CS 6V81-05: System Security and Malicious Code Analysis

Binary Code Reuse

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

April 23rd, 2012
Outline

1. Background
2. Binary Code Reuse
3. Existing Techniques
4. Summary
5. Appendix: Graduate Study
Software Reuse

What is software reuse

- Reuse is a general common-sense principle, pervasively applied in all human activities
- Software reuse is an act of reuse of previously built software artifact/asset to fulfill need in new development task or sub-task

Why Reuse

- Avoiding wasting time and money redoing same task over and over
- Quality standard of previously completed task carries over to new occurrences of same need
What to reuse

Software assets

- Domain/business analysis/model
- Application requirements
- Application architectures and generic architecture patterns
- Application design/model and generic design patterns
- Source code
  - Libraries, modules, packages
  - Classes, templates, idioms
  - Objects, functions, subroutines, data
- Executable code: executable components, bytecode, binary code
- Documentation and meta-data
- Test suites and test generators
Outline

1. Background
2. Binary Code Reuse
3. Existing Techniques
4. Summary
5. Appendix: Graduate Study
The Problem: Binary Executable Transforms

Identify

Binary A
1 2 3

Binary B
1 2 3

Binary C
1 2

Extract

1

2

1

2

3

Combine

1 1 1

2 3

1 3 2

2 2

1 2 1

3 1 2

1 2

BET Basic Research Areas

Automated combinatorial approach to software development given requirements could provide novel outcomes and diverse binary sets

Credit: This slide is from Dan Roelker from a DARPA colloquium in 2011
Our Goal

Mission

Reusing binary code
Our Goal

Mission

Reusing binary code

- **Given**: x86 binary, functionality specification
- **Identify**: modular subroutines performing the functionality
- **Extract**: reusable C functions for the identified subroutine
Our Goal

Mission

Reusing binary code

- **Given**: x86 binary, functionality specification
- **Identify**: modular subroutines performing the functionality
- **Extract**: reusable C functions for the identified subroutine

About Binary Code

- Rich in every platform, with huge quantity ✓
- End users have everything ✓
- Difficulty to analyze the machine code ✗
Challenges

Component **Identification**
- How to specify functionalities to reuse?
- How to identify relevant binary artifacts with accuracy and completeness?
- How to identify re-usable subroutines from relevant artifacts?

Component **Extraction**
- Handling indirect jumps and function pointers
- Inferring variables and their types
- Inferring calling interface of extracted subroutines
Existing Techniques

Decompilation
- HexRay
- Function extraction
- BCR
- Inspector Gadget
- Virtuoso

Binary reverse engineering
- Data structures: REWARDS, TIE, Howard
- Control flow graphs: CFG generation
Existing Techniques

- Decompilation
  - HexRay
## Existing Techniques

### Decompilation
- HexRay

### Function extraction
- BCR
- Inspector Gadget
- Virtuoso
Existing Techniques

- **Decompilation**
  - HexRay

- **Function extraction**
  - BCR
  - Inspector Gadget
  - Virtuoso

- **Binary reverse engineering**
  - Data structures: REWARDS, TIE, Howard
  - Control flow graphs: CFG generation
Figure 1. Our assembly function reuse approach. The core of our approach is the function extraction step implemented by BCR. The three dark gray modules in function extraction have been specifically designed for this work. The execution monitor, disassembler, and semantics inference module (light gray) are reused from previous systems.

- The function does not call system calls directly (e.g., through interrupt or `sysenter` instructions) but instead uses system calls only through well-known functions that are available in the target system where the function is reused (e.g., the standard C library, or the Windows API if the target system is Windows-based).
- The function contains no code that explicitly uses its own location. For example, the code should not check if it is loaded at a specific address or offset. This restriction excludes most self-modifying code. However, the function may still reference global addresses through standard position-independent code and dynamic linking: relocatable and non-relocatable code are both supported.

An important class of functions that usually satisfy these constraints are transformation functions. Transformation functions include encryption and decryption, compression and decompression, code packing and unpacking, checksums, and generally any function that encodes given data in a different format. Such functions are usually well-contained, have clean interfaces, limited side effects, and are interesting for many security applications.

Handling obfuscation. Our approach can be applied to both benign code and malware. When applying it to malware we need to consider the obfuscation techniques that malware often uses. Common obfuscation techniques used to hamper static analysis such as binary packing, adding unnecessary instructions, or replacing calls with indirect jumps do not affect our hybrid disassembly approach because it uses dynamic analysis to complement static disassembly. However, a premise of our approach is that we can observe a sample's execution in our analysis environment (based on a system emulator). Thus, like other dynamic approaches, our approach can be evaded using techniques that detect virtualized or emulated environments [27]. In addition, an adversary may make its code hard to reuse, for example, by mixing unrelated functionality, adding unnecessary parameters, or inlining functions. We have not seen such obfuscations in our examples. We consider them another instance of code that is hard to reuse and may require automated or manual analysis on top of our techniques.

2.4 Approach and System Architecture

Our assembly function reuse approach comprises three steps: dynamic analysis, hybrid disassembly, and function extraction. Figure 1 shows the three steps.

- In the dynamic analysis step the program is run inside the execution monitor, which is an emulator based on QEMU [13, 44] that can produce execution traces containing the executed instructions, the contents of the instructions' operands and optional taint information. The execution monitor tracks when execution reaches a given entry point and when it leaves the assembly function via an exit point. When an exit point is reached, the execution monitor produces a memory dump, i.e., a snapshot of the process memory address space. This step may be repeated to produce multiple execution traces and memory dumps.

Credit: This diagram is from the BCR paper in [NDSS 2010]
Inspector Gadget

Credit: This diagram is from the Inspector Gadget paper in [Oakland 2010]
We can reuse the legacy binary code to build new software.
References

- Boomerang http://boomerang.sourceforge.net/
- BCR [NDSS 2010]
- Inspector Gadget [Oakland 2010]
- Virtuoso [Oakland 2011]
- http://www.debugmode.com/dcompile/
Imagine a circle that contains all of human knowledge.

credit: http://matt.might.net/articles/phd-school-in-pictures/
By the time you finish elementary school, you know a little

credit: http://matt.might.net/articles/phd-school-in-pictures/
By the time you finish high school, you know a bit more

credit: http://matt.might.net/articles/phd-school-in-pictures/
With a bachelor’s degree, you gain a specialty

credit: http://matt.might.net/articles/phd-school-in-pictures/
A master’s degree deepens that specialty

credit: http://matt.might.net/articles/phd-school-in-pictures/
Reading research papers takes you to the edge of human knowledge

credit: http://matt.might.net/articles/phd-school-in-pictures/
Once you’re at the boundary, you focus

credit: http://matt.might.net/articles/phd-school-in-pictures/
You push at the boundary for a few years

credit: http://matt.might.net/articles/phd-school-in-pictures/
Until one day, the boundary gives way

credit: http://matt.might.net/articles/phd-school-in-pictures/
And, that dent you’ve made is called a Ph.D

credit: http://matt.might.net/articles/phd-school-in-pictures/
Of course, the world looks different to you now

credit: http://matt.might.net/articles/phd-school-in-pictures/
So, don’t forget the bigger picture

credit: http://matt.might.net/articles/phd-school-in-pictures/