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

Marcus Botacin[1], Lucas Galante[2],
Paulo Lício de Geus[2], André Grégio[1]

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

[2]University 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

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 do not behave well.
- Malware implement anti-analysis tricks.
- Malware binaries exhibit 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.

State-of-the-art Decompilers

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...

- We could combine analysts manual work with decompiler?
- We could decompile the small pieces debugged by the analyst?
- We could allow the analyst to overcome anti-analysis by themselves?

Topics

Background

Compiler

- Parsing, Pre-Processing, Assemblying, Optimization, Linking and Code Generation.

Decompiler

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

Notice that:

- 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 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

RevEngE-GDB Integration

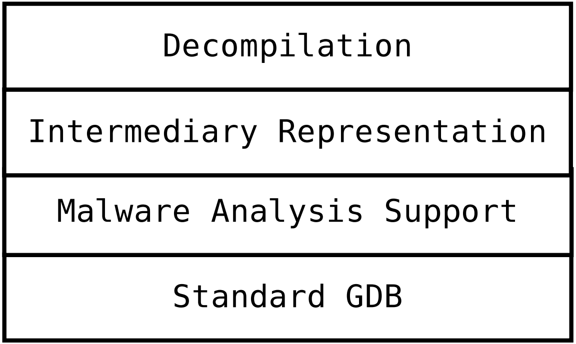

Figure: **RevEngE Architecture.** GDB provides the basic debugging capabilities and was armored to handle malware anti-analysis techniques. RevEngE decompiler is developed on top of the armored GDB.

GDB Armoring

```
__libc_start_main (main=<value>, argc=<value>,
    ubp_av=<value>, init=<value>, fini=<value>,
    rtld_fini=<value>, stack_end=<value>)
```

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

```
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

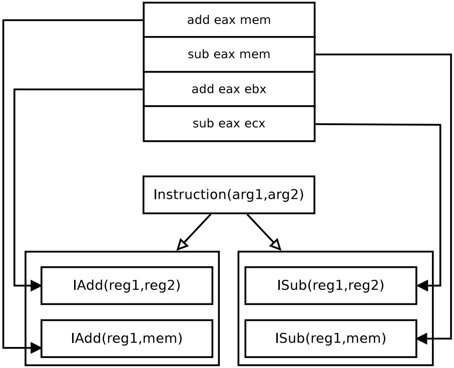

Figure: **Instruction Representation**. REVENGE benefits from Python's polymorphism to model instruction's behaviors and overloads method declarators to support each x86 instruction's possible multiple argument types.

Instruction Factory

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

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

```
self.classes['div']   = "IDiv"
self.classes['divl']  = "IDiv"
self.classes['idiv']  = "IDiv"
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

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

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

```
class HighLevelCompare():
    def __init__ (self,cmp,set):
        self.op1 = cmp.op1
        self.op2 = cmp.op2
        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

```
self.vars = VariableManager()
self.vars.remove_registers(reg=arg1.get_operand())
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.

```
self.var = self.vars.new_var(reg="%eax")
self.var = self.vars.new_var(reg=arg1.get_operand(),
    value=val)
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

```
main movl $0xF -0x4(%rbp)
NAME: [var0]
VAL:  [0xF]
REG:  [NONE]
MEM:  [7fffffffdc7c]

main mov -0x8(%rbp) %eax
NAME: [var0]
VAL:  [0xF]
REG:  [NONE]
MEM:  [7fffffffdc7c]
```

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

Function Introspection

```
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

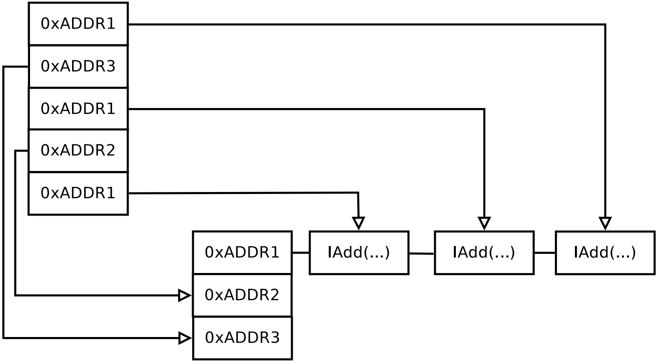

Figure: **Code Generation.** REVENGE keeps distinct objects for the same instruction address, thus representing the multiple calling contexts. Loop unrolling is performed by removing the top of stack each time a given instruction address is referred.

Topics

Instructions Per Binary

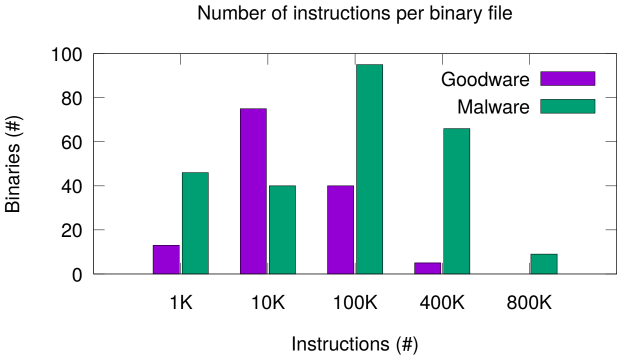

Figure: **Number of instructions per binary.** Malware samples executed more instructions than goodware samples.

Handled Instructions per binary

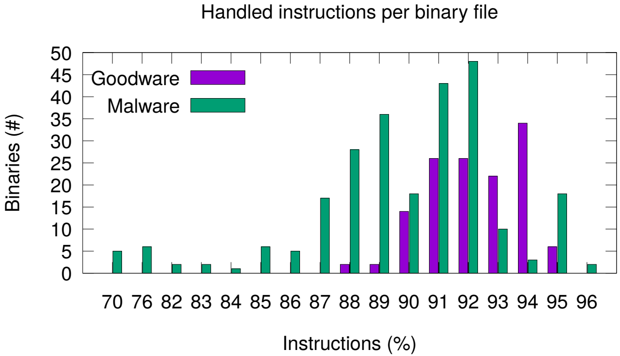

Figure: **Handled instructions per binary.** Most binaries were successfully handled. Malware samples impose greater challenges than goodware samples.

Topics

Tsunami/Backdoor

```
call 0x8048dfc <rand@plt>
mov  %eax,%ecx mov    $0x66666667,%eax
imul %ecx      sar    %edx
mov  %ecx,%eax sar    $0x1f,%eax
sub  %eax,%edx mov    %edx,%eax
shl  $0x2,%eax
```

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

Tsunami/Backdoor

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

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

Exploit/Trojan

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

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

Exploit/Trojan

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

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

Micmp/Backdoor

```
call 0x8048734 <time@plt>
add  $0x4,%esp  push %eax
call 0x8048794 <srand@plt>
add  $0x10,%esp sub  $0x4,%esp
sub  $0xc,%esp  call 0x8048814 <rand@plt>
add  $0xc,%esp  mov  %eax,%edx
sar  $0x1f,%edx idiv %ecx
```

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

Micmp/Backdoor

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

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

Small/Backdoor

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

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

Small/Backdoor

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

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

RST/Virus

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

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

RST/Virus

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

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

Reassembled Malware Detection

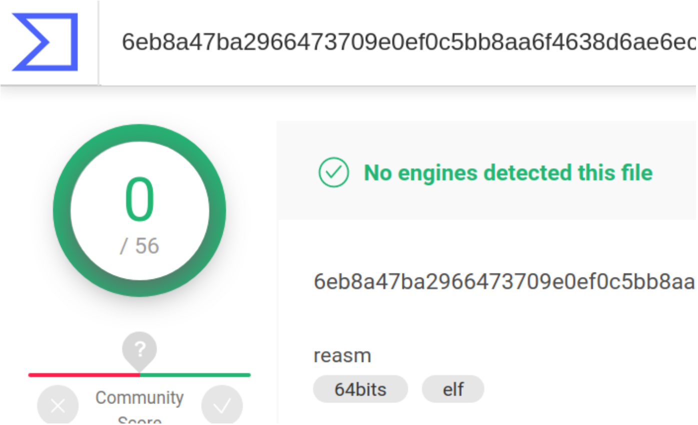

Figure: No AV detected the reassembled malware sample.

Topics

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

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

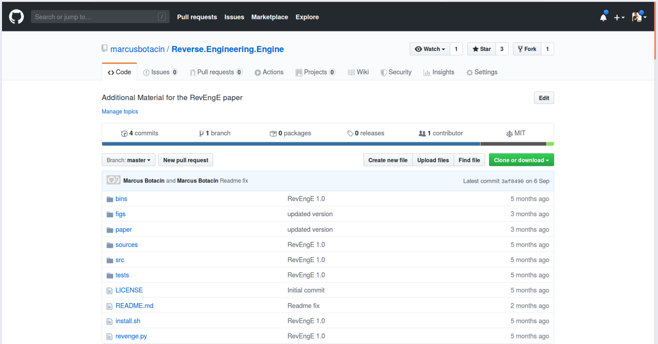

Figure: REVENGE source code.

Try REVENGE (2/2

https://corvus.inf.ufpr.br/

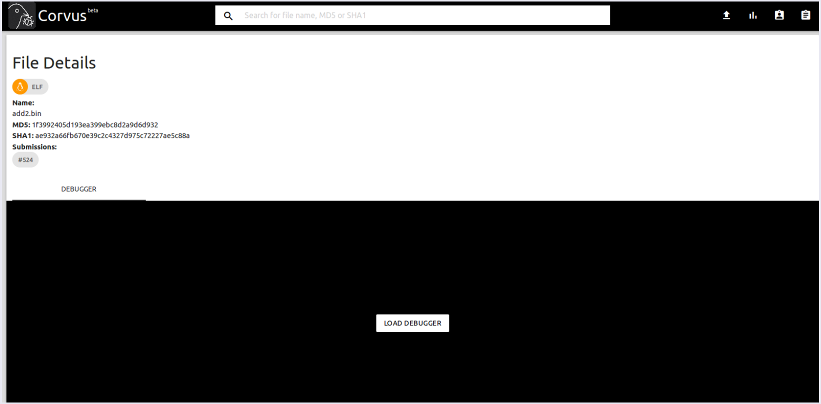

Figure: Interactive, web-based REVENGE console.

Topics

Contact

mfbotacin@inf.ufpr.br
@MarcusBotacin