The Pharos Static Analysis Framework for Software Assurance

Cory Cohen, Senior Malware Researcher

CERT Coordination Center

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Distribution Statements

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.

DM18-1339

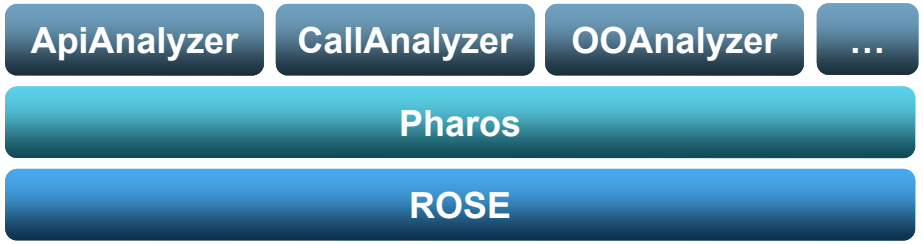
The Pharos Static Analysis Framework

The Vision for the Pharos Infrastructure

Research platform for program analysis
- Explore techniques in literature
- Develop innovative new algorithms
- Build tools to solve real problems

Targets malware specific challenges
- Handle obfuscated control flow
- Support malformed PE Headers
- Defensive use of provided data
- Simple packaging and distribution

Provides stability and consistency to our tool development and transition efforts.



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
- http://rosecompiler.org/

Pharos Framework is publicly available on GitHub:

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

Malware Analysis Tools Built in the Pharos Framework

FN2Yara

Produces hashes and statistics for functions in a binary executable

Detects identical and similar code in executable programs, allowing analysts to leverage previous analysis effort

FN2Hash

Automatically generates YARA signatures and function hashes

Promotes high quality signatures for network defense, identifies similar files and facilitates clustering

API Analyzer

Detects patterns of API calls representing malicious behaviors

Focuses analyst attention on important malware behavior, detects unexpected patterns for software assurance

Call Analyzer

Reports constant parameters to calls in binary executables

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

OO Analyzer

Detects object oriented constructs, resolves virtual function calls

Greatly reduces malware analysis effort required for deep understanding of object oriented malware capabilities

Malware Design Matcher

Detects high level design abstractions in malware files

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

Code Clone Detection with FN2Yara and FN2Hash

FN2Hash Generates Hashes and More for Functions

FN2Hash analyses the program structurally and produces a variety of data:

- MD5 hash of the file and the address of function
- Number of basic blocks (known to be in control flow and not)
- Number of instructions and the number of bytes in the function
- The exact hash, the PIC hash, and the composite PIC hash of the function
- Several mnemonic hashes that use disassembled instruction names
- Output in CSV format and more recently in JSON

The “fse.py” script allows analysts to examine the intersection of sets of functions

- Each function in the set has a line in the output with address and other data
- The files are represented as columns with X marks (sorted intelligently)
- Using this tool analysts can easily determine which functions are in which files

Example Output from fse.py Script

```
$ ./tools/fn2hash/fse.py /tmp/ooex?.csv
...
F|F|F| |   |   |           |                                 |
0|1|2|c|  i|  b|       addr|                              PIC|
-|-|-|-|---|---|-----------|---------------------------------|
X|X|X| |161|519| ??????????|496569DBC1D3334166AA63CC8F54BD09|
X|X|X| | 79|250| ??????????|7118DA88B6F46D97F2325E375B46BCDA|
X|X|X| | 12| 26| ??????????|61A3B4B96481F0AEB7EA7536B4C05936|
X|X|X| | 33|103| ??????????|6E39921F1917D90ED9D05D8D2412DCE3|
X|X| | | 47|146| ??????????|70C9734F423EF90CBF093CD6F1F0571C|
X|X| | | 18| 46| ??????????|CD7682D556A404EADF1C69A3435C8022|
X| | | | 22| 51| 0x00411900|885868DC9776FE65066D939ED971E475|
 |X| | | 27| 75| 0x004116F0|4869D78202590835039046C122E9A455|
 | |X| | 26| 71| 0x00411670|84ED7EED1325F349A9B1A25F88AD22F6|
```

FN2Yara Generates Yara Signatures for Functions & Files

Hashes are useful, but you can't search for them.

YARA is a file searching and matching tool widely used in malware analysis.

It uses our position independent code (PIC) hashing technique.

- Removes bytes from instructions that vary based on code location.
- Allows analysts to identify fixed sequences of bytes to search for.

Generates standard Yara signatures to match functions and files.

FN2Yara Automated Signature Generation

Analyst uses the function address on each rule to understand what's being matched.

```
rule Function_1004C610
{
  strings:
    // File malware.ex_ @ 0x1004C610 (2015-11-09)
    $Match_1004c610 = { 55 8b ec 83 ec 04 ?? ?? ?? ??
7d 08 8b 4d 0c c1 e9 07 66 0f ef c0 eb ?? ?? ?? ??}
```

Bytes of function chunks have relocatable addresses wild carded.

Each non-contiguous chunk has a separate match string.

```
    $Match_1004c730 = { 66 0f 7f 07 66 0f 7f 47 10 66 0f
7f 47 20 66 ?? ?? ?? ?? 66 0f 7f 47 40 66 0f 7f 47 50 66
0f 7f 47 60 66 0f 7f 47 70 8d bf ?? ?? ?? ?? 49 75 d0 8b
7d fc 8b e5 5d c3 }
```

YARA can match a percentage of the function chunks.

```
  condition:
    all of them
}
```

Matching the bytes doesn't require disassembly at matching time, but the signatures have semantic meaning if the analyst knows the significance of the function.

FN2Yara Modes of Operation

Per-Function

Fn2yara command:

```
$ ./fn2yara -o malware.yara malware.exe
OPTI[INFO ]: Analyzing executable: malware.exe
...
OPTI[INFO ]: Wrote 1750 rules to malware.yara
OPTI[INFO ]: Complete.
```

Fn2yara Output:

```
rule Function_md5_XXX_1004C610
{
  strings:
    // File malware.ex_ @ 0x1004C610 (2015-11-09)
    $Match_1004c610 = { 55 8b ec 83 ec 04 89 7d fc 8b 7d 08 8b
4d 0c c1 e9 07 66 0f ef c0 eb ?? }
    $Match_1004c630 = { 66 0f 7f 07 66 0f 7f 47 10 66 0f 7f 47 20
66 0f 7f q47 70 8d bf 80 00 00 00 49 75 d0 8b 7d fc 8b e5 5d c3 }
  condition:
    all of them
}
....
```

One rule for each function. All chunks must match in each function

Whole File (--comparison)

Fn2yara command:

```
$ ./fn2yara -o malware_p50.yara --comparison -T 50 malware.exe
OPTI[INFO ]: Analyzing executable: malware.exe
...
OPTI[INFO ]: Wrote 1978 strings to malware_p50.yara
OPTI[INFO ]: Complete.
```

Fn2yara Output:

```
rule md5_0D57D2BEF1296BE62A3E791BFAD33BCD_50_percent
{
  strings:
    // string $Match_00401000 contains 33 bytes and 10 instructions
    $Match_00401000 = { 80 f9 40 73 16 80 f9 20 73 06 0f ad d0 d3 fa
c3 8b c2 c1 fa 1f 80 e1 1f d3 f8 c3 c1 fa 1f 8b c2 c3 }
    $Match_00401030 = { 8b 44 24 08 8b 4c 24 10 0b c8 8b 4c 24 0c 75
09 8b 44 24 04 f7 e1 c2 10 00 53 f7 e1 8b d8 8b 44 24 08 f7 64 24 14
03 d8 8b 44 24 08 f7 e1 03 d3 5b c2 10 00 }
     ...
  condition:
    989 of them
}
```

One rule for the entire file. Condition is 50% of the 1,978 strings must match.

FN2Hash & FN2Yara Applications to Software Assurance

FN2Hash can be used to:

- Detect differences between samples and “known good” similar samples
- Identify library code reused in applications (by hashing functions in the library to find)

FN2Yara can be used to:

- Scan large software collections for specific functions or code patterns
- As a poor man’s custom anti-virus solution

Software Behavior Detection with API Analyzer

ApiAnalyzer Detects API Behavior Patterns

The overall design of ApiAnalyzer is:

- Search the control flow graph for call to APIs
- Analyze parameters to conduct data flow analysis between calls
- Express patterns to search for as JSON data structures
- Find and report matched patterns in programs

ApiAnalyzer example output:

Found: ReverseShellv1 starting at address 0x00401052 Path:
0x00401052(KERNEL32.DLL!CREATEPIPE) ->
0x00401073(KERNEL32.DLL!CREATEPIPE) ->
0x0040108E(KERNEL32.DLL!GETSTARTUPINFOA) ->
0x004010EC(KERNEL32.DLL!CREATEPROCESSA)

API Analyzer – Statically searches for API sequences

Name and description makes signature accessible to analyst

List of API calls defines what the sequence is that we're looking for

```
"Sig": {
 "Name": "WriteProgramResourceToFile",
 "Description": "Load a program resource and write it to disk",
 "Category": "MALWARE",
 "Pattern": [
  { "API": "Kernel32.dll!FindResourceA",
    "Retn": {"Name": "HRSRC"} },
  { "API": "Kernel32.dll!LoadResource",
    "Args": [{"Name": "HRSRC", "Index": 1, "Type": "IN"}],
    "Retn": {"Name": "HANDLE"} },
  { "API": "Kernel32.dll!LockResource",
    "Args": [{"Name": "HANDLE", "Index": 0, "Type": "IN"}],
    "Retn": {"Name": "RES"} },
  { "API": "Kernel32.dll!WriteFile",
    "Args": [{"Name": "RES", "Index": 1, "Type": "IN"}] }
 ]}
```

Reused variable names describe how calls are connected via data flow

API parameters are now stored in an SQLite database, but APIAnalyzer is not using it yet

ApiAnalyzer Applications to Software Assurance

ApiAnalyzer can be used to:

- Look for API calls that should not be present
- Detect recognized patterns of malicious or suspicious behavior
- Identify locations in the software where specific functionality exists for further review

Static Recovery of API Parameters with CallAnalyzer

Call Analyzer: A Tool for Binary Analysis of Function Calls

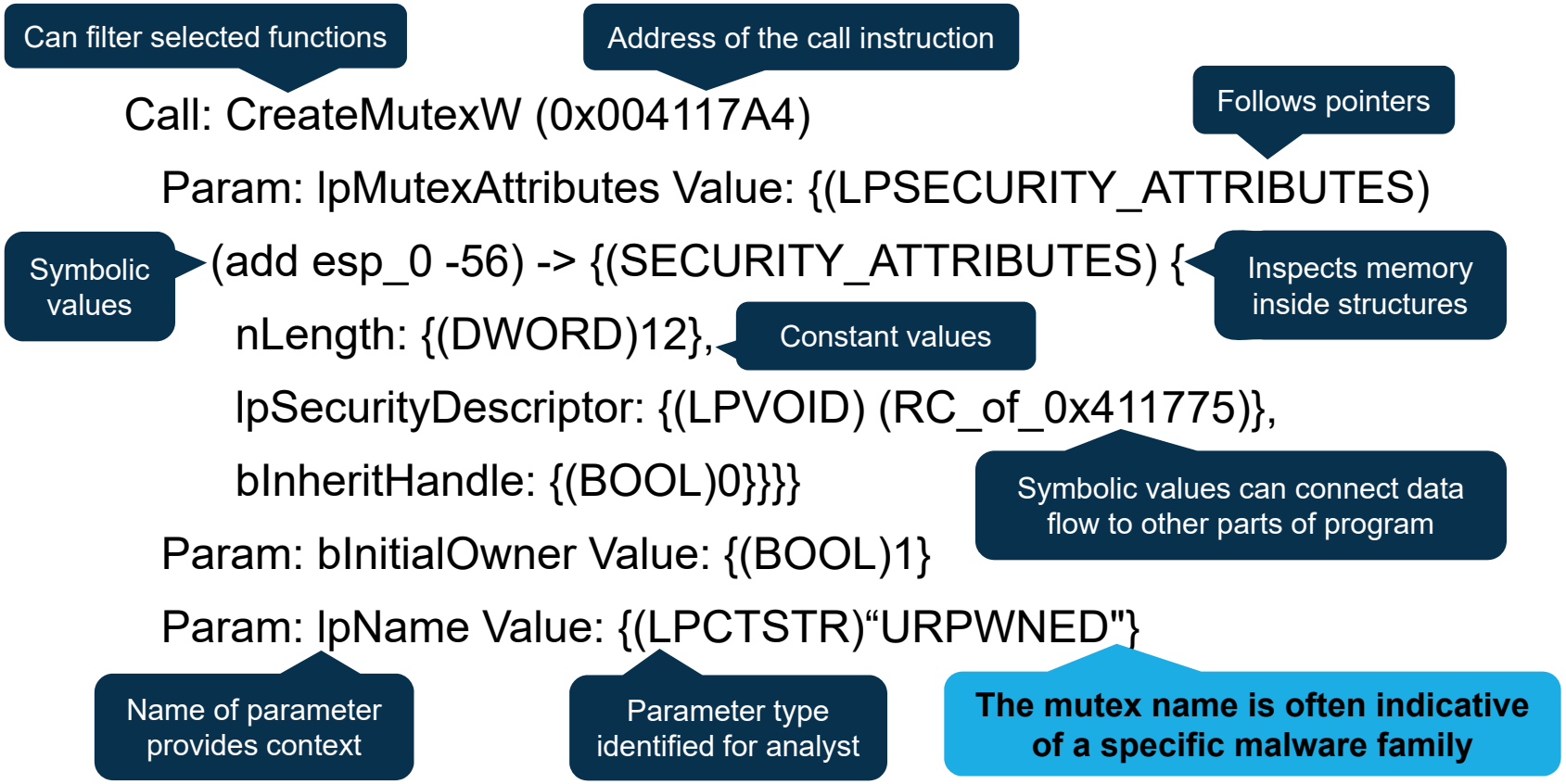
The overall design of CallAnalyzer is:

- Visit each call instruction in the program
- Perform call parameter analysis
- Report known static parameters (e.g. file names, registry keys, etc.)
- Can also report symbolic values

Relies heavily on our API database, which provides:

- API calling convention
- Parameter counts and stack deltas
- Parameter names
- Parameter types
- In/out properties

Call Analyzer: A Tool for Binary Analysis of Function Calls



CallAnalyzer Applications to Software Assurance

CallAnalyzer can be used to:

- Identify files, registry keys and mutexes used by the software
- Identify API calls with specific parameter values (desired or undesired)

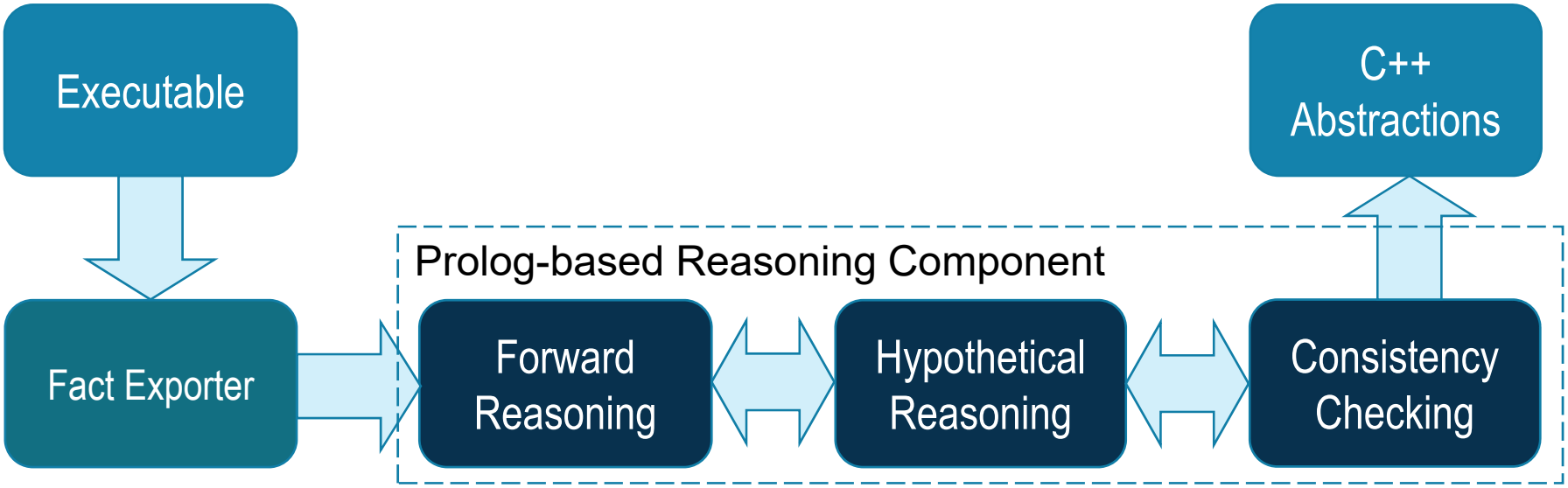
Object Oriented Class Recovery with OOAnalyzer

Object Oriented Class Recovery with OOAnalyzer

Export low level facts using light weight static analysis in ROSE/Pharos.

Prolog-based reasoning component deduces facts and makes hypothetical assertions.

Resulting model is validated for consistency and presented when no guesses remain.



OOAnalyzer Lightweight Fact Exporter

```
MyObject* obj = new MyObject();
```

```
push    1730h
call    ??2@YAPAXI@Z
add     esp, 4
mov     [esp+8], eax
test    eax, eax
mov     [esp+4], 0
jz      failed
mov     ecx, eax
call    ctor
```

Object size

New() method

Thiscall

Constructor

Converted to Prolog facts

Prolog facts don't require complex inter-procedural analysis to generate!

```
returnsSelf(0x411350).
thisPtrAlloc(0x411352, 0x1730).
callReturn(0x411352, SV45).
callParam(0x411380, 1, SV45).
noCallsBefore(0x411380).
callingConv(0x411450, 'thiscall').
callTarget(0x411380, 0x411450).
...
```

(example facts have been fudged a little for the slide)

OOAnalyzer Prolog Reasoning Rules

Which Prolog?

- Evaluated multiple Prolog implementations
- Selected XSB Prolog for tabling features
- Lots of bugs and crashes before success

Why Prolog?

- Represents human knowledge clearly
- Reasons forward conclusively given facts
- Allows rules to fire in arbitrary order

Why Guess?

- Too few facts for forward reasoning alone
- Guess facts using weaker rules as needed
- Backtrack using custom constraint system

Giving better results than the old approach!

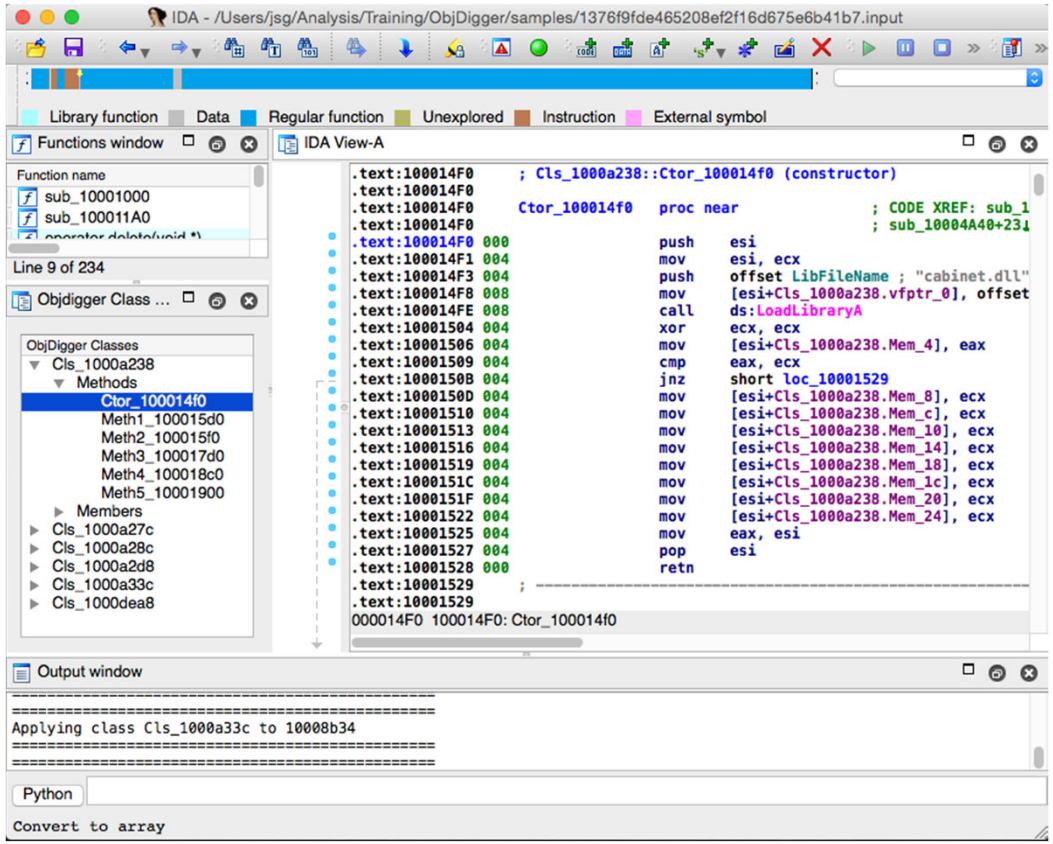
```
Constructor(Method) :-
  VFTableWrite(Method, _VFT, _),
  NOTDestructor(Method).

DerivedClass(Ctor1, Ctor2, Offset) :-
  Constructor(Ctor1),
  Constructor(Ctor2),
  Calls(Ctor1, Ctor2),
  VFTableWrite(Ctor1, VFT1, Offset),
  VFTableWrite(Ctor2, VFT2, Offset),
  Ctor1 \= Ctor2, VFT1 \= VFT2.
```

(example rules are a little fuzzy too)

OOAnalyzer Results

Software Analyzed			Method Edit Distance (without RTTI)							Edit Distance (with RTTI)		Edit Distance (no guessing)	
Name	Classes	Methods	Move	Add	Rem	Split	Join	Total	%	Total	%	Total	%
CImg	29	220	3	15	1	1	1	21	9.5	21	9.5	200	90.9
Firefox	141	638	40	64	67	40	1	212	33.2	212	33.2	499	78.2
Light POP3	44	295	15	15	0	12	2	44	14.9	41	13.9	263	89.2
muParser	180	1439	207	111	17	98	27	460	32.0	451	31.3	1312	91.2
MYSQL	202	1395	229	69	4	74	63	439	31.5	433	31.0	1110	79.6
OptionParser	11	56	3	3	0	0	0	6	10.7	6	10.7	56	100.0
PicoHTTPD	95	656	54	58	10	24	20	166	25.3	161	24.5	569	86.7
TinyXML	35	415	30	23	1	5	10	69	16.6	68	16.4	384	92.5
x3c	6	28	1	4	0	0	0	5	17.9	5	17.9	28	100.0
Malware 1	207	1920	121	179	24	25	24	373	19.4	369	19.2	1724	89.8
Malware 2	55	339	5	10	5	6	3	29	8.6	29	8.6	174	51.3
Average	120	848							21.5		21.2		82.2

OOAnalyzer IDA Plugin – User Interface for Malware Analysts



IDA Python Plugin:

- Displays classes
- Helps rename classes
- Group methods & members
- Creates IDA structures
- Updates assembly operands

Not the most important part of our research, but an important part of our transition and adoption efforts.

OOAnalyzer Applications to Software Assurance

OOAnalyzer can be used to:

- Understand the design of object oriented software
- Gain automated insight into OO library usage?
- Compare design documents to observed OO constructs?

Malware Design Matcher (Research Prototype)

Malware Design Matcher

Our experiences with OOAnalyzer led us to new questions:

- Can we compare object-oriented program designs?
- Can we detect specific designs? (e.g. a specific malware family)
- Can we bring multiple tools together to make this easier?

The malware design matcher research prototype does this

- Express patterns to look for as Prolog rules
- Export facts from OOAnalyzer & APIAnalyzer analyses
- Match patterns against facts using Prolog

Malware Design Matcher Example

Enumerate the features that define the pattern:

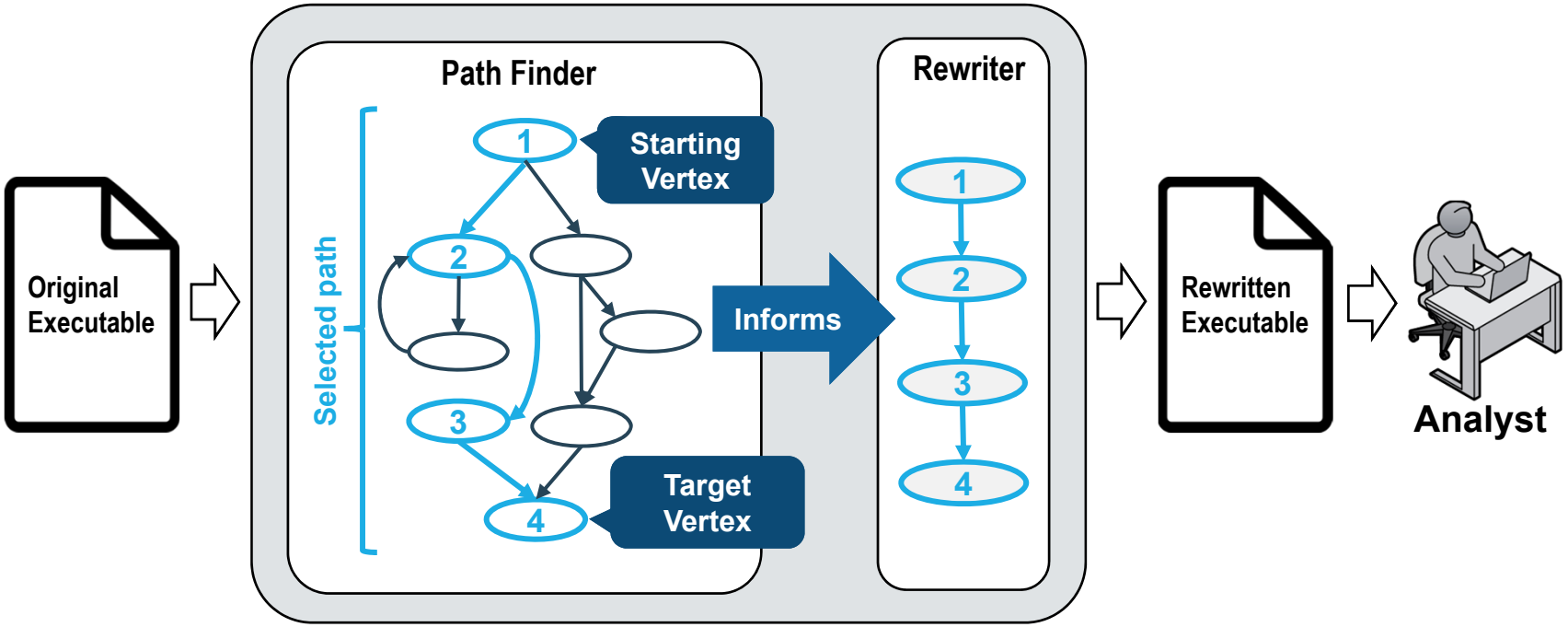
1. There exist four unnamed classes (we'll call them C, CC, I, & R).
2. CC inherits from C (begin by temporarily labeling C & CC)
3. The constructor for CC (#2) takes an R as a parameter.
4. There's a method E on CC (#2) that calls a method in R (#3).
5. The method E (#4) is virtual.
6. Class C (#2) contains an instance of R (#3) as a member.
7. Class I that has a method X that takes C or CC (#2) as a parameter.
8. The method X (#7) calls method E (#5).

Test for each feature. Pattern is present if all features are present.

Identified components can be labelled automatically after detection.

Path Finding and Executable Rewriting (Current Research)

Path Finding & Executable Rewriting



Publicly Available Pharos Resources & Presentations

Pharos GitHub Repository

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

SEI Cyber Minute Podcast on Pharos Framework

- https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=509431

2017 SEI Blog Post Announcing GitHub Software Release

- https://insights.sei.cmu.edu/sei_blog/2017/08/pharos-binary-static-analysis-tools-released-on-github.html

Design Pattern Recovery from Malware - 2015 SEI Research Review

- https://resources.sei.cmu.edu/asset_files/Presentation/2015_017_001_446350.pdf

Practical, Large-Scale Detection of Obfuscated Malware Code Via Flow Dependency Indexing

- http://repository.cmu.edu/dissertations/389/ (2014 CMU Dissertation by Wesley Jin)

Dagstuhl Seminar 17281 – Malware Analysis: From Large-Scale Data Triage to Targeted Attack Recognition

- https://materials.dagstuhl.de/files/17/17281/17281.CoryCohen.Slides.pdf

Recovering C++ Objects From Binaries Using Inter-Procedural Data-Flow Analysis (PPREW paper)

- https://dl.acm.org/citation.cfm?id=2556465

Questions?

Cory Cohen, Senior Malware Researcher

CERT Coordination Center

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Pharos & ROSE Static Analysis Capabilities

Cory Cohen, Senior Malware Researcher

CERT Coordination Center

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Distribution Statements

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.

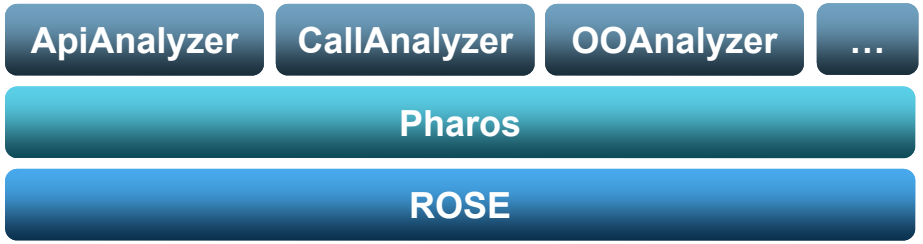
DM18-0786

The Vision for the Pharos Infrastructure

Research platform for program analysis
- Explore techniques in literature
- Develop innovative new algorithms
- Build tools to solve real problems

Targets malware specific challenges
- Handle obfuscated control flow
- Support malformed PE Headers
- Defensive use of provided data
- Simple packaging and distribution

Provides stability and consistency to our tool development and transition efforts.



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
- http://rosecompiler.org/

Pharos Framework is publicly available on GitHub:

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

ROSE & Pharos Capabilities

Today's presentation is about the Pharos and ROSE frameworks

In contrast to yesterday's talk which was about the tools we've built

The boundaries between ROSE and Pharos are subtle but I've tried to be clear

- Many of the features in ROSE originated in experiments we conducted
- Our Pharos add-ons tend to drift back into the ROSE code base
- We have a close partnership the ROSE developers that makes this possible

Executable Container File Format Parsing (ROSE)

Windows Portable Executable (PE)

- Based on extensively tested CERT file parsing library
- Handles malware malformations, e.g. overlapping sections

Linux Executable and Linkable (ELF)

- Not extensively tested by us, appears to work fine
- Custom implementation (not elfutils)

DOS, LE, NE, raw memory maps, Motorola SREC

Currently no parser currently for Mach-O file format

Data stored in Abstract Syntax Tree (AST) representation

Import symbols, entry point, virtual memory mapping, etc.

Provides some standardization across diverse formats

File Format Parsing

Instruction Disassembly (ROSE)

Disassembly turns bytes at an address into instances of an instruction object in the AST

Intel x86 and AMD x64

- x86 used and tested fairly extensively
- Instruction coverage complete (?)

Power PC & MIPS (not tested by us)

Motorola 68000 series (through Coldfire)

ARM - code exists but it has known problems

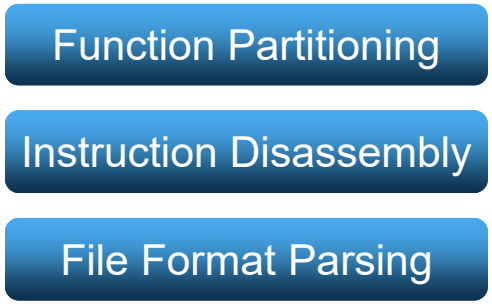
Instruction Disassembly

File Format Parsing

Consistent API across multiple architectures

- e.g. operand object, shared register API

Function Partitioning (ROSE & Pharos)



Function partitioning groups instructions into functions

- This is a difficult open research problem (ground truth?)

ROSE has a custom-coded control flow following partitioner:

- Evaluates previous basic block to identify opaque predicates
- Assembles functions even from non-contiguous basic blocks
- Overlapping instructions are handled correctly
- Uses patterns for finding code not in control flow

Pharos adds features to the stock ROSE partitioner:

- Better detection of code not obviously in the control flow
- Detection of “bad code” (e.g. jumping into unpacked code)

Each architecture has different function boundary properties

Partitioners for x86, m68k but not MIPS, PowerPC or ARM

Instruction Semantics & Emulation Framework (ROSE)

Instruction semantics describe what each instruction does

Expressed in terms of 41 “RISC Operators” defined in ROSE

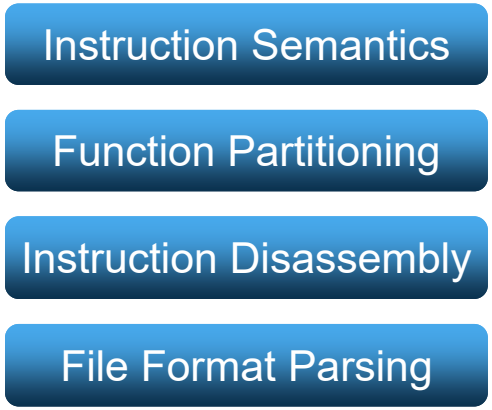
User calls processInstruction() on an instruction in the AST

An instructionDispatcher object invokes code for architecture

Architecture specific code is a big switch block on instruction

Instructions are implemented as calls to RISC operators

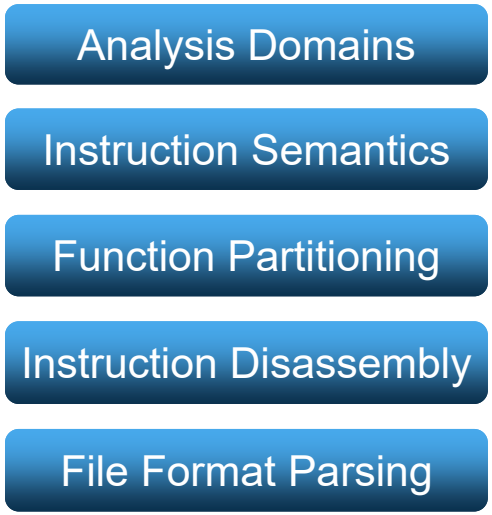
RISC operators are implemented as callbacks into a domain



Instruction semantics are available for:

- x86, x64, m68k, PowerPC and LLVM

Analysis Domains (ROSE)



What a RISC Operator does depends on your analysis domain.

You can define an analysis domain however you like…

ROSE defines several analysis domains (value representation)

- Concrete (bit vectors, aka constants)
- Symbolic (algebraic expressions, variables and constants)
- Partial (algebraic but only one variable and constant)
- Multi (combines multiple other domains)
- Null (does nothing, for testing and tracing)

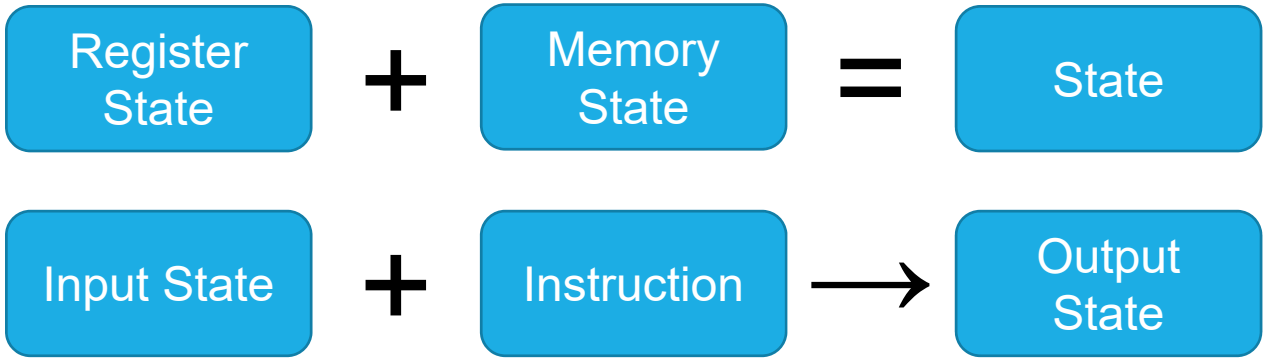
Could be used for specific analyses (e.g. value sets…)

Pharos adds its own domain that we'll discuss later

Register and Memory State Modeling (ROSE)

Domains must implement RISC Operators… but how?

Transform an input state into an output state



- Analysis Domains
- Instruction Semantics
- Function Partitioning
- Instruction Disassembly
- File Format Parsing

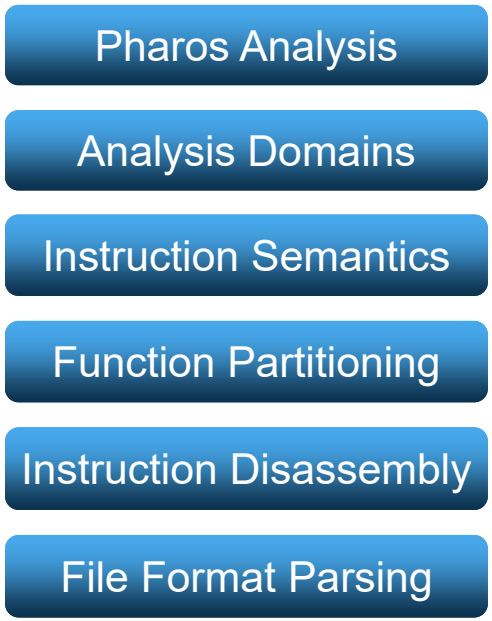
ROSE RegisterState is architecture independent

ROSE MemoryState provides several memory models:

- Ordered list of reads and writes (handles aliasing, slower)
- Map based key value pairs (less accurate, faster)

Pharos Analysis Domain

We've defined our own custom analysis domain



Features include:

- Based on ROSE symbolic domain with several changes
- Transforms a **function** input state into an output state
- Processes functions in "bottom up call order"
- Propagates stack deltas for Win32 stdcall calling convention
- Handles branches with ITE (if-then-else) RISC Operator
- Loops through control flow graph seeking fixed point
- Unroll loops up to five times (inaccurate but easy)
- Computes instruction usage and definition chains
- Analyses register usage, call parameters and return values
- Analyses types of local and global variables

Pharos Analysis Algorithms – Type Analysis (for example)

- Algorithms
- Pharos Analysis
- Analysis Domains
- Instruction Semantics
- Function Partitioning
- Instruction Disassembly
- File Format Parsing

Types in source code are fairly clear:

```
int x = 0; int y = *p;
char *z = func(x, y)
```

Types are less clear in assembly:

```
mov eax, [ebx]
shl eax, 2
push eax
mov [ebp-C], 0
push [ebp-C]
call func
```

EBX is a pointer

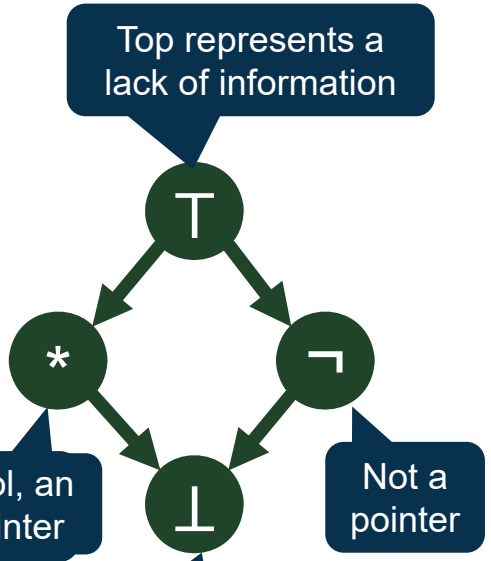
EAX is not a pointer

and solve

for solution

Zero could be a bool, an integer or a null pointer

API database can assign types through data flow



Pharos Analysis Tools

The Pharos tools are built on top of this infrastructure:

- FN2Yara
- FN2Hash
- ApiAnalyzer
- CallAnalyzer
- OOAnalyzer

Framework includes features to simplify this:

- Shared command line option parsing
- Logging, configuration, JSON, Prolog interaction

Pharos Static Analysis Framework Future Directions

We're actively developing and maintain the Pharos framework

We're currently working on path finding and binary rewriting.

We've recently added a more traditional intermediate representation approach.

That source (along with our path finding code) should be available soon.

Also planning to release another update to the API database.

It's freely available in source code format from GitHub:

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

Also check out our stand alone Visual Studio C++ name demangler library:

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

Questions?

Cory Cohen, Senior Malware Researcher

CERT Coordination Center

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213