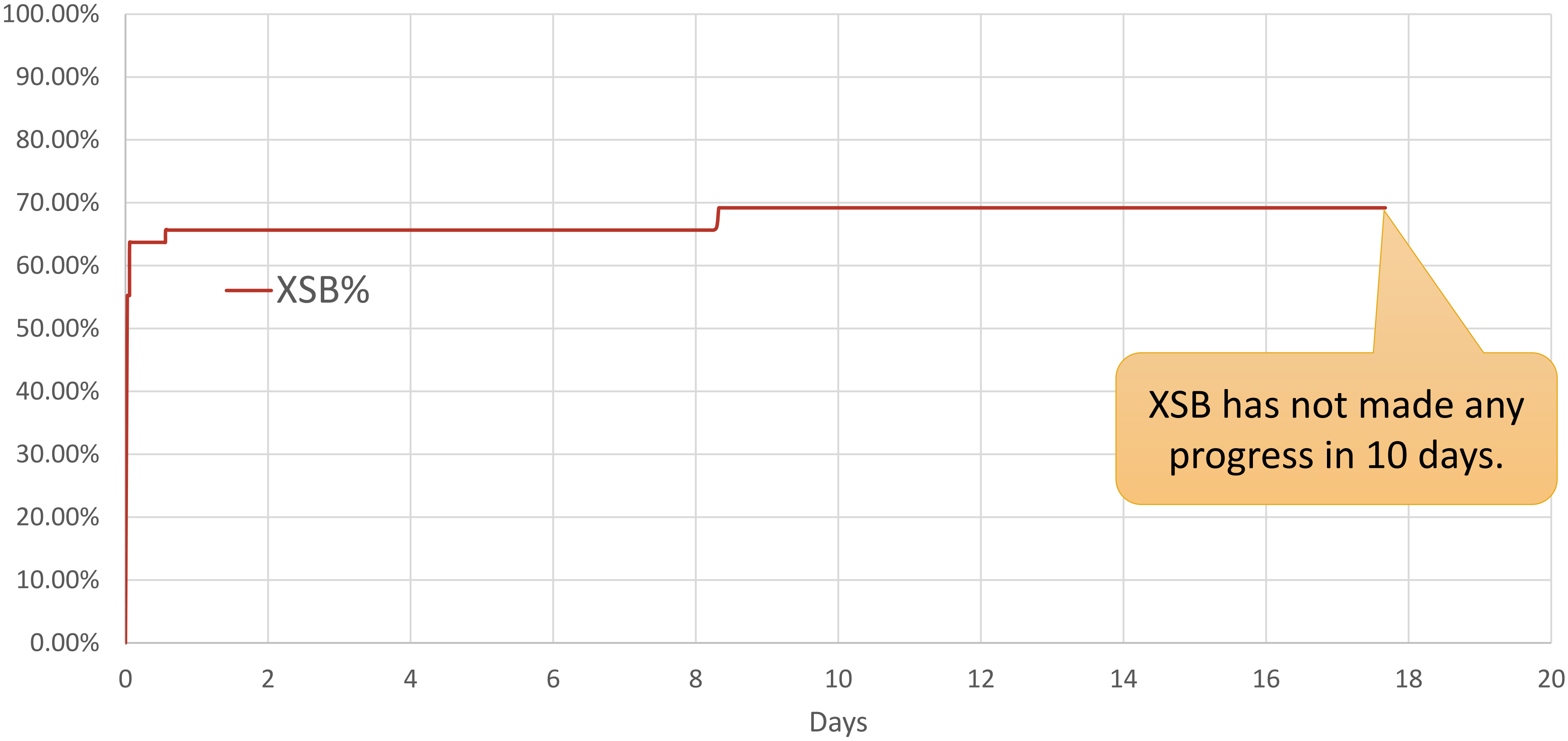
Before And After



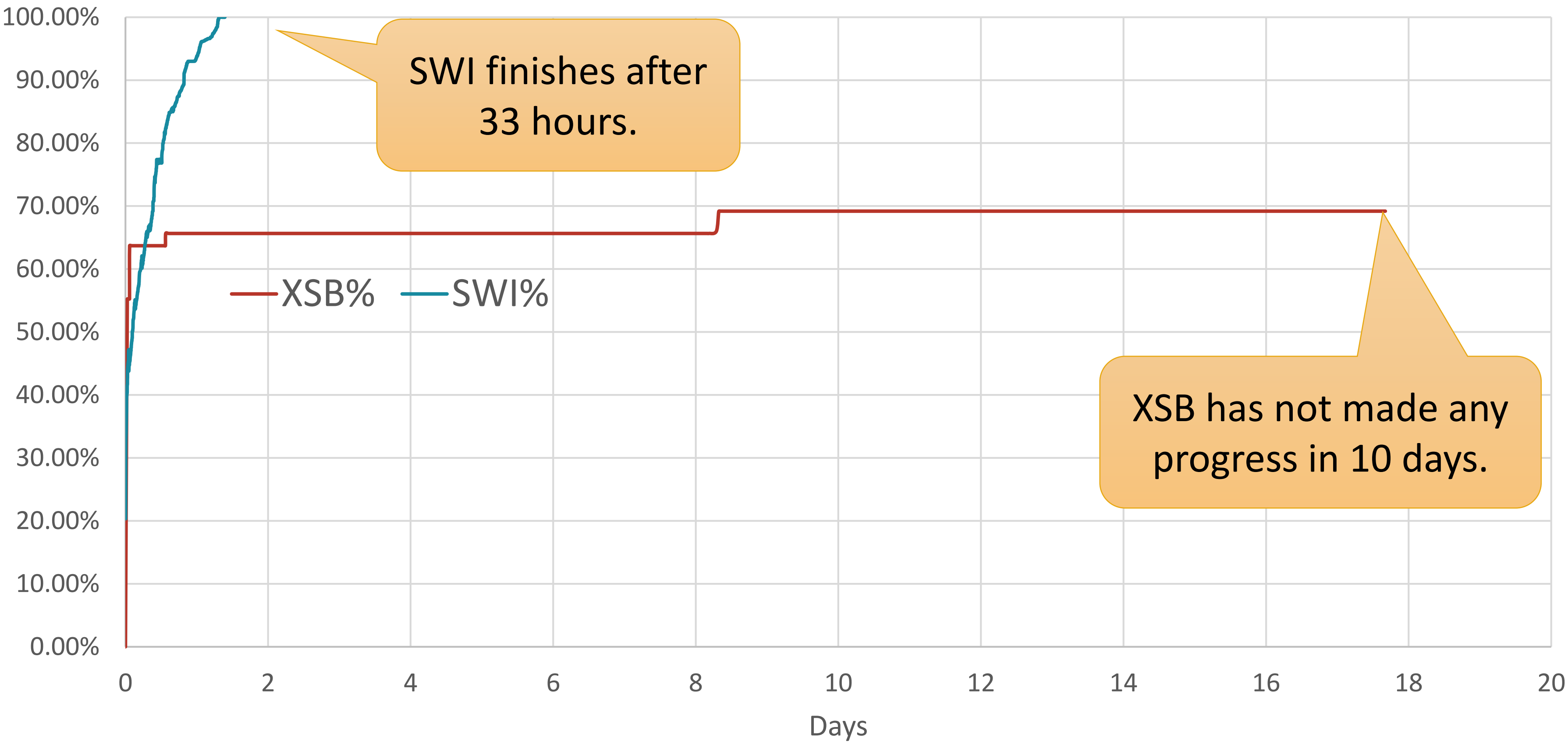
Before And After



Before and After on mysqld.exe



Before and After on mysqld.exe



Program Reachability	Variable Name Recovery	OOAnalyzer
Early Research	Near Transition	Fully Transitioned
2,184 test configurations found several successful approaches, but **none** that **consistently outperformed** the others, suggesting that a **hybrid approach** is needed.	We can **exactly** predict **74.3%** of variable names in decompiled executable code by training a neural network on a large corpus of C source code from GitHub.	OOAnalyzer was too slow to be used on the programs that the DoD needs it for the most. It is now **50x** faster and can analyze large programs.

https://github.com/cmu-sei/pharos

Team Members

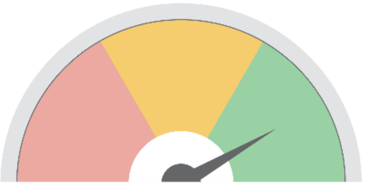
Cory Cohen

Dr. Edward Schwartz

END OF PRESENTATION

Null Function Abstraction: Simplify!

Accuracy = Poor



Speed = Good

Key observation: Some functions don't matter!

Replace those functions with null semantics or a greatly simplified representation.

Why bog down the SMT solver with irrelevant constraints?

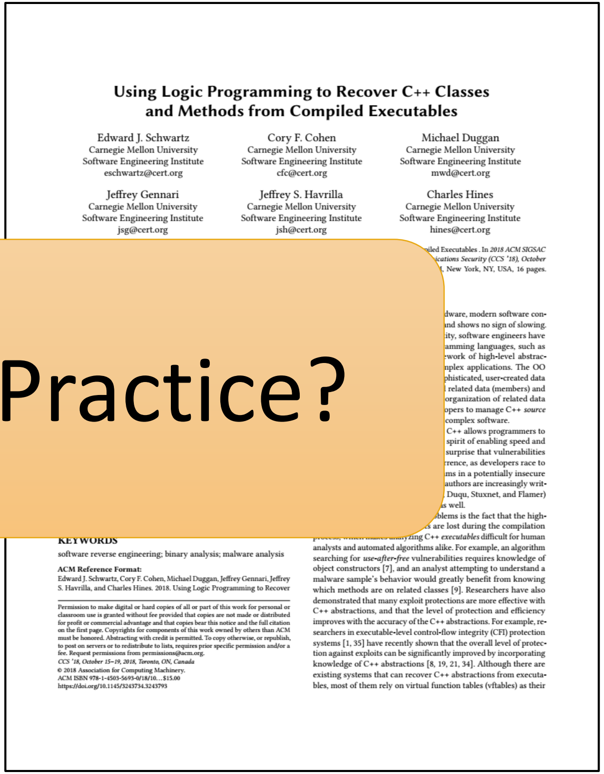
Accuracy	Speed	
		Irrelevant functions are removed entirely or simplified greatly.
		This approach can be used in combination with other approaches.

OOAnalyzer is the State of the Art in Research

- Static
  - Analyze program without executing it
  - No need for test cases
  - C…
- Ta…
  - …

Is It the State of the Art in Practice?

- Recovers **67-84%** of class abstractions correctly
  - Existing work recovers **<50%** of class abstractions correctly
  - Most existing work only attempts to recover virtual classes (because they are easier)


Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables

Edward J. Schwartz, Carnegie Mellon University, Software Engineering Institute, eschwartz@cert.org
Cory F. Cohen, Carnegie Mellon University, Software Engineering Institute, cfc@cert.org
Michael Duggan, Carnegie Mellon University, Software Engineering Institute, mwd@cert.org
Jeffrey Gennari, Carnegie Mellon University, Software Engineering Institute, jsg@cert.org
Jeffrey S. Havrilla, Carnegie Mellon University, Software Engineering Institute, jsh@cert.org
Charles Hines, Carnegie Mellon University, Software Engineering Institute, hines@cert.org

KEYWORDS
software reverse engineering; binary analysis; malware analysis

ACM Reference Format:
Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. 2018. Using Logic Programming to Recover

ACM CCS 2018

ObjDigger vs. OOAnalyzer Edit Distances on Cleanware

Program	# Class	# Method	ObjDigger Edits	ObjDigger Edits (%)	OOAnalyzer Edits	OOAnalyzer Edits (%)
Firefox.exe	141	638	507	79.5%	212	33.2%
Log4cpp Debug	139	893	829	92.8%	239	26.8%
Log4cpp Release	76	378	272	72.0%	75	19.8%
muParser Debug	180	1437	1361	94.7%	483	33.6%
muParser Release	94	598	369	61.7%	183	30.6%
MySQL cfg_editor.dll	190	1266	∞	∞	391	30.9%
MySQL mysql.exe	202	1395	∞	∞	439	31.5%
TinyXML Debug	35	415	268	64.6%	69	16.6%
TinyXML Release	33	283	174	61.5%	55	19.4%

OOAnalyzer recovers 67% to 84% of methods on cleanware programs.

ObjDigger vs. OOAnalyzer Edit Distances on Malware

Program	# Class	# Method	ObjDigger Edits	ObjDigger Edits (%)	OOAnalyzer Edits	OOAnalyzer Edits (%)
Malware 0faaa3d3	21	135	121	89.6%	21	15.6%
Malware 29be5a33	19	130	91	70.0%	15	11.5%
Malware 6098cb7c	55	339	131	38.6%	29	8.6%
Malware 628053dc	207	1920	1245	64.8%	378	19.7%
Malware 67b9be3c	400	2072	1299	62.7%	670	32.3%
Malware cfa69fff	39	184	125	67.9%	37	20.1%
Malware d597bee8	19	133	68	51.1%	17	12.8%
Malware deb6a7a1	283	2712	1900	70.1%	639	23.6%
Malware f101c05e	169	1601	987	61.6%	329	20.5%

OOAnalyzer recovers 68% to 91% of methods on smaller malware samples.

OOAnalyzer Method Classification on Cleanware

Program	Constructors			Destructors			Virtual Function Tables			Virtual Methods		
	Recall	Prec.	F	Recall	Prec.	F	Recall	Prec.	F	Recall	Prec.	F
Firefox.exe	40/51	40/54	0.76	1/39	1/1	0.05	18/33	18/18	0.71	85/101	85/98	0.85
Log4cpp Debug	192/209	192/197	0.95	40/118	40/40	0.51	18/18	18/18	1.00	84/101	84/86	0.92
Log4cpp Release	135/165	135/170	0.81	24/73	24/36	0.44	18/21	18/18	0.92	84/101	84/86	0.90
muParser Debug	293/325	293/314	0.92	28/156	28/30	0.30	12/12	12/13	0.96	35/47	35/43	0.78
muParser Release	197/252	197/269	0.76	15/91	15/21	0.27	12/14	12/13	0.89	35/47	35/37	0.83
MySQL cfg_editor.dll	260/290	260/311	0.87	107/281	107/111	0.55	69/69	69/69	1.00	321/427	321/325	0.85
MySQL mysql.exe	282/314	282/341	0.86	115/300	115/121	0.55	75/75	75/75	1.00	341/453	341/345	0.85
TinyXML Debug	53/60	53/57	0.91	0/39	0/3	0.00	24/24	24/24	1.00	101/119	101/102	0.91
TinyXML Release	49/60	49/53	0.87	27/39	27/36	0.72	24/24	24/24	1.00	101/119	101/103	0.91

Precision: How many were found? Recall: Were they correct? F-measure: A harmonic mean.

Some problems with destructor identification, but quite good in other areas

OOAnalyzer is The State Of The Art

OOAnalyzer is The State Of The Art

… in Research

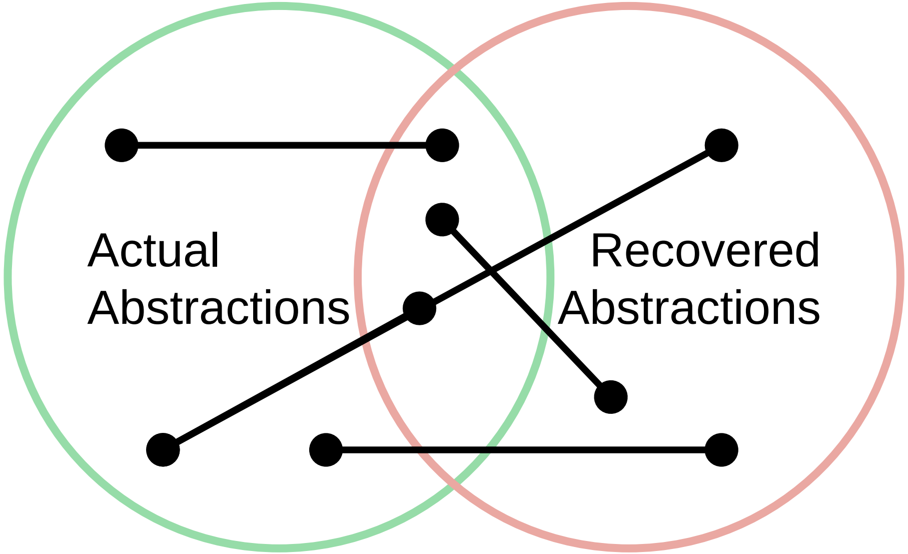
How Can We Measure Accuracy?

Measuring the accuracy of the recovered C++ abstractions has been very difficult.

There are
- multiple correct answers
- nearly infinite incorrect answers
- many partially correct answers

Solution: **Edit distances** - compute the number of changes required to transform our answer into the correct answer.

Smaller **edit distances** are better!



How Can We Measure Accuracy?

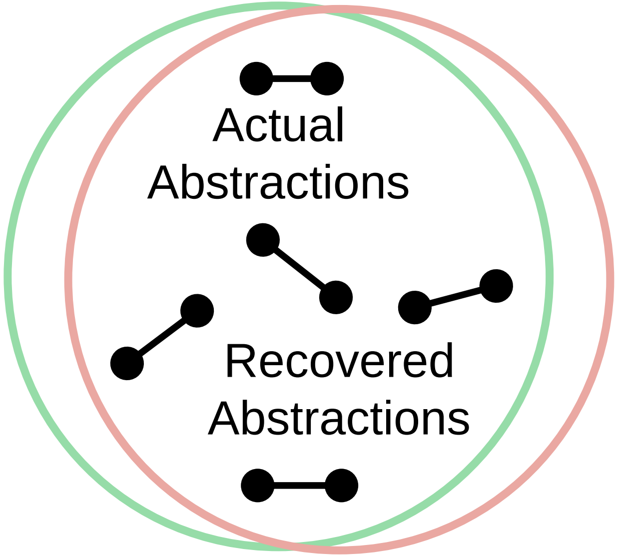
Measuring the accuracy of the recovered C++ abstractions has been very difficult.

There are:

- multiple correct answers
- nearly infinite incorrect answers
- many partially correct answers

Solution: **Edit distances** - compute the number of changes required to transform our answer into the correct answer.

Smaller **edit distances** are better!



Are we going to introduce ObjDigger?

- Cory could use the first few slides from my CCS talk
- Alternative is to remove ObjDigger results, but then there is nothing to compare to
- Another alternative is simply to summarize results without tables
  - OOAnalyzer recovers X% …

OOAnalyzer is the State of the Art in Research

- Static
  - Analyze program without executing it
  - No need for test cases
  - Can be used on unknown software (malware)

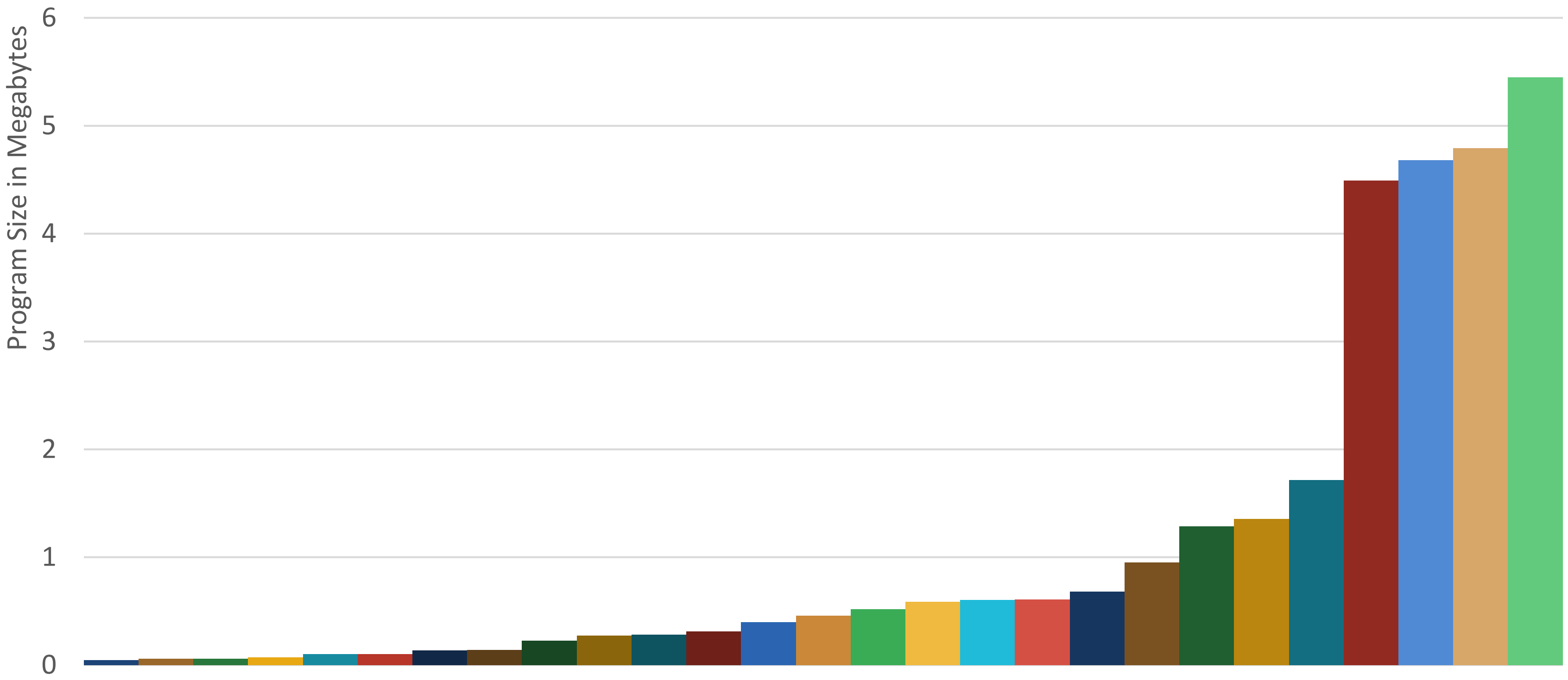
- Targets all classes and all methods
  - Existing work focuses on virtual classes/functions (because they are easier)

- Recovers **67-84%** of class abstractions correctly
  - Existing work recovers **<50%** of class abstractions correctly
  - Most existing work only attempts to recover virtual classes (because they are easier)

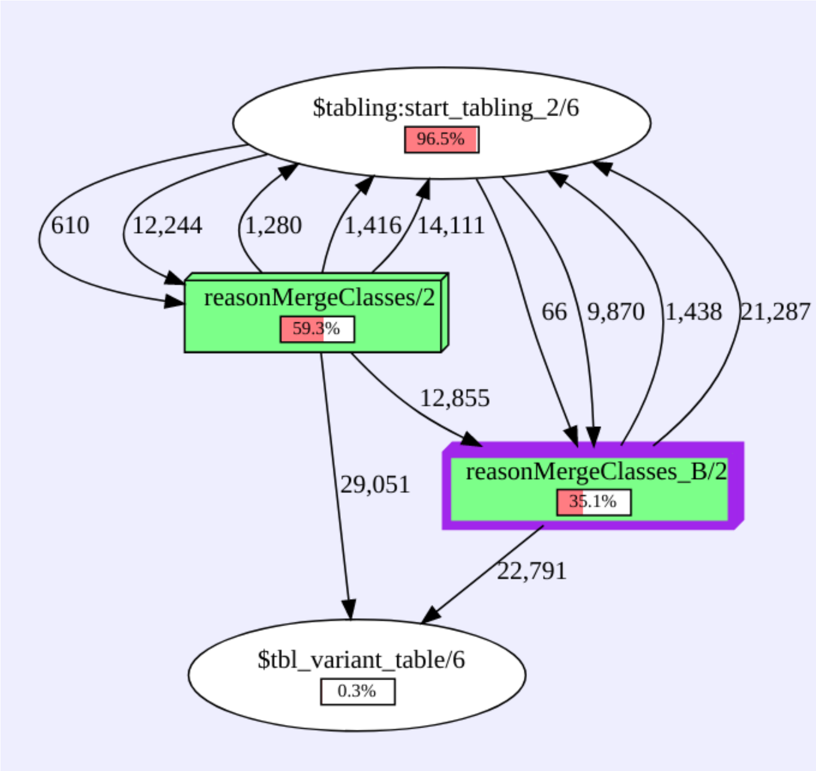
Research vs Practice

- Larger programs take longer to analyze ➔ Automation is more valuable on larger programs
- Prolog makes for a nice academic story
  - But does it actually scale?
- Prolog scales… up to a point

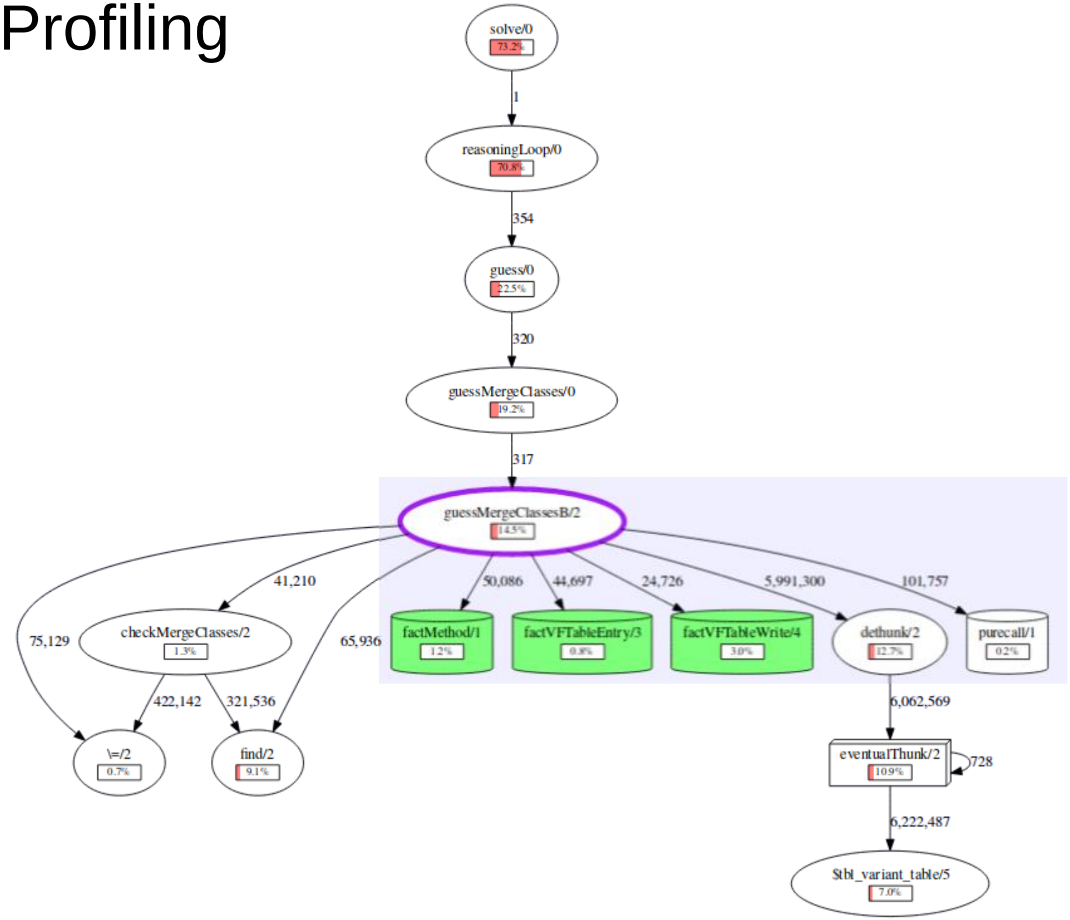
We Originally Looked at a Few Medium Sized Programs … and a Lot of Small Programs



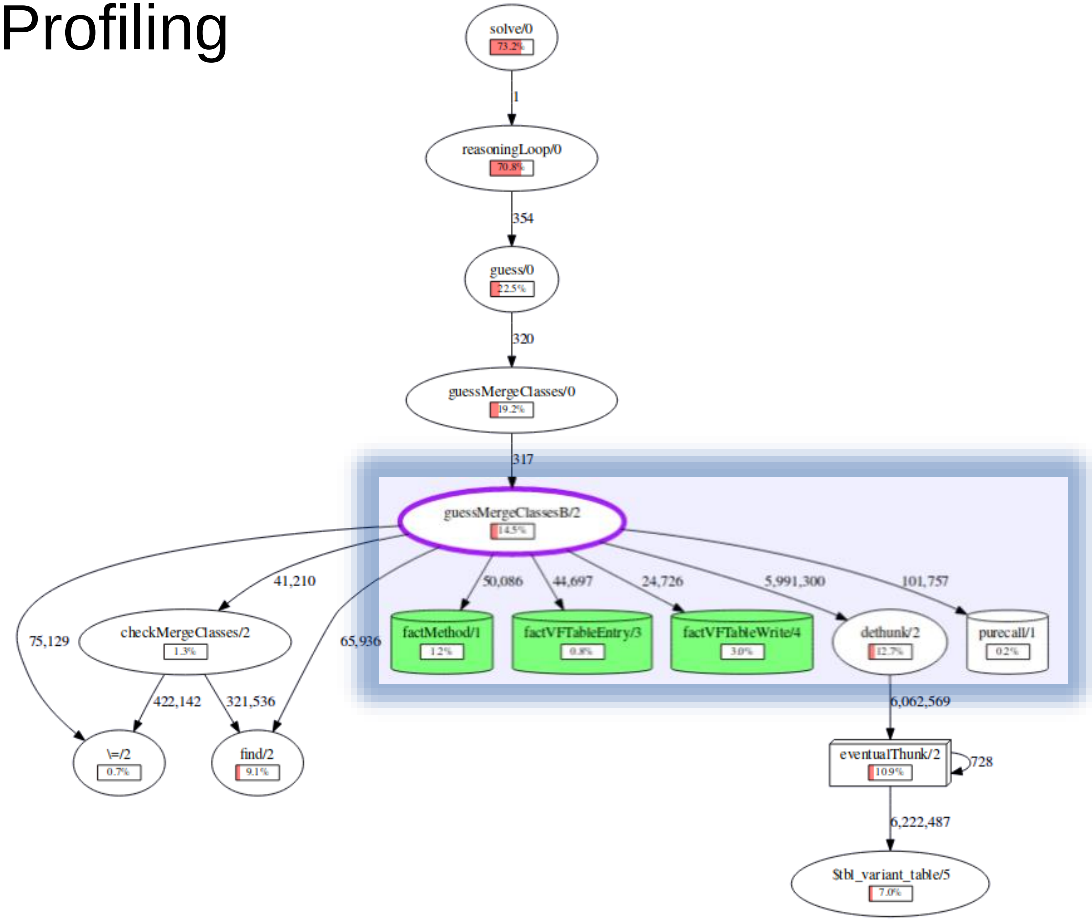
Different screenshot



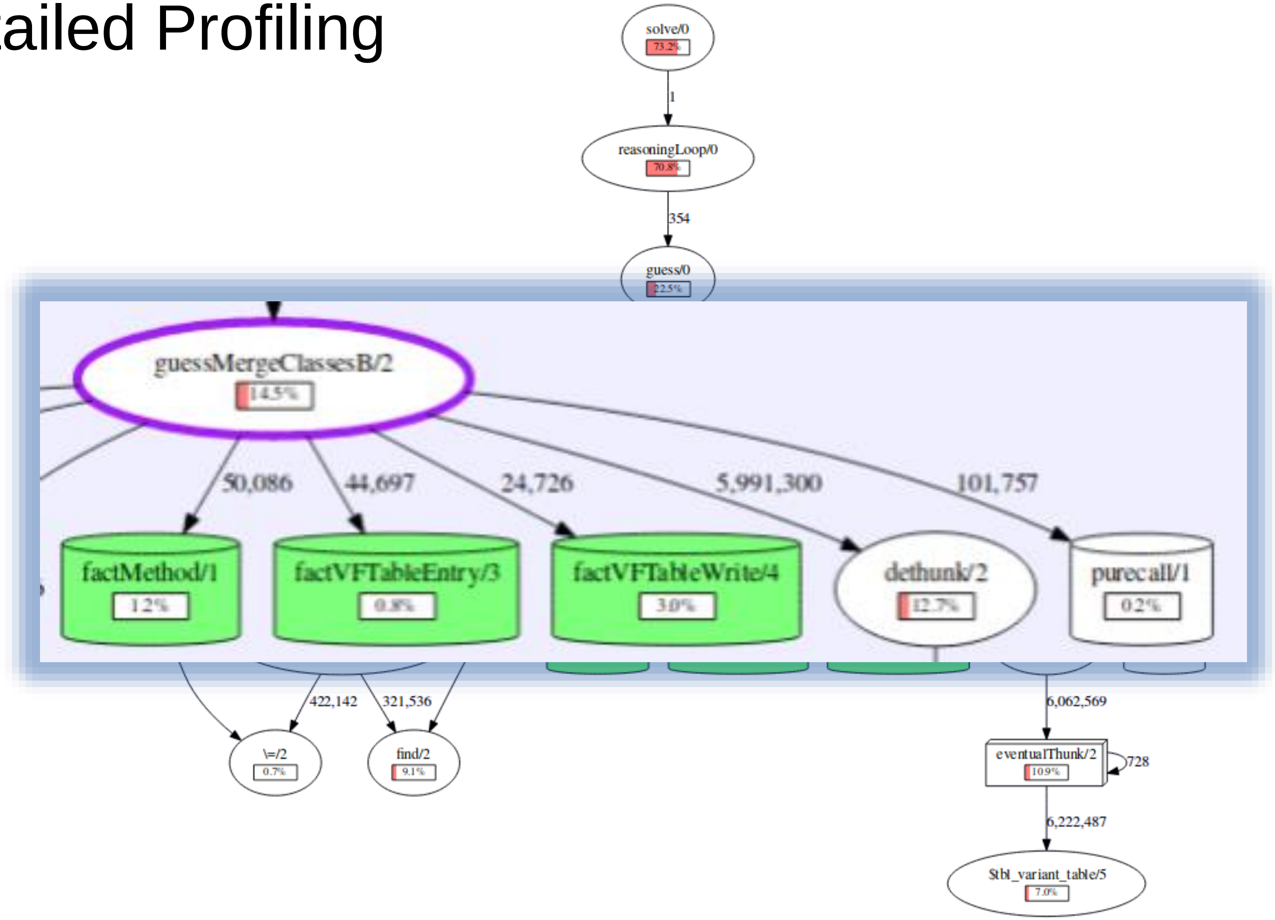
SWI Detailed Profiling



SWI Detailed Profiling



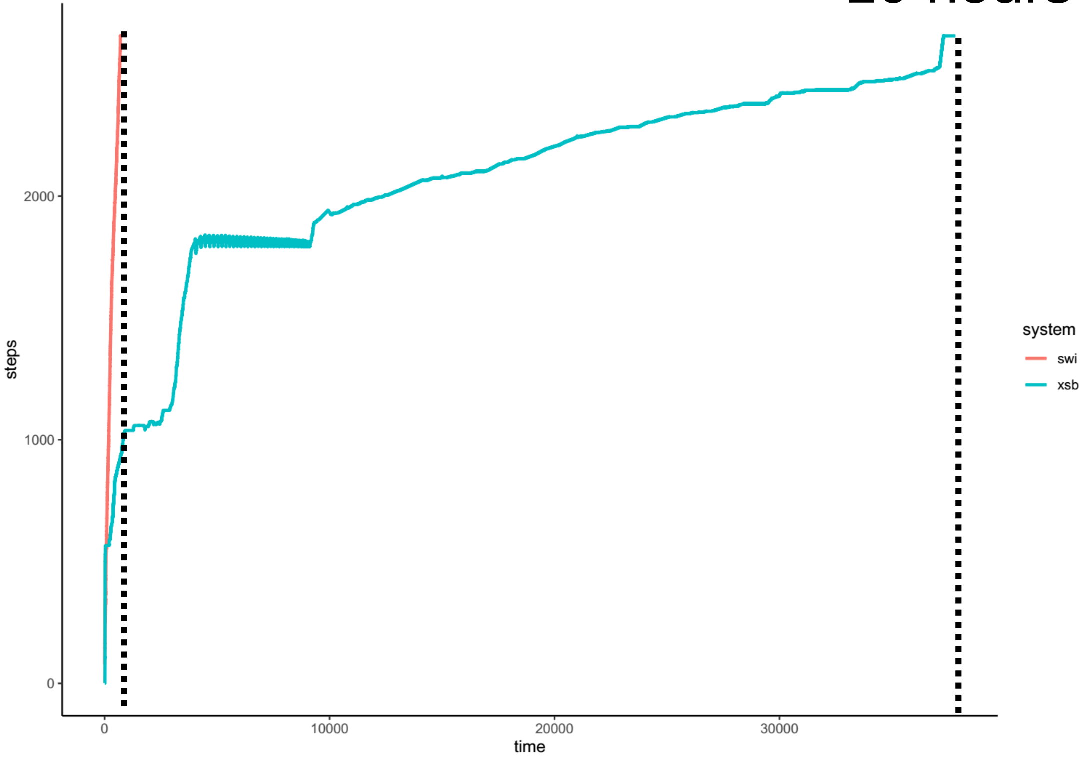
SWI Detailed Profiling



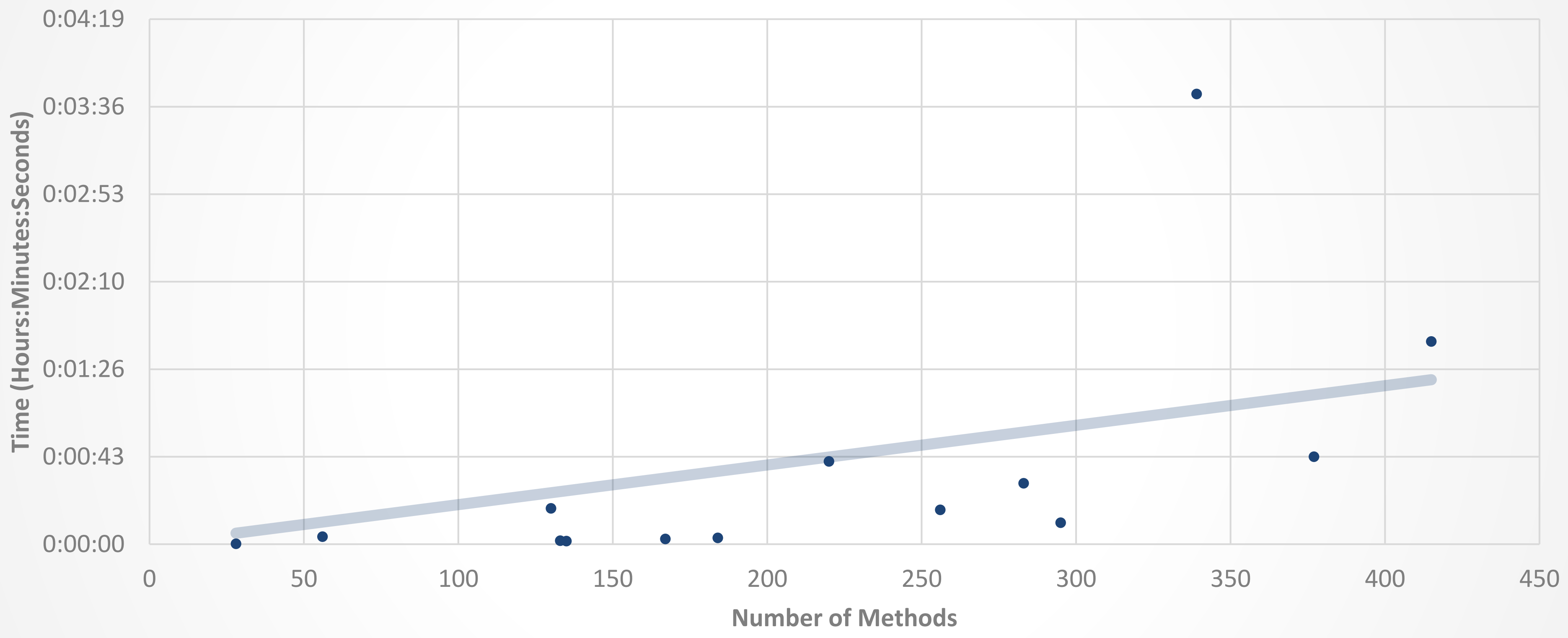
mysql_upgrade.exe

11 minutes

10 hours

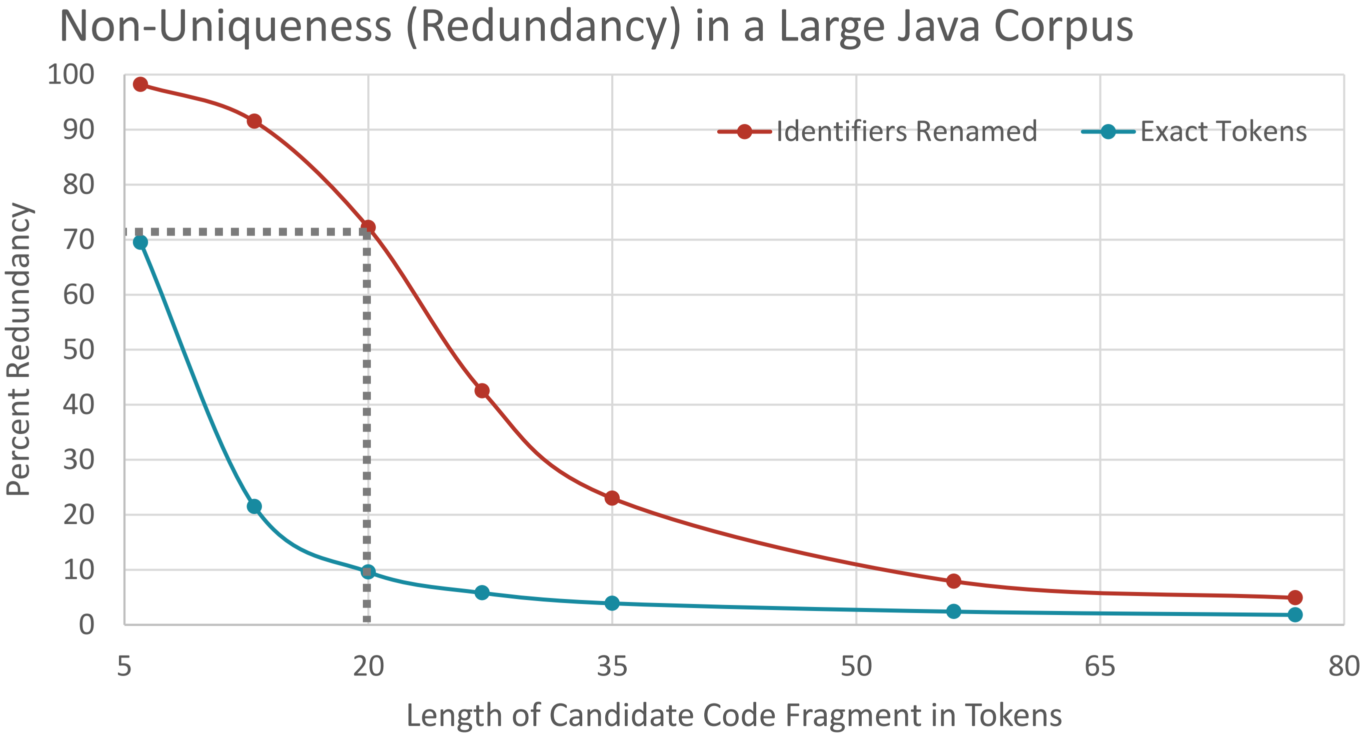


OOAnalyzer Scales Well…



Software is really repetitive

Gabel & Su, 2010



Transitioning from Research to Practice

Research was a proof of concept

- Python command line tools that are difficult to use
- Now implemented as a Hex-Rays Plugin for easy use

Model insufficient for use in practice

- One compiler (`gcc`)
- One optimization level (`-O0`)
- One architecture (x86-64)
- We are training a model that operates in more realistic environments

NSA Ghidra Integration to Display C++ Decompilation

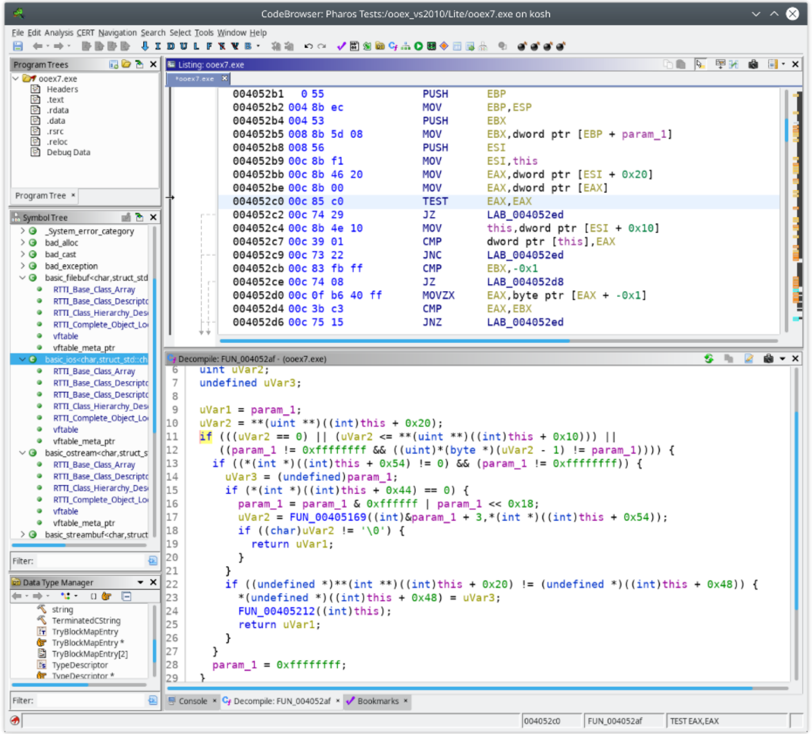
Integrates OOAnalyzer abstractions into NSA's Ghidra software reverse engineering tool

- Integrates with symbols and types
- Improves decompiler
- Eases transition

Plugin significantly overhauled

- Testing with large programs
- Progress reporting during import
- Automatic builds for Ghidra versions

Also available for IDA Pro



Fixing Performance Bottlenecks

Trigger rules

- If there is a new fact *F*, what conclusions *C* can be made using rule *R* that could not be made previously?
- No need for recomputation ☺
- Manually written/analyzed ☹

Moving toward automation

- Manual effort is tedious and error-prone
- Inspired monotonic tabling in SWI Prolog