Università degli Studi di Milano
Facoltà di Scienze Matematiche, Fisiche e Naturali

A Smart Fuzzer for x86 Executables

Andrea Lanzi, Lorenzo Martignoni, Mattia Monga,
Roberto Paleari

May 19, 2007

Motivation and Goal

Motivation

Problem: automatic detection of security vulnerabilities in computer software

- *closed source* software is widely spread
- actual computer programs are rather complex

⇒ need of new tools able to *automatically* analyze *executable* code

Goal

Design and development of a new analysis model able to identify security relevant flaws (i.e. sensitive information overwritten with input-related data) in *stripped* executable code

Previous Approaches

Exhaustive input testing

- The program is executed on *every* possible input data
- *Black-box*
- Input space is virtually unbounded

Fuzzing

- Randomly generated input data
- *Black-box*
- Incomplete coverage

Other analysis techniques

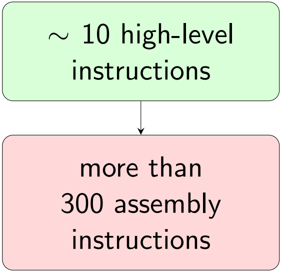
- Symbolic execution, static analysis, . . .
- Interesting results, but with scalability problems (exacerbated at the executable code level)

Example

```
void copy(char *src)
{
    char buf[5];
    int i;

    if(strncmp(src, ”abcd”, 4)) {
        printf(”error\n”);
        return;
    }

    for(i=0; src[i]; i++)
        buf[i] = src[i];
}
```

Example

```c
void copy(char *src)
{
   char buf[5];
   int i;

   if(strncmp(src, "abcd", 4)) {
      printf("error\n");
      return;
   }

   for(i=0; src[i]; i++)
      buf[i] = src[i];
}
```



~ 10 high-level instructions

more than 300 assembly instructions

Analysis of Executable Code

Motivations

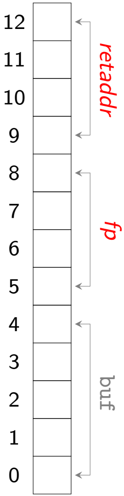
- Many applications are only available as executable code
- Many vulnerabilities depend on low-level details (e.g. memory layout)

Problems

- Explosion of code complexity
- High-level information must be completely rebuilt
- Need to handle a lot of details concerning the underlying architecture, operating system and compiler

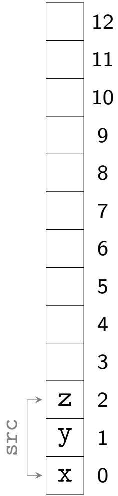
Smart Fuzzing

Example



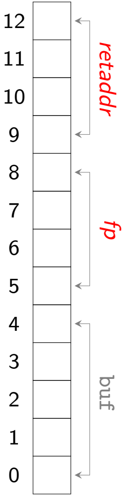
```
void copy(char *src)
{
  char buf[5];
  int i;
  if(strncmp(src, "abcd", 4)) {
    printf("error\n");
    return;
  }
  for(i=0; src[i]; i++)
    buf[i] = src[i];
}
```

$\mathcal{PC} = \emptyset$



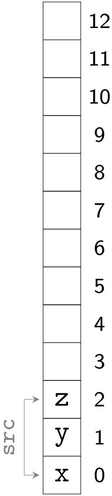
Smart Fuzzing

Example



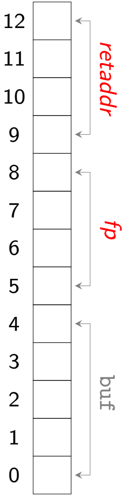
```
void copy(char *src)
{
  char buf[5];
  int i;
  if(strncmp(src, "abcd", 4)) {
    printf("error\n");
    return;
  }
  for(i=0; src[i]; i++)
    buf[i] = src[i];
}
```

$$\mathcal{PC} = \{\texttt{strncmp(src,"abcd",4)} \neq 0\}$$



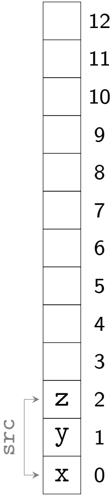
Smart Fuzzing

Example



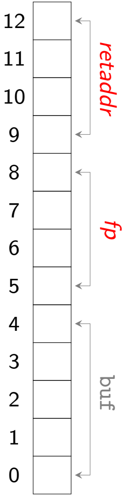
```
void copy(char *src)
{
  char buf[5];
  int i;
  if(strncmp(src, "abcd", 4)) {
    printf("error\n");
    return;
  }
  for(i=0; src[i]; i++)
    buf[i] = src[i];
}
```

$$\mathcal{PC} = \{\texttt{strncmp(src,"abcd",4)} \neq 0\}$$



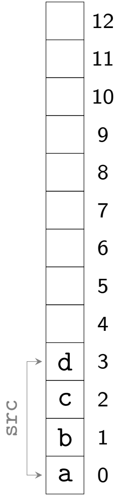
Smart Fuzzing

Example



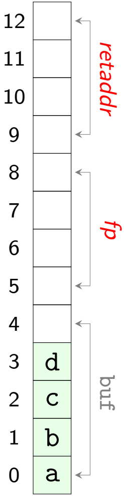
```
void copy(char *src)
{
  char buf[5];
  int i;
  if(strncmp(src, "abcd", 4)) {
    printf("error\n");
    return;
  }
  for(i=0; src[i]; i++)
    buf[i] = src[i];
}
```

$\mathcal{PC} = \emptyset$



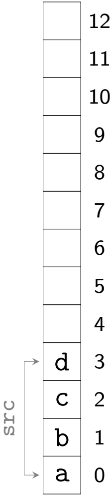
Smart Fuzzing

Example



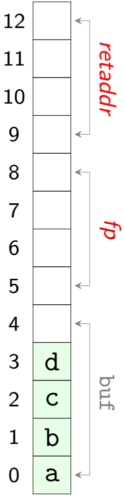
```
void copy(char *src)
{
  char buf[5];
  int i;
  if(strncmp(src, "abcd", 4)) {
    printf("error\n");
    return;
  }
  for(i=0; src[i]; i++)
    buf[i] = src[i];
}
```

$$\mathcal{PC} = \{\texttt{strncmp(src,"abcd",4)} = 0\}$$



Smart Fuzzing

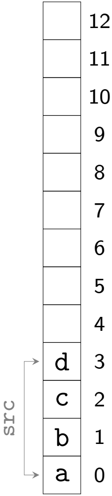
Example



```
void copy(char *src)
{
  char buf[5];
  int i;
  if(strncmp(src, "abcd", 4)) {
    printf("error\n");
    return;
  }
  for(i=0; src[i]; i++)
    buf[i] = src[i];
}
```

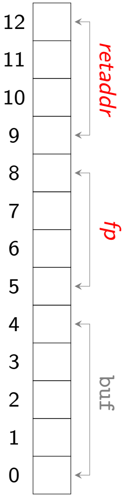
loop analysis

$$\mathcal{PC} = \{\texttt{strncmp(src,"abcd",4)} = 0 \wedge \texttt{strlen(src)} \geq 13\}$$

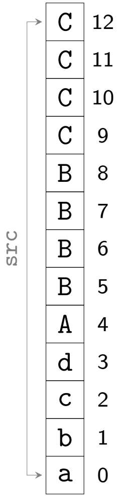


Smart Fuzzing

Example



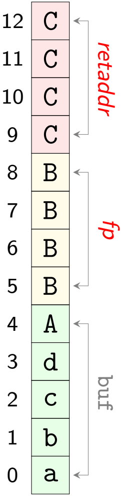
```
void copy(char *src)
{
  char buf[5];
  int i;
  if(strncmp(src, "abcd", 4)) {
    printf("error\n");
    return;
  }
  for(i=0; src[i]; i++)
    buf[i] = src[i];
}
```



$\mathcal{PC} = \emptyset$

Smart Fuzzing

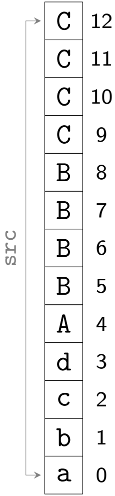
Example



12	C
11	C
10	C
9	C
8	B
7	B
6	B
5	B
4	A
3	d
2	c
1	b
0	a

```
void copy(char *src)
{
  char buf[5];
  int i;
  if(strncmp(src, "abcd", 4)) {
    printf("error\n");
    return;
  }
  for(i=0; src[i]; i++)
    buf[i] = src[i];
}
```

$$\mathcal{PC} = \{\texttt{strncmp(src,"abcd",4)} = 0\}$$



C	12
C	11
C	10
C	9
B	8
B	7
B	6
B	5
A	4
d	3
c	2
b	1
a	0

Smart Fuzzing

1. *Hybrid* analysis of the executable code to infer relationships between input data and program behavior:
    - *static* analysis: sound but often overly conservative
    - *dynamic* analysis: unsound but accurate
2. Generate new input data in order to drive the execution towards "dangerous" paths (from the security point of view)
3. Execution monitoring to detect the overwriting of sensitive information with untrusted data

Challenges

CISC architecture

Intel IA-32 includes over 300 different *opcodes*

intermediate representation (4 instructions, 5 expressions, *side effects* are made explicit)

Conditional predicates

Infer existing relationship between conditional predicates on processor control registers and input data

reconstruction of input-dependent conditional predicates

Loops

Programs that contain loops could also have an infinite number of execution paths

abstraction of loop behavior

+

heuristics

Challenges

CISC architecture

Intel IA-32 includes over 300 different *opcodes*

intermediate representation (4 instructions, 5 expressions, *side effects* are made explicit)

Conditional predicates

Infer existing relationship between conditional predicates on processor control registers and input data

reconstruction of input-dependent conditional predicates

Loops

Programs that contain loops could also have an infinite number of execution paths

abstraction of loop behavior
+
heuristics

Reconstruction of Conditional Predicates

1. disassembly
2. intermediate representation
3. predicate analysis
4. path analysis

Example: `strncmp(src, "abcd", 4)`, first iteration

```
je 0x08048394
```

Reconstruction of Conditional Predicates

static

1. disassembly
2. intermediate representation →
3. predicate analysis
4. path analysis

- the evaluated condition is made explicit

Example: `strncmp(src, "abcd", 4)`, first iteration

```
JMP (r1(ZF) == c1(0x1)) c32(0x08048394)
```

Reconstruction of Conditional Predicates

static

1. disassembly
2. intermediate representation
3. predicate analysis →
4. path analysis

- dynamic propagation of control flags reaching definitions
- expression simplification

Example: `strncmp(src, "abcd", 4)`, first iteration

```
JMP (m8[(r16(DS)+r32(ESI))] == m8[(r16(ES)+r32(EDI))]) c32(0x08048394)
```

Reconstruction of Conditional Predicates

static

1. disassembly
2. intermediate representation
3. predicate analysis
4. path analysis →

dynamic

- recursive propagation of dynamic reaching definitions
- subexpression propagation is halted when an input-related variable is found

Example: `strncmp(src, "abcd", 4)`, first iteration

```
JMP (m8[c32(0xbfffffdc)] == c8(0x61)) c32(0x08048394)
```

Loop Behavior Abstraction

1. loops identification (*performed statically*)
2. identification of induction variables used in loop condition
3. search for "*dangerous*" memory assignments in loop's body
4. guess of the number of iterations required in order to overwrite the nearest sensitive memory area

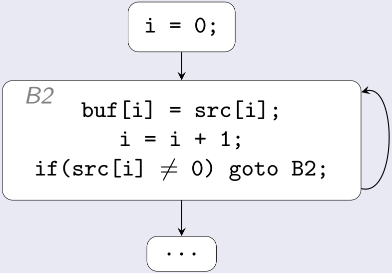
Example: *for* loop, from the `copy()` procedure

```
          ...
for(i=0; src[i]; i++)
    buf[i] = src[i];
          ...
```

Loop Behavior Abstraction

1. loops identification (*performed statically*)
2. identification of induction variables used in loop condition
3. search for "*dangerous*" memory assignments in loop's body
4. guess of the number of iterations required in order to overwrite the nearest sensitive memory area

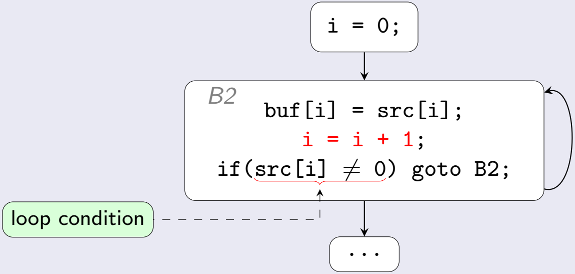
Example: *for* loop, from the `copy()` procedure



Loop Behavior Abstraction

1. loops identification (*performed statically*)
2. identification of induction variables used in loop condition
3. search for "*dangerous*" memory assignments in loop's body
4. guess of the number of iterations required in order to overwrite the nearest sensitive memory area

Example: *for* loop, from the copy() procedure



```
i = 0;
```

B2

```
buf[i] = src[i];
i = i + 1;
if(src[i] ≠ 0) goto B2;
```

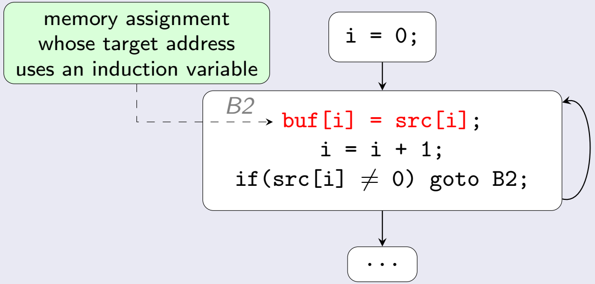
loop condition

```
...
```

Loop Behavior Abstraction

1. loops identification (*performed statically*)
2. identification of induction variables used in loop condition
3. search for "*dangerous*" memory assignments in loop's body
4. guess of the number of iterations required in order to overwrite the nearest sensitive memory area

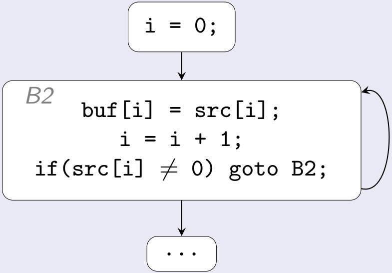
Example: *for* loop, from the copy() procedure


memory assignment whose target address uses an induction variable

Loop Behavior Abstraction

1. loops identification (*performed statically*)
2. identification of induction variables used in loop condition
3. search for "*dangerous*" memory assignments in loop's body
4. guess of the number of iterations required in order to overwrite the nearest sensitive memory area

Example: *for* loop, from the `copy()` procedure



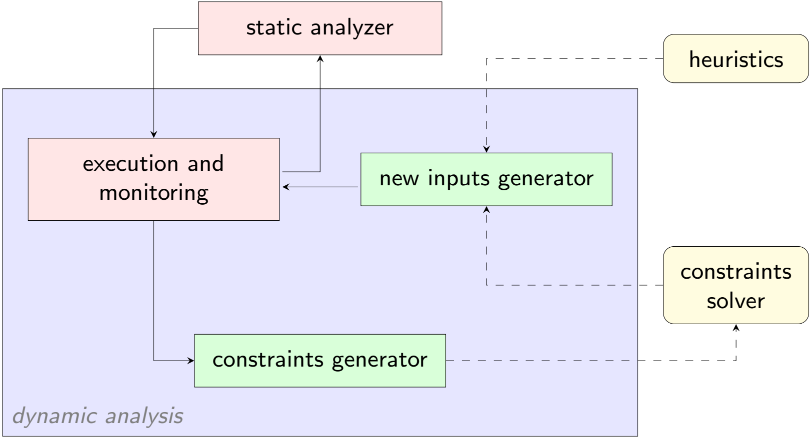
```
i = 0;
```

B2

```
buf[i] = src[i];
i = i + 1;
if(src[i] ≠ 0) goto B2;
```

```
...
```

Architecture



static analyzer

heuristics

execution and monitoring

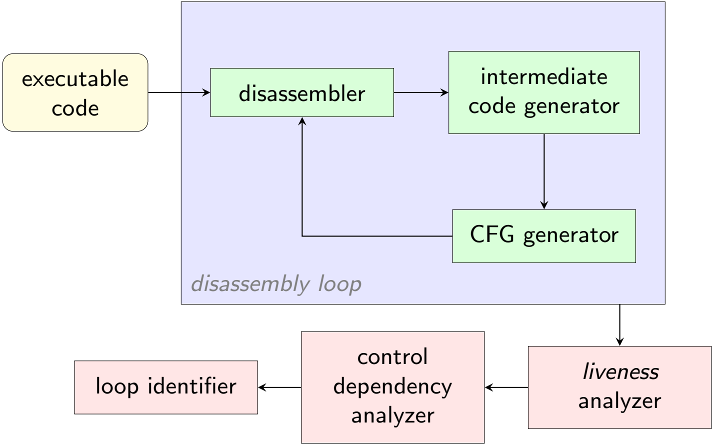
new inputs generator

constraints solver

constraints generator

dynamic analysis

Static Analysis



executable code

disassembler

intermediate code generator

CFG generator

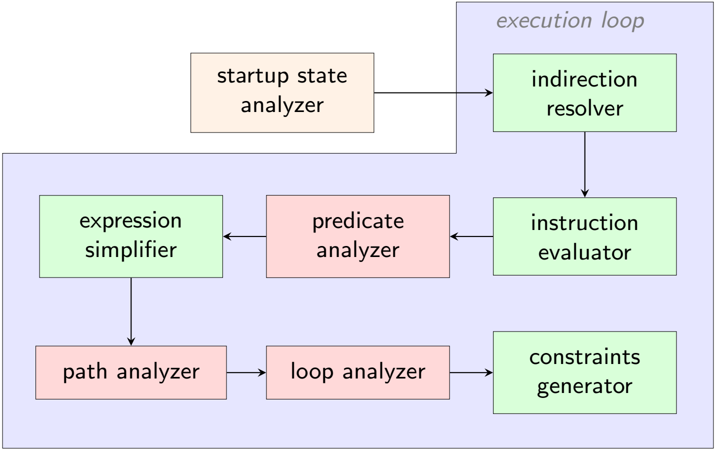
disassembly loop

liveness analyzer

control dependency analyzer

loop identifier

Dynamic Analysis



execution loop

startup state analyzer

indirection resolver

instruction evaluator

predicate analyzer

expression simplifier

path analyzer

loop analyzer

constraints generator

Prototype

Implementation

- Able to analyze ELF executables compiled with GCC, running on Linux operating system and IA-32 platform
    - $\sim$ 20000 lines of Python code
    - $\sim$ 1000 lines of C code
- `ptrace()`-based dynamic analysis

Problem: efficiency

Analyses are computationally expensive ($\sim$ 50 seconds for `/bin/ls`).

Solutions:

- *dynamic instrumentation* instead of debugging
- native compilation (Python → C)

Conclusions

Contributions

- New *smart fuzzing* model for the automatic detection of security relevant flaws in executable code
- Prototype tool (*open source* infrastructure for the analysis of executable code)
- Adaption of *program analysis* techniques to executable code
- New algorithms for loop and jump conditions analysis

Future work

- Our prototype must still be completed
- Use our prototype for finding previously unknown vulnerabilities
- Improve model precision and prototype performances
- Can our approach be applied to malware analysis? (i.e. need to support self modifying code, . . .)

Thank you! Questions?