

Design Pattern Recovery from Malware Binaries

Cory F. Cohen

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213



Software Engineering Institute

Carnegie Mellon University

© 2015 Carnegie Mellon University

Distribution Statement A: Approved for
Public Release; Distribution is Unlimited

Copyright 2015 Carnegie Mellon University

This material is based upon work funded and supported by Department of Defense and Department of Homeland Security under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center sponsored by the United States Department of Defense.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Department of Defense and Department of Homeland Security or the United States Department of Defense.

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.

This material has been approved for public release and unlimited distribution except as restricted below.

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.

Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM-0002840

Automated Binary Analysis Challenges

Software Assurance

- We need to answer basic questions about functionality
- Does it contain known bad or suspicious code?
- Does this binary program do what we think it does?

Malware Analysis

- Time consuming and complex manual process
- Requires highly specialized reverse engineering skills
- We need to fully automate malware analysis tasks
- Custom tools must be built on a solid foundation



Binary Static Analysis Infrastructure

Components needed for binary analysis framework

- File format parsing
- Disassembler
- Function partitioner
- Instruction semantics
- Emulation framework
- Use-def chains
- SMT solver integration
- Algebraic simplification

We built on the ROSE platform:

- Binary analysis capabilities
- Working closely with LLNL
- BSD Licensed
- C++ Library Implementation
- Highly extensible

We extended ROSE with:

- Calling convention detection
- Stack delta analysis
- Parameter tracking
- Type recovery (in progress)



Objdigger: Object Oriented Analysis

```

00401010 _main      proc near
00401010
00401010 var_B4      = byte ptr -0B4h
00401010 var_C       = dword ptr -0Ch
00401010 var_4       = dword ptr -4
00401010 argc      = dword ptr 4
00401010 argv      = dword ptr 8
00401010 envp      = dword ptr 0Ch
00401010
00401010
00401010
00401012
00401017
0040101D
00401020
0040102C
00401030
00401035
0040103C
00401043
00401044
00401045
00401049
00401054
00401059
0040105D
0040105F
0040106A
0040106F
00401076
00401078
00401079
00401080
00401086
00401086 _main      endp

```

Constructor

Method

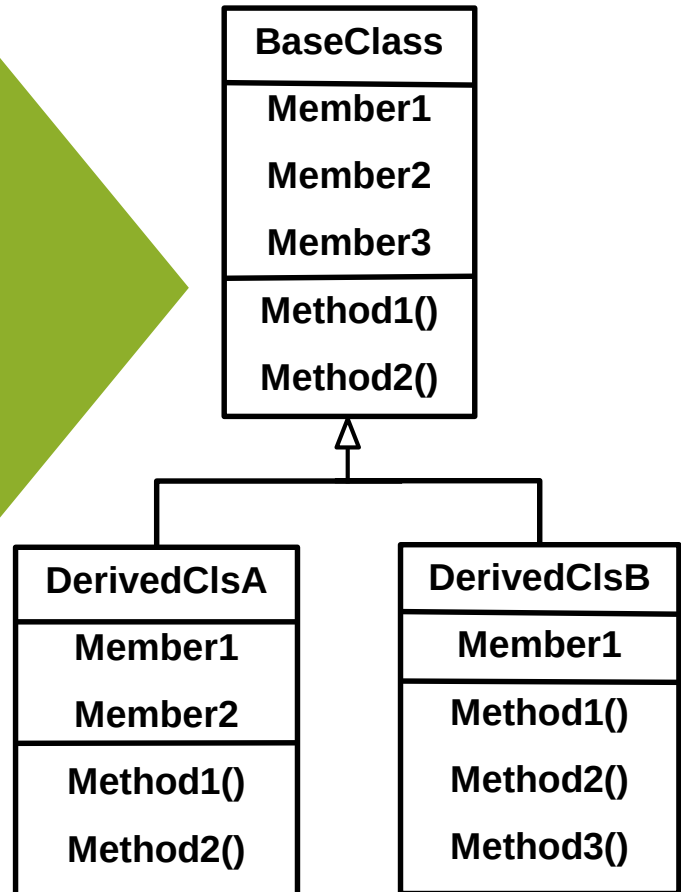
Method

```

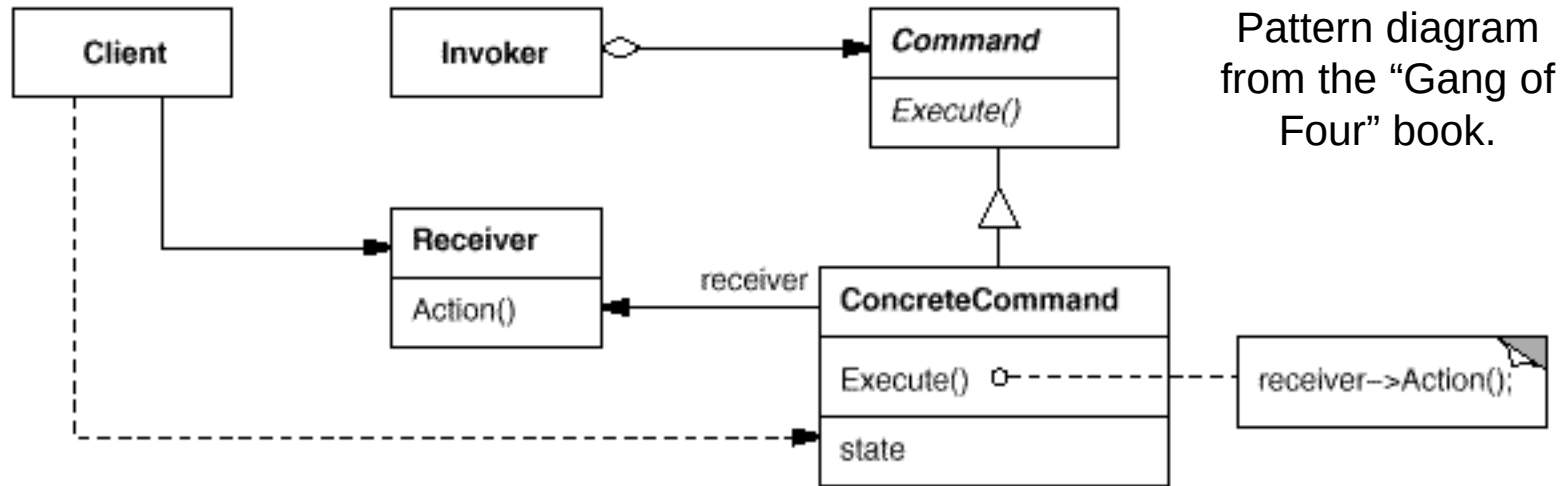
proc near
= byte ptr -0B4h
= dword ptr -0Ch
= dword ptr -4
= dword ptr 4
= dword ptr 8
= dword ptr 0Ch

```

Stack
Allocated



Design Pattern Recovery Problem



Malware authors face similar software design challenges

- Develop reusable components to ease software evolution
- Combine components in new ways to accomplish goals
- Code reuse is challenged by anti-virus detection efforts

Analysts want to match these patterns in executables

- Recognize higher abstractions in low-level assembly
- Anecdotal evidence supports "malware specific" patterns

A Command Pattern Source Implementation

```
class Receiver {
public:
    void RunCP(PTSTR proc);
    void RunDF(PTSTR filename); };
```

```
class Cmd {
public: virtual void Exec() = 0;
protected: Receiver rcvr; };
```

```
class Invoker {
public: void runCmd(Cmd& c) {
    c.Exec(); } };
```

```
class CPCmd : public Cmd {
private: PTSTR proc;
public:
    CPCmd(Receiver &r, PTSTR p) {
        rcvr = r; proc = p; }
    virtual void Exec() {
        rcvr.RunCP(proc); }
};
```

```
class DFCmd : public Cmd {
private: PTSTR file;
public:
    DFCmd(Receiver &r, PTSTR f) {
        rcvr = r; file = f; }
    virtual void Exec() {
        rcvr.RunDF(file); }
};
```

```
int main() {
    Receiver r;
    CPCmd cp(r, "c:\\\\calc.exe");
    DFCmd del(r, "mal.txt");
    Invoker i;
    i.runCmd(cp);
    i.runCmd(del);
}
```



A Command Pattern Binary

```
mov    [ebp+this], ecx
mov    ecx, [ebp+this]
call   Cmd_Ctor
mov    eax, [ebp+this]
mov    [eax], offset vftable
mov    eax, [ebp+this]
mov    ecx, [ebp+c]
mov    [eax+8], ecx
mov    eax, [ebp+this]
```

```
mov    [ebp+this], ecx
mov    eax, [ebp+this]
mov    ecx, [eax+8]
push   ecx
mov    ecx, [ebp+this]
add    ecx, 4
call   Receiver_RunCP
```

Example on left is part of `CPCmd::CPCmd()` on right `CPCmd::Exec()`. Obviously, many of the source code features are lost or obscured. But many features are still there as well (as required for execution). Calling convention identified this pointer, vftable virtual functions, etc. Features can be extracted using our binary analysis framework.



Design Pattern Features & Detection

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.



Prototype Tool & Experimental Results

We implemented a design pattern matching prototype

- Framework exports facts about program as Prolog facts
- Patterns are very naturally expressed as Prolog rules
- Prolog finds the pattern and reports the matching classes

We conducted an experiment in malware family detection

- Built a gh0st/evilight malware variant from source code
- Detected a variety of classes, methods and functions
- Used class relationships, API sequences, and the call graph
- Core pattern was a socket and a command design pattern
- Primarily leveraged a reciprocal relationship between classes
- Identified command classes both generically and specifically
- Also key constructs like procedural command dispatch loop



Conclusions & Future Research

More work yet to be done on design pattern matching

- Continue to improve accuracy and completeness of features
- Conduct more experiments on pattern variation in malware
- Evaluate expressiveness of patterns given current features
- Evaluate new feature exporters to implement in framework

Successfully detected numerous abstractions in a malware sample

- Allows malware analysts to share knowledge about family
- Reduces effort by assigning semantic labels to abstractions
- Focuses analyst attention on unmatched features in new variants

Future Research in Decompilation

- Focusing on decompilation to source code in FY 2016
- Goal is to allow source analysis tools to be applied to binaries



Questions?

For more information about the Pharos suite of Automated Static Binary Analysis tools, please contact:

Cory Cohen <cfc@cert.org> 1-412-268-7925

