

REVENGE is a dish served cold: Debug-Oriented Malware Decompilation and Reassembly

Marcus Botacin¹, Lucas Galante²,
Paulo Lício de Geus², André Grégio¹

¹Federal University of Paraná (UFPR-BR)
{mfbotacin, gregio}@inf.ufpr.br

²University of Campinas (UNICAMP-BR)
{galante, paulo}@lasca.ic.unicamp.br

Who Am I?

Background

- Computer Engineer (University of Campinas–Brazil).
- CS Master (University of Campinas–Brazil).
- CS PhD Student (Federal University of Paraná–Brazil).
- Malware Analyst (Since 2012).

Research Interests

- **Malware Analysis & Detection.**
- Hardware-Assisted Security.

Topics

- 1 Introduction
 - The Problem
 - Background
- 2 REVENGE
 - Overview
 - Architecture
- 3 Evaluation
 - Malware Decompilation
 - Malware Reassembly
- 4 Final Remarks
 - Limitations
 - Conclusion
 - Questions?

The Problem

Malware

- Hard to Understand at low level (e.g. assembly).

Decompilers

- Lift low level constructions to high level semantics.
- Allow API and/or source code analyses.

Decompilation Challenges

- Malware is not well-behaved.
- Malware implement anti-analysis tricks.
- Malware binaries present dead code.

Insights & Proposal (1/2)

Current Decompilers

- They perform reasonably well with small pieces of code.
- They do not perform well with static disassembly.

Current Debuggers

- They can perform dynamic disassembly and/or inspection.

Insights & Proposal (2/2)

Current Analysts' Tasks

- Analysts already debug binaries in a sliced manner.
- Analysts perform their own anti-anti-analysis routines.

What If?

- Could we combine analysts manual work with decompiler?
- And decompile the small pieces debugged by the analyst?
- And let the analyst to overcome anti-analysis by themselves?

Topics

- 1 Introduction
 - The Problem
 - Background
- 2 REVENGE
 - Overview
 - Architecture
- 3 Evaluation
 - Malware Decompilation
 - Malware Reassembly
- 4 Final Remarks
 - Limitations
 - Conclusion
 - Questions?

Background

Compiler

- Parsing, Pre-Processing, Assembling, Optimization, and Code Generation.

Decompiler

- Disassembly, Lifting, data type recovery, and Code Generation.

Notice:

- Not the same code generation routines.
- Decompiler is an inverse compiler.
- There are cross-platform compilers and decompilers.

The Challenges (1/2)

Disassembly

- Opaque Constants.
- Overlapping Instructions.
- Data and Code are mixed.

Lifting

- A typical ISA is **VERY** large.
 - Have you ever executed VFMADDSUBPS?
- and O.S. support as well...
 - Do you know what is NUMA?

The Challenges (2/2)

Data Type Reconstruction

- What's is the difference between an array (`int a[2];`) and consecutive variables (`int a,b;`)?
- Is `0x77FF...` an integer or a pointer?

Code Generation

- How to implement?
- Which optimizations?
- How to name variables?

Evaluation

- Is recovered code a good metric for malware decompilation?

Topics

- 1 Introduction
 - The Problem
 - Background
- 2 **REVENGE**
 - **Overview**
 - Architecture
- 3 Evaluation
 - Malware Decompilation
 - Malware Reassembly
- 4 Final Remarks
 - Limitations
 - Conclusion
 - Questions?

Reverse Engineering Engine

Overview

- PoC Decompiler focused on malware analysis.
- GDB-powered (no-reimplementation).
- Dynamic Inspection (no static analysis constraints).
- Trace-Oriented (decompile what is debugged).
- Reassembler (merge the decompiled pieces in a new software).

Topics

- 1 Introduction
 - The Problem
 - Background
- 2 **REVENGE**
 - Overview
 - **Architecture**
- 3 Evaluation
 - Malware Decompilation
 - Malware Reassembly
- 4 Final Remarks
 - Limitations
 - Conclusion
 - Questions?

REVENGE-GDB Integration

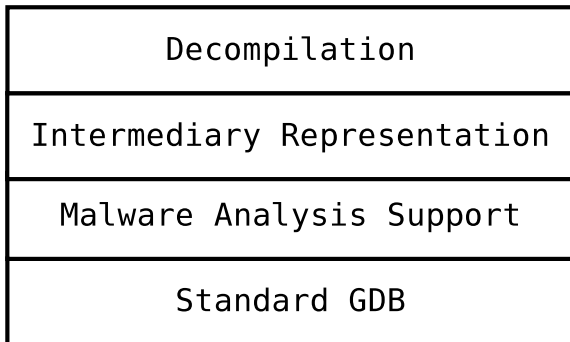


Figure: RevEngE Architecture. GDB provides the basic debugging capabilities and was armored to handle malware anti-analysis techniques. REVENGE decompiler is developed on top of the armored GDB.

GDB Armoring

```

1  __libc_start_main (main=<value>, argc=<value>,
    ubp_av=<value>, init=<value>, fini=<value>,
    rtdl_fini=<value>, stack_end=<value>)

```

Code Snippet 1: Libc Entry Point. First argument points to application entry point.

```

1  output = gdb.execute("set_␣$eflags|=0x%x" % self.
    flag_map[flag], to_string=True)

```

Code Snippet 2: Invert Branch Direction. Flags register is changed according a map of possible flags for such command.

Instruction Representation

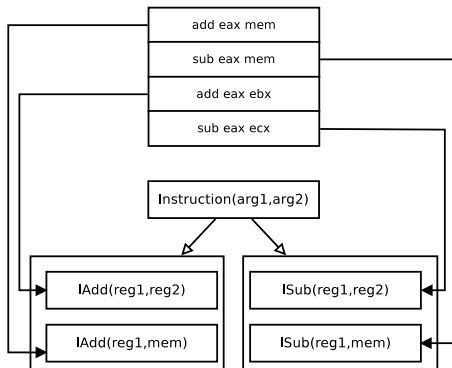


Figure: Instruction Representation. REVENGE benefits from Python's polymorphism to model instruction's behaviors and overloads method declarators to support each x86 instruction's possible multiple argument types.

Instruction Factory

```
1 class IFactory(...):
2     def get(self, args):
3         newclass = globals()[name](args)
4         return newclass
```

Code Snippet 3: Instruction Factory. The Factory design pattern allows instantiating objects from the proper class by exploring Python OOP capabilities.

```
1 self.classes['div'] = "IDiv"
2 self.classes['divl'] = "IDiv"
3 self.classes['idiv'] = "IDiv"
4 self.classes['idivl'] = "IDiv"
```

Code Snippet 4: Instruction Lifting. REVENGE assumes only signed integer operations to handle all instructions via the same high-level class.

Lifting Complex Instructions

```
1 0x4004eb cmp -0x8(%rbp),%eax
2 0x4004ee jle 4004fb <main+0x25>
```

Code Snippet 5: Low level representation of a conditional decision. IF instructions are composed by multiple assembly instructions.

```
1 class HighLevelCompare():
2     def __init__(self,cmp,set):
3         self.op1 = cmp.op1
4         self.op2 = cmp.op2
5         self.op3 = set.op3
```

Code Snippet 6: High level conditional decision representation. Assembly instructions are promoted to a single class that represents a high level conditional structure (e.g., IFs).

Handling Variables

```
1 self.vars = VariableManager()
2 self.vars.remove_registers(reg=arg1.get_operand())
3 self.vars.check_is_pointer(var.get_value())
```

Code Snippet 7: Variable Management. REVENGE does not handle variables directly but via a centralized manager to keep context consistent.

```
1 self.var = self.vars.new_var(reg="%eax")
2 self.var = self.vars.new_var(reg=arg1.get_operand(),
    value=val)
3 self.var = self.vars.new_var(value=arg1.get_value(),
    mem=arg2.get_operand())
```

Code Snippet 8: Variable Manager. Context complexity is encapsulated by the manager, thus releasing REVENGE to focus on decompilation logic.

Variable Disambiguation

```
1  main  movl  $0xF  -0x4(%rbp)
2  NAME:  [var0]
3  VAL:   [0xF]
4  REG:   [NONE]
5  MEM:   [7fffffffdc7c]
6
7  main  mov  -0x8(%rbp) %eax
8  NAME:  [var0]
9  VAL:   [0xF]
10 REG:   [NONE]
11 MEM:   [7fffffffdc7c]
```

Code Snippet 9: Memory References Disambiguation. Variables are referenced by their memory addresses instead of pointed registers.

Function Introspection

```
1 printf@stdio.h: int printf ( const char * format,  
    ... ); (Return: int) (N_Args: 2)
```

Code Snippet 10: Introspection Procedure. External function prototypes are identified by searching for function and library names on the Internet and parsing them to a format suitable for REVENGE decompilation.

Code Generation

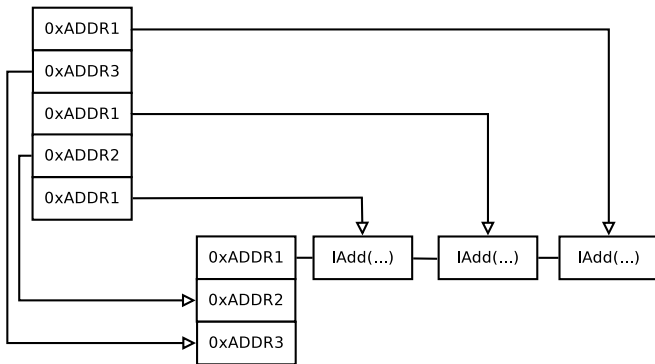


Figure: Code Generation. REVENGE keeps distinct objects for the same instruction address, thus representing the multiple calling contexts. Loop unrolling is performed by removing the top of stack each time a given instruction address is referred.

Topics

- 1 Introduction
 - The Problem
 - Background
- 2 REVENGE
 - Overview
 - Architecture
- 3 Evaluation
 - Malware Decompileation
 - Malware Reassembly
- 4 Final Remarks
 - Limitations
 - Conclusion
 - Questions?

Instructions Per Binary

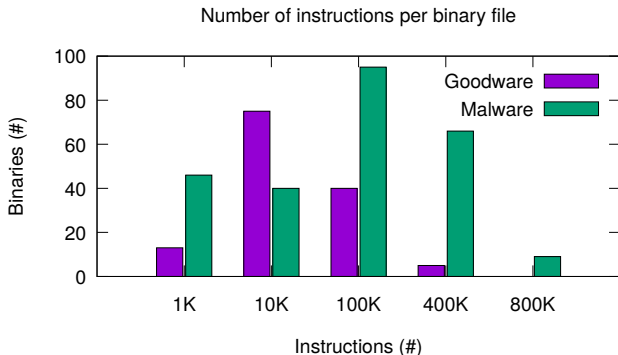


Figure: Number of instructions per binary. Malware samples executed more instructions than goodware samples.

Handled Instructions per binary

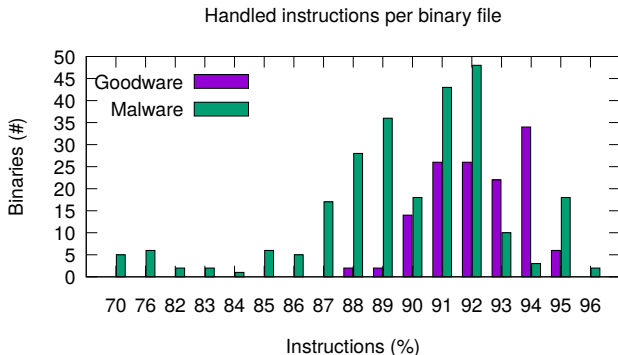


Figure: Handled instructions per binary. Most binaries were successfully handled. Malware samples impose greater challenges than goodware samples.

Topics

- 1 Introduction
 - The Problem
 - Background
- 2 REVENGE
 - Overview
 - Architecture
- 3 Evaluation
 - Malware Decompilation
 - Malware Reassembly
- 4 Final Remarks
 - Limitations
 - Conclusion
 - Questions?

Tsunami/Backdoor

```
1  call 0x8048dfc <rand@plt>
2  mov  %eax,%ecx  mov    $0x66666667,%eax
3  imul %ecx      sar    %edx
4  mov  %ecx,%eax  sar    $0x1f,%eax
5  sub  %eax,%edx  mov    %edx,%eax
6  shl  $0x2,%eax
```

Code Snippet 11: Tsunami/Backdoor. Assembly code for the traced function.

Tsunami/Backdoor

```
1 void makestring(char *var3) {
2     int var1=0, var2=MAX_STRING,
3     var6=0x666667, var9=0x1f, var12=2;
4     for(var4=var1;var4<var2;var4++){
5         var5=rand();          var7=var6/var5;
6         var8=var6%var5;      var10=var7>>var9;
7         var11=var8-var10;    var13=var11<<var12;
8         var3[var4]=var13;
```

Code Snippet 12: Tsunami/Backdoor. Decompiled code function.

Exploit/Trojan

```
1  call 0x80484b4 <atoi@plt>
2  add  $0x10,%esp  mov  %eax,%eax
3  mov  %eax,%eax  mov  %eax,-0x18(%ebp)
4  cml  $0x2,0x8(%ebp)
5  jle  0x804862a <main+90>
6  push $0x1  call 0x80484a4 <exit@plt>
```

Code Snippet 13: Exploit/Trojan. Assembly code for the traced function.

Exploit/Trojan

```
1 char var1[MAX_STRING];
2 int var2=0, var3=3, var4=1,
3   var6=0xf, var7=2, var8=0xff;
4 if(argc==var3){ var5=atoi(argv[var4]);
5   if(var5==var6){ var5=atoi(argv[var7]);
6     if(var5==var8){
```

Code Snippet 14: Exploit/Trojan. Decompiled code function.

Micmp/Backdoor

```
1  call 0x8048734 <time@plt>
2  add  $0x4,%esp  push %eax
3  call 0x8048794 <srnd@plt>
4  add  $0x10,%esp sub  $0x4,%esp
5  sub  $0xc,%esp  call 0x8048814 <rand@plt>
6  add  $0xc,%esp  mov  %eax,%edx
7  sar  $0x1f,%edx idiv %ecx
```

Code Snippet 15: Micmp/Backdoor. Assembly code for the traced function.

Micmp/Backdoor

```
1 void return_randip(char *var1){
2   int var3=0xB;   srand(time(NULL));
3   var2 = rand(); var4 = var2 / var3;
4   var5 = rand(); var6 = var5 / var3;
5   var7 = rand(); var8 = var7 / var3;
6   var9 = rand(); var10 = var9 / var3;
7   sprintf(var1, "%d.%d.%d.%d", var...);
```

Code Snippet 16: Micmp/Backdoor. Decompiled code function.

Small/Backdoor

```
1  movl $0x8049798 ,(%esp)
2  call 0x80487a8 <system@plt>
3  movl $0x80497bb ,(%esp)
4  call 0x80487a8 <system@plt>
```

Code Snippet 17: Small/Backdoor. Assembly code for the traced function.

Small/Backdoor

```
1 void open_firewall(){
2     char var1 []="iptables-F-INPUT";
3     char var2 []="iptables-P-INPUT-ACCEPT";
4     system(var1); system(var2);
```

Code Snippet 18: Small/Backdoor. Decompiled code function.

RST/Virus

```
1  call 0x804a104 <openlog@plt>
2  push %ebx push $0x806f5e7 push $0x7
3  call 0x8049fa4 <syslog@plt>
4  call 0x804a1b4 <closelog@plt>
5  <userfile_remove>:
6  call 8049f54 <remove@plt>
```

Code Snippet 19: RST/Virus. Assembly code for the traced function.

RST/Virus

```
1  int debug(){
2      FILE *var1;
3      char var2 []="/var/log/syslog",
4      char var4 []="r";
5      int var3=0;
6      var1 = fopen(var2,var4);
7      if(var1){ var3=1; }
8      return var3;
```

Code Snippet 20: RST/Virus. Decompiled code function.

Topics

- 1 Introduction
 - The Problem
 - Background
- 2 REVENGE
 - Overview
 - Architecture
- 3 Evaluation
 - Malware Decompilation
 - Malware Reassembly
- 4 Final Remarks
 - Limitations
 - Conclusion
 - Questions?

Limitations & Future Work

Limitations

- Proof-of-Concept (PoC) for future developments.
- Limited instruction set (x86, no floats).
- C-like binaries only.
- ELF binaries only.

Future Work

- Implementing REVENGE in a real decompiler.
- Radare2? IDA/HexRays? What else?

Topics

- 1 Introduction
 - The Problem
 - Background
- 2 REVENGE
 - Overview
 - Architecture
- 3 Evaluation
 - Malware Decompilation
 - Malware Reassembly
- 4 Final Remarks
 - Limitations
 - **Conclusion**
 - Questions?

Conclusion

Take Aways

- Decompilers enable high-level analyses.
- Full semantic reconstruction is challenging.
- We know how to decompile small pieces of code.
- Analysts already debug sliced binaries.
- Moving towards trace-driven decompilation is the right move!

Try REVENGE (1/2)

`https://github.com/marcusbotacin/Reverse.Engineering.Engine`

The screenshot shows the GitHub repository page for 'Reverse.Engineering.Engine' by 'marcusbotacin'. The repository has 1 watch, 3 stars, and 1 fork. It contains 4 commits, 1 branch, 0 packages, 0 releases, and 1 contributor. The repository is on the 'master' branch. The file list includes:

File	Commit	Time
bins	RevEngE 1.0	5 months ago
figs	updated version	3 months ago
paper	updated version	3 months ago
sources	RevEngE 1.0	5 months ago
src	RevEngE 1.0	5 months ago
tests	RevEngE 1.0	5 months ago
LICENSE	Initial commit	5 months ago
README.md	Readme fix	2 months ago
install.sh	RevEngE 1.0	5 months ago
revenge.py	RevEngE 1.0	5 months ago

Figure: REVENGE source code.

Try REVENGE (2/2)

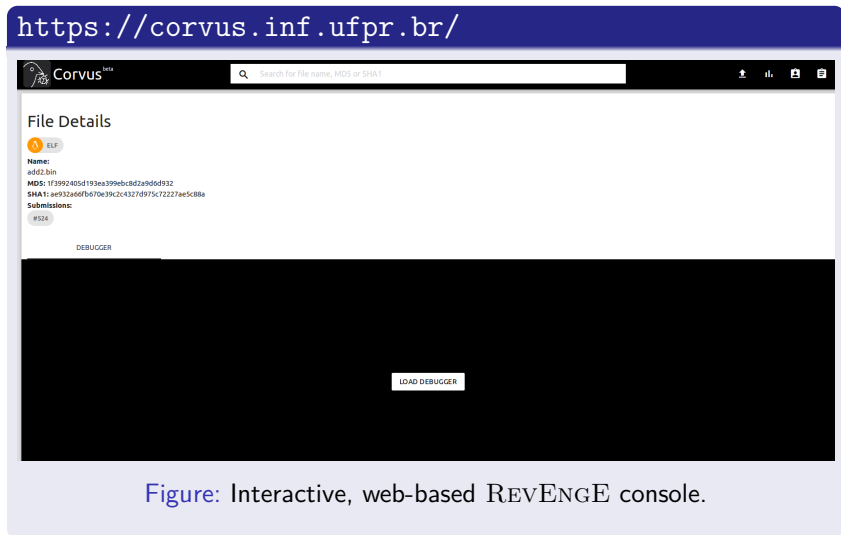


Figure: Interactive, web-based REVENGE console.

Questions?

Topics

- 1 Introduction
 - The Problem
 - Background
- 2 REVENGE
 - Overview
 - Architecture
- 3 Evaluation
 - Malware Decompilation
 - Malware Reassembly
- 4 Final Remarks
 - Limitations
 - Conclusion
 - Questions?

Contact

mfbotacin@inf.ufpr.br
@MarcusBotacin