Advancing Cyber Operator Tradecraft through Automated Static Binary Analysis

Cory Cohen & Dr. Edward Schwartz
Copyright 2020 Carnegie Mellon University.

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.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.


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

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OO Analyzer</strong></td>
<td>Detects object-oriented constructs, resolves virtual function calls</td>
<td>Greatly reduces the malware analysis effort required for deep understanding of malware capabilities</td>
</tr>
<tr>
<td><strong>Call Analyzer</strong></td>
<td>Reports constant parameters to calls in binary executables</td>
<td>Permits analyst to identify parameters to important operating system API calls to detect undesired behaviors in software</td>
</tr>
<tr>
<td><strong>FN2Yara</strong></td>
<td>Automatically generates YARA signatures</td>
<td>Promotes high-quality signatures to detect similarity in malware families, which can be converted to Snort signatures for use in network defense</td>
</tr>
<tr>
<td><strong>FN2Hash</strong></td>
<td>Generates function hashes to identify functions in malware files</td>
<td>Reduces analyst time spent doing repetitive tasks, automates identification of functions of interest in malware</td>
</tr>
<tr>
<td><strong>Malware Design Matcher</strong></td>
<td>Detects high-level design abstractions in malware files</td>
<td>Automated identification of key abstractions in known families, permits human analysts to record abstract knowledge precisely</td>
</tr>
<tr>
<td><strong>Api Analyzer</strong></td>
<td>Detects patterns of API calls representing malicious behaviors</td>
<td>Focuses analyst attention on important aspects of code via automated analysis, detects unexpected patterns for software assurance</td>
</tr>
</tbody>
</table>
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
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

**Under what conditions does the malicious code execute?**

```
cmp ecx, 76h
jge exit

cmp ecx, 42h
jnz normal

call check
cmp eax, 34h
jnz normal

inc ecx
jmp loop
```

**Z3 SMT Constraint Problem**

Satisfiable: ecx = 42h check() = 34h

Rewritten Executable Always Runs Code
# Evaluating Multiple Approaches/Implementations

<table>
<thead>
<tr>
<th>Pharos Function Summaries</th>
<th>Weakest Precondition</th>
<th>Property Directed Reachability (PDR)</th>
<th>Ghidra + Seahorn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completely remove or greatly simplify functions that are not important to improve performance.</td>
<td>Analyze function input and output states to minimize complexity for solver while being as accurate as possible.</td>
<td>Base analysis on complete symbolic behavior of instructions to increase accuracy.</td>
<td>A more source-code centric approach to resolving the problems presented by our early PDR attempts.</td>
</tr>
</tbody>
</table>

### Accuracy?  

### Scalability?
Pharos Function Summaries

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.

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Poor</td>
<td>Fair</td>
</tr>
</tbody>
</table>

- 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)

- 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?

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>🟢 🟡</td>
<td>🟢 🟡</td>
</tr>
<tr>
<td>Memory is represented precisely as a single large array.</td>
<td></td>
</tr>
<tr>
<td>Loops are unrolled, which is unable to prove some paths.</td>
<td></td>
</tr>
<tr>
<td>Efficient algorithm generates formulas that are linear in size.</td>
<td></td>
</tr>
</tbody>
</table>
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

We’re improving support for bit vectors and arrays.
Property Directed Reachability Approach (PDR)

- 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.

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Good" /></td>
<td><img src="image" alt="Poor" /></td>
</tr>
</tbody>
</table>

- Memory is represented precisely as a single large array.
- SPACER is able to reason about loops correctly but slowly.
Ghidra & Seahorn Approach

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?

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fair</td>
<td>Fair</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Test Case Configuration</th>
<th>Pharos Function Summaries</th>
<th>Weakest Precondition</th>
<th>Property Directed Reachability</th>
<th>Ghidra/Seahorn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized</td>
<td>Arch</td>
<td>Fail</td>
<td>Tout</td>
<td>Pass</td>
</tr>
<tr>
<td>None</td>
<td>32-bit</td>
<td>55</td>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>None</td>
<td>64-bit</td>
<td>47</td>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>Medium</td>
<td>32-bit</td>
<td>40</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>Medium</td>
<td>64-bit</td>
<td>53</td>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>High</td>
<td>32-bit</td>
<td>50</td>
<td>0</td>
<td>41</td>
</tr>
<tr>
<td>High</td>
<td>64-bit</td>
<td>32</td>
<td>1</td>
<td>58</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>257</td>
<td>3</td>
<td>266</td>
</tr>
</tbody>
</table>

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?
Advancing Cyber Operator Tradecraft through Automated Static Binary Analysis

Recovering Meaningful Variable Names in Decompiled Code
Disassembler

IDA - dd /media/DATA/DEBUG/debug/src/dd

Graph overview

xor edx, edx
div rdi
mov cs:skip_records, rax
mov cs:skip_bytes, rdx
cmp rbx, 0FFFFFFFFFFFFFFFFh
jnz short loc_40299C

loc_402996:
cmp rbx, 0FFFFFFFFFFFFFFFFh
jz short loc_4029C0

jmp short loc_4029C0

loc_4029C0:
mov eax, esi
and eax, 4
jz short loc_4029C0

loc_4029C0:
cmp rbx, 0FFFFFFFFFFFFFFFFh
jz short loc_4029CD
Decompiler

```
usage(1);
}

v8 = v7 + 1;

switch { _OR8__(*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, "invalid");
        goto LABEL_50;
    case 3:
        if (v6[1] != 102)
            goto LABEL_46;
        v12 = v6[2];
        if (v12 && v12 != 61)
            {
                    {
                        v13 = v6[5];
                        if (v13 || v13 == 61)
                            {
                                break;
                            }
                        }
                    }
            }
```
Decompiler

```c
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, "invalid");
        goto LABEL_90;
    case 3:
        if (v6[1] != 102)
            goto LABEL_46;
        v12 = v6[2];
        if (v12 && v12 != 61)
            {
                {
                v13 = v6[5];
                if (!v13 || v13 == 61)
                    {
                
```
The problem:

Decompilers are typically unable to assign meaningful names to variables.
Our Work

Decompiler output

```c
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;
}
```

Refactored decompiler output

```c
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

```
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;
}
```
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

Tiger, Tiger burning bright
In the forests of the night..
Natural Language

TIGER!!
RUN!!!
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,earl.barr,zu}@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, with all the attendant constraints and limitations, and thus 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, memory, and secondary storage) to store and manipulate.

(Presented at the 2012 International Conference on Software Engineering)

Software is really repetitive

Gabel & Su, 2010

70% of sequences longer than 20 tokens have been written elsewhere.
How can we use this?
Learn typical variable names in a given context from examples … many, many examples.

If software is repetitive, so are names.

```c
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

Encoder

- Lexical Encoder (LSTM)
- Structural Encoder (GGNN)

Decoder

- Code Element Representations
- Identifier Representations
- Sequential Decoder with Attention
How good are the renamings?
Assumption: Original (human-written) names are good.

How many can we recover?
The Amount of Training Data Matters

Accuracy

Size of Training Set

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%

1% 10% 100%

74%
The Uniqueness of Data Matters

![Graph showing accuracy of data uniqueness](image)
Example

```c
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;
}
```

<table>
<thead>
<tr>
<th>Developer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>filename</td>
</tr>
<tr>
<td>V2</td>
<td>mode</td>
</tr>
<tr>
<td>V3</td>
<td>is_private</td>
</tr>
</tbody>
</table>
Example

```c
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;
}
```

<table>
<thead>
<tr>
<th>Developer</th>
<th>Recovered</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>filename</td>
</tr>
<tr>
<td>V2</td>
<td>mode</td>
</tr>
<tr>
<td>V3</td>
<td>is_private</td>
</tr>
</tbody>
</table>
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
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.
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?)

C++ Abstractions

Square
virtual draw()
virtual getArea()
getLength()

int side_len
SquareHelper sqh

Shape
virtual draw()
virtual getArea()
print()

int cached_area

SquareHelper
areaLogic()

int calculated

Composition

Inherits From
OOAnalyzer Design Overview

### C++ Component
- **Pharos Fact Exporter**

### Prolog Reasoning Component
- **Forward Reasoning**
- **Hypothetical Reasoning**
- **Consistency Checking**

#### OOAnalyzer Framework
- **OOAnalyzer Tool**
- **Recovered Object Oriented Abstractions**
- **Decompiled C++ Source Code Displayed in Ghidra**

**Input C++ Executable**

- **Prolog Reasoning Component**
- **Hypothetical Reasoning**
- **Consistency Checking**

**Recover Object Oriented Abstractions**

- **Shape**
  - virtual draw()
  - virtual getArea()
  - int cached_area
- **SquareHelper**
  - areaLogic()
  - int calculated
- **Composition**
- **Inherits From**

**C++ Abstractions**

- **Square**
  - virtual draw()
  - virtual getArea()
  - int side_len
  - SquareHelper sm

**Pharos Framework**

- **Pharos Fact Exporter**
- **OOAnalyzer Tool**
- **Decompiled C++ Source**
- **Executable**

**Forward Reasoning**

**Hypothetical Reasoning**

**Consistency Checking**

**C++ Component**

- **Pharos Fact Exporter**

**Prolog Reasoning Component**

- **Forward Reasoning**
- **Hypothetical Reasoning**
- **Consistency Checking**
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 initial 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.

<table>
<thead>
<tr>
<th>Fact Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjPtrAllocation(I, F, P, S)</td>
<td>Instruction I in function F allocates S bytes of memory for the object pointed to by P.</td>
</tr>
<tr>
<td>ObjPtrInvoke(I, F, P, M)</td>
<td>Instruction I in function F calls method M on the object pointed to by P.</td>
</tr>
<tr>
<td>ObjPtrOffset(P₁, O, P₂)</td>
<td>Object pointer P₂ points to P₁ + O.</td>
</tr>
<tr>
<td>MemberAccess(I, M, O, S)</td>
<td>Instruction I in method M accesses S bytes of memory at offset O from the current object's pointer.</td>
</tr>
<tr>
<td>ThisCallMethod(M, P)</td>
<td>Method M receives the object pointed to by P in the ecx register.</td>
</tr>
<tr>
<td>NoCallsBefore(M)</td>
<td>No methods are called on any object pointer before method M.</td>
</tr>
<tr>
<td>ReturnsSelf(M)</td>
<td>Method M returns the object pointer that was passed as a parameter.</td>
</tr>
<tr>
<td>UninitializedReads(M)</td>
<td>Method M reads memory that was not written to by M.</td>
</tr>
<tr>
<td>PossibleVFTableEntry(VFT, O, M)</td>
<td>Method M may be at offset O in vtable VFT.</td>
</tr>
</tbody>
</table>
## Entity Facts

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

<table>
<thead>
<tr>
<th>Fact Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method(M)</td>
<td>Method M is an OO method on a class or struct.</td>
</tr>
<tr>
<td>Constructor(M)</td>
<td>Method M is an object constructor.</td>
</tr>
<tr>
<td>Destructor(M)</td>
<td>Method M is an object destructor.</td>
</tr>
<tr>
<td>( Cl_a = Cl_b )</td>
<td>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.</td>
</tr>
<tr>
<td>( Cl_a \leq Cl_b )</td>
<td>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 ).</td>
</tr>
<tr>
<td>( M \in Cl )</td>
<td>Method M is defined directly on class Cl.</td>
</tr>
<tr>
<td>ClassCallsMethod(Cl, M)</td>
<td>An instance of class Cl calls method M.</td>
</tr>
</tbody>
</table>

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

Forward reasoning
- Unambiguous scenarios
- Interpretation: If $P_1, P_2, \ldots, P_n$ are satisfied, then $C$ is true
- If inconsistency is reached, $P_1, P_2, \ldots, P_n$ must not be true

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

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

\[
\text{Constructor}(M_d) \quad M_d \in C_l_d \\
\text{Constructor}(M_b) \quad M_b \in C_l_b \\
\text{ClassCallsMethod}(C_l_d, M) \quad M_d \neq M_b \\
\text{ClassCallsMethod}(C_l_b, M) \quad M \in C_l_m \quad C_l_d \neq C_l_b \quad \text{DerivedClass}(C_l_d, C_l_b, _) \\
\hline
C_l_m \neq C_l_d
\]
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.

\[
\text{ClassCallsMethod}(Cl_d, M) \quad \neg \text{ClassCallsMethod}(Cl_b, M)
\]

\[
M \in Cl \quad \text{DerivedClass}(Cl_d, Cl_b, \_)
\]

\[
Cl_d = Cl
\]
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

- Targets all classes and all methods
  - Most 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
OOAnalyzer Scales Well…
OOAnalyzer Scales Well… Until It Doesn't

**Graph:**
- **X-axis:** Number of Methods
- **Y-axis:** Hours
- **Legend:** mysql.exe
- **Data Points:**
  - (0, 0)
  - (500, 1)
  - (1000, 2)
  - (1500, 3)
  - (2000, 4)
  - (2500, 5)
  - (3000, 5)

**Observation:**
- The graph shows a trend where the number of methods increases significantly with an increase in hours, indicating that OOAnalyzer scales well until the method count reaches a certain threshold.
OOAnalyzer Scales Well… Until It Doesn't

DoD needs solutions here.

mysql.exe

mysql.exe
OEAnalyzer has not made any progress in 10 days.
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 \( P \Rightarrow Q \) 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

- Pharos facts
- Pharos #guesses
- Pharos #reasoningForward steps
- Global stack
- Trail stack
- Local stack
- Total stack usage
- Thread CPU
- Process CPU
- Atoms
- Functors
- Modules
- Predicates
-.causes
- Memory for VM code
- Tables
- Table space
- Heap memory
- Malloc lost
- RSS memory

Sample rate: 1 min
Advancing Cyber Operator Tradecraft through Automated Static Binary Analysis
© 2020 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.
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

![Graph showing time (old) and time (new) with number of methods and hours on the x and y axes, respectively.](image-url)
Before And After

DoD needs solutions here.

mysqld.exe

Number of Methods

Days

mysql.exe
Before And After

![Graph showing the comparison between mysql.exe and mysql.exe after applying automated static binary analysis. The graph illustrates the decrease in days required with an increase in the number of methods analyzed.]
Before and After on mysqld.exe

XSB has not made any progress in 10 days.
Before and After on mysqld.exe

SWI finishes after 33 hours.

XSB has not made any progress in 10 days.
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
RESEARCH REVIEW 2020

Team Members

Cory Cohen

Dr. Edward Schwartz
END OF PRESENTATION
Null Function Abstraction: Simplify!

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 = Poor

Speed = Good

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Accuracy" /></td>
<td><img src="image2" alt="Speed" /></td>
</tr>
</tbody>
</table>

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
  - Can be used on unknown software (malware)

- Targets all classes and 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)
## ObjDigger vs. OOAnalyzer Edit Distances on Cleanware Programs

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

OOAnalyzer recovers 67% to 84% of methods on cleanware programs.
## RESEARCH REVIEW 2020

ObjDigger vs. OOAnalyzer Edit Distances on Malware

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

OOAnalyzer recovers 68% to 91% of methods on smaller malware samples.
## OOAnalyzer Method Classification on Cleanware

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

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
WeOriginallyLookedataaFewMediumSizedPrograms
...andaLotofSmallPrograms
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 (-00)
• One architecture (x86-64)
• We are training a model that operates in more realistic environments
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