A generic approach to the definition of low-level components for multi-architecture binary analysis

Cédric Valensi

PhD advisor: William Jalby

University of Versailles Saint-Quentin-en-Yvelines, France
Exascale Computing Research Center, France
LRC ITACA, France

July 2nd, 2014
High Performance Computing

Supercomputers
- Front-line of the computing capacity
- Multiprocessor systems
- Current top speed 33 Petaflop/s

Applications
- Physical simulations
- Natural resources exploration
- Molecular modeling
- Weather forecasts
Performance analysis for optimisation

- Optimising performance of HPC applications
  - Optimise use of processors in terms of speed and power
  - Pinpoint bottlenecks
  - Estimate gain from improvements

- Performance analysis
  - Static or dynamic
  - Instrumentation
  - Possible in all steps of the design process
Steps of an application design process

Algorithm

Source code

Compiler
Internal representation

Assembly code

Binary executable

myfunction:
	mov $6,%r0
.L1: lea (%rip, %r0, 4),%rdx
	mov (%rip, %rdx, 32),%r0
	cmp $42,%r0
	jle .L3

```
while (is_null != true) {
    let earth = DEVS.startWith:vector:
    if (range == "M1")
        resistance &= M1Fe
    float.fierz = any * firo;

01100111 10000111 00000110 01100111 11000100
10001101 00001100 11101110 01111111 11000111 11000111
00000100 00000010 10000011 11111010 00000111
```
Steps of an application design process

Introduction
Multi architecture support
Disassembly of binary files
Binary rewriting
Conclusion

Algorithm

Source code

Compiler
Internal representation

Assembly code

Binary executable

```
with (result = 42) {
let result = Double(result);,
if (op == "add")
resistance &= False;
float/float = any * any;
}
```

```
myfunction:
mov $6,%rcx
.L1:Jos (msg, msgi:4),9or9d
mov ($8,%r3,2),%rcx
cmp $42,%rcx
je  L3
```
Steps of an application design process

**Algorithm**

**Source code**

```c
while (r < b) {
    let earth = earth - 2
    if (r > b) {
        resistance = 5 + 10*b
    }
    else earth = any * r

myfunction:
  mov $6,%r0c
  .L1: lia (#edi,#esi,4),%rdx
  mov (%rdx,%edi,3),%r0c
  cmp $42,%r0c
  jle .L3
```

**Compiler**

**Internal representation**

**Assembly code**

```
01100111 10001111 00000100 01000111 01000100
10011011 00001100 10110010 01100011 10000101
00000100 00000110 10000011 11111011 00001111
```

**Binary executable**
Introduction
Multi architecture support
Disassembly of binary files
Binary rewriting
Conclusion

Steps of an application design process

Algorithm

Source code

Compiler
Internal representation

Assembly code

 Binary executable
Steps of an application design process

Algorithm

Source code

Assembly code

Compiler

Internal representation

Binary executable

```
myfunction:
    mov $6, %r0
.L1: lea (%esi, %esi, 8), %rdx
    mov (%rdx, %rdx, 32), %r0
    cmp $42, %r0
    jle .L1
```

```c
while (vector != NULL)
    if (vector->type == "TRUE")
        resistance = 1.0;
    float divider = any * n;
```
Performance analysis levels

- Knowledge of source language
- Requires access to source files
-Compilation may perform complex transformations
- Instrumenting at the source level may modify these transformations
Performance analysis levels

Compiler Internal Representation

- More accurate
- Requires access to compiler internals
- Requires intrusion into compilation process
- Ineffective for code written in assembly
Performance analysis levels

Assembly analysis

- Closer to the actual executable
- Not available by default
- Requires intrusion into compilation process
Performance analysis levels

Binary analysis

- “What you see is what you run”
- Allows to retrieve additional information
- More complex
Challenges of binary analysis

Dependent on the architecture
- Multiple architectures may be used by a single application
- Binary architectures evolve frequently

Static Analysis
- Requires disassembly of binary code

Instrumentation
- Requires static or dynamic patching
- Extensive changes can be needed
## Contribution

<table>
<thead>
<tr>
<th>Low level binary encoder and decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Able to support multiple architectures</td>
</tr>
<tr>
<td>- Minimised implementation workload</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage in analysis context</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Customisable behaviour</td>
</tr>
<tr>
<td>- Unified output format</td>
</tr>
<tr>
<td>- Acceptable performance</td>
</tr>
<tr>
<td>- Static analysis and instrumentation</td>
</tr>
</tbody>
</table>
## Objectives

<table>
<thead>
<tr>
<th>Generic encoder and decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Multi-architecture support</td>
</tr>
<tr>
<td>- Customisable output and behaviour</td>
</tr>
<tr>
<td>- Reduced implementation workload</td>
</tr>
</tbody>
</table>

## Challenges

- Complex binary coding rules
- Coding rules and assembly vary significantly between architectures
- Avoid hard coding
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 $\iff$ mov %r9, 0x20(%eax,%edx,4)

01100111 01001100 10001001 01001100 10010000 00100000

mov %r9,0x20 (%eax,%edx,4)
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)

01100111 01001100 10001001 01001100 10 010000 00100000

mov %r9,0x20 (%eax,%edx,4)
Example: Encoding of an Intel 64 instruction

\[ 0x66 49 89 4C 90 20 \leftrightarrow \text{mov} %r9, 0x20(%eax,%edx,4) \]

\[
\begin{array}{cccccccc}
01100111 & 01001100 & 10001001 & 01001100 & 10010000 & 00100000 \\
\end{array}
\]

\[ \text{mov} \ %9,0x20 \ (%eax,%edx,4) \]
Example: Encoding of an Intel 64 instruction

\[
0x66\ 49\ 89\ 4C\ 90\ 20 \iff \text{mov} \ %r9,\ 0x20(\%eax,\%edx,4)
\]

```
01100111 01001100 10001001 01001100 10010000 00100000
```

```
\text{mov} \ %r9, 0x20(\%eax,\%edx,4)
```
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)

01100111 01001100 10001001 01001100 10 010000 00100000

mov %9,0x20 (%eax,%edx,4)
Example: Encoding of an Intel 64 instruction

\[ \text{0x66 49 89 4C 90 20} \leftrightarrow \text{mov} \ %r9, \ 0x20(\%eax,\%edx,4) \]

\[ \begin{array}{cccccccc}
01100111 & 01001100 & 10001001 & 01001100 & 10010000 & 00100000 \\
\end{array} \]
Example: Encoding of an Intel 64 instruction

\[0x66 \ 49 \ 89 \ 4C \ 90 \ 20 \ <=> \ mov \ %r9, \ 0x20(\%eax,\%edx,4)\]
Example: Encoding of an Intel 64 instruction

```
0x66 49 89 4C 90 20  <=>  mov %r9, 0x20(%eax,%edx,4)
```

Binary representation:

```
01100111 01001100 10001001 01001100 10 0100000 00100000
```
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 \(\iff\) \texttt{mov \%r9, 0x20(%eax,%edx,4)}
Example: Encoding of an Intel 64 instruction

\[ 0x66 \, 49 \, 89 \, 4C \, 90 \, 20 \leftrightarrow \text{mov} \, \%r9, \, 0x20(\%eax,\%edx,4) \]
Example: Encoding of an Intel 64 instruction

0x66 49 89 4C 90 20 <=> mov %r9, 0x20(%eax,%edx,4)
Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!
Example: Encoding of an ARM instruction

\[0x15\ 2D\ 40\ 05\ \Leftrightarrow\ \text{strne}\ r4,\ [sp,\ #-5]!\]

00010101 00101101 01000000 00000101

\text{strne}\ r4,\ [sp,\ #-5]!
Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!

00010101 00101101 01000000 00000101

strne r4,[sp,#-5]!
Example: Encoding of an ARM instruction

0x15 2D 40 05 \(\leftrightarrow\) strne r4, [sp, #-5]!

00010101 00101101 01000000 00000101

strne  r4,[sp,#-5]!
Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!

00010101 0101101 01000000 00000101
Example: Encoding of an ARM instruction

0x15 2D 40 05 \(\leftrightarrow\) strne r4, [sp, #-5]!

00010101 0101101 01000000 00000101

strne r4, [sp, #-5]!
Example: Encoding of an ARM instruction

0x15 2D 40 05 <=> strne r4, [sp, #-5]!

00010101 0101101 01000000 00000101

strne r4, [sp, #-5]!
Example: Encoding of an ARM instruction

$0x15\ 2D\ 40\ 05 \iff \text{strne r4, [sp, \#-5]}$!
Example: Encoding of an ARM instruction

\[ \text{0x15 2D 40 05} \iff \text{strne r4, [sp, #-5]}! \]
## Requirements

**Ensuring agnosticism with regard to architecture**
- Unified representation of an architecture encoding rules
- Decorrelation of decoding from post parsing actions
- Same representation to generate encoder and decoder

**Remaining close to the documentation format**
- Handling exclusions and restricted cases
- Possibility of fields with no fixed value
Using a context-free grammar formalism

**Advantages**
- Allows to decorrelate the encoding rules from the actions performed
- Decoder implemented as the corresponding parser
- Multiple possible uses for the decoder
- Encoder built from the same grammar

**Challenges**
- Grammars usually operate at the character level
- Using a bit by bit parsing would be inefficient
- Lookahead challenged by instructions of variable sizes
### Standard notions

#### Context free grammars
- Symbols associated to list of productions
- A production contains terminal and nonterminal symbols
- Terminal symbols have no production
- Semantic actions associated to productions

#### LR parsers
- Processing left to right
- Bottom-up matching
- Implemented as finite state automata
- Shift and reduction states
Our algorithm for parser generation

New principles

- Bits can have a fixed or unfixed value
- Terminals are defined as groups of bits
- A state represents the matching of bits anywhere in the production
- Transitions over terminals can include bits ahead of the parsing step
- Shift/reduce states are authorised

Parser execution

- Processing left to right
- Terminals containing less unfixed bits are tested first
Example: Context free grammar

%token <2> d
Start:
A 00
|B 01
;
A:
C
|0111
; ;
B:
0111
|d 11
;
C:
00 d
|0000
;
Example: Context free grammar

%token <2> d
Start:
A 00
|B 01
;  
A:
C
|0111
;  
B:
0111
|d 11
;  
C:
00 d
|0000
;
Example: Context free grammar

```plaintext
%token <2> d
Start:
A 00
| B 01
| ;
A:

C 00
| 0111 00
| ;
B:
0111
d 11
| ;
C:
00 d
| 0000
| ;
```
Example: Context free grammar

%token <2> d
Start:
A 00
|B 01 ;
A:
C
|0111 ;
B:
0111 |d 11 ;
C:
00 d |
|0000 ;
Example: Context free grammar

```plaintext
%token <2> d
Start:
A 00
|B 01
 ;
A:
C
|0111
|0111 00xx 00
|0000 00
|0111 00

B:
0111
|d 11
;
C:
00 d
|0000
;
```
Example: Context free grammar

%token <2> d

Start:
A 00
|B 01
;
A:
\textbf{C}
|0111
;\textcolor{blue}{00xx 00}
0000 00
0111 00
B:
\textcolor{green}{0111 01}
|d 11\textcolor{red}{xx11 01}
;
C:
\textcolor{red}{00 d}
|0000
;
Example: Context free grammar

%token <2> d
Start:
A 00
|B 01
;
A:
C
|0111
|0111
;
B:
0111
d 11
;
C:
00 d
|0000
;
Example: Parser Generation from grammar

```plaintext
%token <2> d
Start:
  A 00
  | B 01
  |
A:
  C
  | 0111
  |
B:
  0111
  | d 11
  |
C:
  00 d
  | 0000
  |
Start->. A 00
Start->. B 01
```
Example: Parser Generation from grammar

```plaintext
%token <2> d
Start:
 A 00
 | B 01 
A:
 | 0111 
B:
 | 0111 | d 11 
C:
 00 d | 0000

Start->. A 00
____00

Start->. B 01
____01
```
Example: Parser Generation from grammar

```flex
%token <2> d
Start:
 A 00
 | B 01
;
A:
 | 0111
;
B:
0111
 | d 11
;
C:
00 d
 | 0000
;
```

```
Start->. A 00
Start->. B 01

_start_
00

Start->. A 00
A->. C
A->.0111

01

Start->. B 01
```
Example: Parser Generation from grammar

```plaintext
%token <2> d
Start:
  A 00
| B 01
;
A:
  C
| 0111
;
B:
  0111
| d 11
;
C:
  00 d
| 0000

Start->. A 00
Start->. B 01
00
01

Start->. A 00
A->. C
A->.0111
C->.00xx
C->.0000
```
Example: Parser Generation from grammar

```verbatim
%token <2> d
Start:
 A 00
 | B 01
; A:
 | 0111
; B:
 | 0111
 | d 11
; C:
 | 00 d
 | 0000
; Start->. A 00
 | A->. C
 | A->.0111
 | C->.00xx
 | C->.0000
00
01
Start->. B 01
00
01
00
A->01.11
C->00.xx
C->00.00
Start->. A 00
Start->. B 01
```
Example: Parser Generation from grammar

```
%token <2> d
Start:
  A 00
  B 01
A:
  C
| 0111
B:
  0111
| d 11
C:
  00 d
| 0000

Start->. A 00
Start->. B 01

Start->. A 00
  A->. C
  A->.0111
  C->.00xx
  C->.0000

Start->. B 01

A->01.11
A->0111.

C->00.xx
C->00.00
```
Example: Parser Generation from grammar

```plaintext
%token <2> d
Start:  
  A 00
    | B 01
  ;
A:  
  | 0111
  ;
B:  
  0111
    | d 11
  ;
C:  
  00 d
    | 0000
  ;
01 00
  00
  01

Start->. A 00
  A->. C
  A->.0111
  C->.00xx
  C->.0000

Start->. B 01
  Start->. B 01

A->01.11
  11
  A->0111.

C->00.xx
  xx
  C->00xx.
```
Example: Parser Generation from grammar

```
%token <2> d
Start:
 A 00
| B 01
;  
A:
  |0111
;  
B:
  0111
|d 11
;  
C:
  00 d
|0000

Start->. A 00
A->. C
A->.0111
C->.00xx
C->.0000

Start->. B 01
B->. 0111
B->. xx11
```
Example: Parser Generation from grammar

%token <2> d
Start:
A 00
| B 01
;
A:
| 0111
;
B:
0111
| d 11
;
C:
00 d
| 0000
;

Start->. A 00
Start->. B 01

00

01

00

01

Start->. A 00
A->. C
A->. 0111
C->. 00xx
C->. 0000

B->. 0111
B->. 00xx

0111

xx11

A->0111.
C->00xx.

A->0111.
C->0000.

00

xx

A->0111.
C->0000.
Encoder generation

Building an encoder from the same grammar file

- Semantic actions are redefined as matching functions
- Input tentatively matched over all productions of nonterminals
- Shortest productions are matched first
- Nonterminals in a matching production are recursively matched
- Resulting encoder algorithm corresponds to a top-down parser
Example: Encoder algorithm

%token <2> d
Start:
A 00 #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]# ;
A:
C 0111 #[ A_ACT1($1) ]#
| 0111 #[ A_ACT2() ]# ;
B:
0111 #[ B_ACT1() ]#
| d 11 #[ B_ACT2($1) ]# ;
C:
00 d #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]# ;
Example: Encoder algorithm

%token <2> d
Start:
 A 00  # [S_ACT1($)1] #
 | B 01  # [S_ACT2($)1] # ;
 A:
 C   # [A_ACT1($)1] #
 | 0111  # [A_ACT2()] # ;
 B:
0111  # [B_ACT1()] #
 | d 11  # [B_ACT2($)1] # ;
 C:
00 d  # [C_ACT1($)1] #
 | 0000  # [C_ACT2()] # ;
Example: Encoder algorithm

```
%token <2> d
Start:
A 00 #[ S_ACT1($1) ]#
| B 01 #[ S_ACT2($1) ]#;
A:
C 0111 #[ A_ACT1($1) ]#
| 0000 0111 #[ A_ACT2() ]#;
B:
0111 #[ B_ACT1() ]#
| d 11 #[ B_ACT2($1) ]#;
C:
00 d 0000 #[ C_ACT1($1) ]#
| 0000 #[ C_ACT2() ]#;
```

```
INPUT

Match S_ACT1 (INPUT)
```
Example: Encoder algorithm

```plaintext
%token <2> d
Start:
A 00  # S_ACT1($1) ]#
| B 01 # S_ACT2($1) ]#;
A:
C  # A_ACT1($1) ]#
| 0111 # A_ACT2() ]#;
B:
0111 # B_ACT1() ]#
| d 11 # B_ACT2($1) ]#;
C:
00 d  # C_ACT1($1) ]#
| 0000 # C_ACT2() ]#;  
```

INPUT

Match S_ACT1 (INPUT)

Match S_ACT2 (INPUT)
Example: Encoder algorithm

```
%token <2> d
Start:
  A 00  #[ S_ACT1($1)]#
  B 01  #[ S_ACT2($1)]# ;
A:
  C 0111  #[ A_ACT1($1)]# ;
B:
  0111  #[ B_ACT1() ]# ;
  d 11  #[ B_ACT2($1)]# ;
C:
  00 d  #[ C_ACT1($1)]#
  0000  #[ C_ACT2() ]# ;
```

Diagram:
- INPUT
- Match S_ACT1 (INPUT)
- Match S_ACT2 (INPUT)
Example: Encoder algorithm

```
%token <2> d
Start:
A 00  #[ S_ACT1($1)]#
  | B 01  #[ S_ACT2($1)]# ;
A:
  C 0111  #[ A_ACT1($1)]#
  | 0111  #[ A_ACT2()]# ;
B:
  0111  #[ B_ACT1()]# ;
  | d 11  #[ B_ACT2($1)]# ;
C:
  00 d  #[ C_ACT1($1)]#
  | 0000  #[ C_ACT2()]# ;
```

Example: Encoder algorithm

%token <2> d
Start:
A 00  ![S_ACT1($1)]#
| B 01 ![S_ACT2($1)]# ;
A:
C  ![A_ACT1($1)]#
 | 0111 ![A_ACT2()] # ;
B:
0111 ![B_ACT1()] #
 | d 11 ![B_ACT2($1)]# ;
C:
00 d  ![C_ACT1($1)]#
 | 0000 ![C_ACT2()] # ;
Example: Encoder algorithm

```
%token <2> d
Start:
A 00 # S_ACT1($1) #
| B 01 # S_ACT2($1) # ;
A:
C | 0111 # A_ACT1($1) #
| 0111 # A_ACT2() #;
B:
0111 # B_ACT1() #
| d 11 # B_ACT2($1) # ;
C:
00 d # C_ACT1($1) #
| 0000 # C_ACT2() # ;
```

---

**INPUT**

- **Match S_ACT1 (INPUT)**
- **Match S_ACT2 (INPUT)**

---

**INPUT for A**

- **Match A_ACT1 (INPUT for A)**

**INPUT for C**

- **Match C_ACT1 (INPUT for C)**

**CODING: 000000**

---

**Match A_ACT2 (INPUT for A)**

**Match C_ACT2 (INPUT for C)**
Example: Encoder algorithm

```plaintext
%token <2> d

Start:
A 00  #[ S_ACT1($1) ]#
| B 01  #[ S_ACT2($1) ]#;
A:
C  #[ A_ACT1($1) ]#
  | 0111  #[ A_ACT2() ]#;
B:
0111  #[ B_ACT1() ]#
  | d 11  #[ B_ACT2($1) ]#;
C:
00 d  #[ C_ACT1($1) ]#
  | 0000  #[ C_ACT2() ]#;
```
Example: Encoder algorithm

%token <2> d
Start:
A 00  #S_ACT1($1) #
| B 01 #S_ACT2($1) # ;
A:
C  #A_ACT1($1) #
| 0111 #A_ACT2() # ;
B:
0111 #B_ACT1() #
| d 11 #B_ACT2($1) # ;
C:
00 d  #C_ACT1($1) #
| 0000 #C_ACT2() # ;
Example: Encoder algorithm

```plaintext
%token <2> d
Start:
  A 00 # S_ACT1(1) #
  | B 01 # S_ACT2(1) #
  | A 011 # A_ACT1(1) #
  | 0111 # A_ACT2() #
B:
  0111 # B_ACT1() #
  | d 11 # B_ACT2(1) #
C:
  00 d # C_ACT1(1) #
  | 0000 # C_ACT2() #
```

```
```

```
INPUT

Match S_ACT1 (INPUT)

Match S_ACT2 (INPUT)

ENCODING ERROR

```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Example: Encoder algorithm

%token <2> d
Start:
A 00 #S_ACT1($1)#
| B 01 #S_ACT2($1)#
| C 0111 #A_ACT1($1)#
| 0111 #A_ACT2()#
| d 11 #B_ACT2($1)#
| 00 d #C_ACT1($1)#
| 0000 #C_ACT2()#

INPUT for A

Match A_ACT1 (INPUT for A)

Match A_ACT2 (INPUT for A)

CODING: 011100

INPUT for C

Match C_ACT1 (INPUT for C)

Match C_ACT2 (INPUT for C)

CODING: 000000

CODING: 00??00

INPUT for B

Match B_ACT1 (INPUT for B)
Example: Encoder algorithm

```plaintext
%token <2> d
Start:
A 00  #S_ACT1($1)#
| B 01  #S_ACT2($1)#
| ;
A:
| 0111  #A_ACT1($1)#
| d 11  #A_ACT2()#
| ;
B:
| 0111  #B_ACT1()#
| d 11  #B_ACT2($1)#
| ;
C:
| 00 d  #C_ACT1($1)#
| 0000  #C_ACT2()#
|
```

![Diagram of the encoder algorithm]

- **INPUT for A**
  - Match A_ACT1
    - (INPUT for A)
    - CODING: 011100
  - Match A_ACT2
    - (INPUT for A)
- **INPUT for C**
  - Match C_ACT1
    - (INPUT for C)
    - CODING: 000000
  - Match C_ACT2
    - (INPUT for C)
    - CODING: 000001
Example: Encoder algorithm

%token <2> d
Start:
A 00  #S_ACT1($1)#
| B 01 #S_ACT2($1)#;
A:
| 0111 #[A_ACT1($1)]#
| 0111 #[A_ACT2() ]#;
B:
| 0111 #[B_ACT1() ]#
| d 11 #[B_ACT2($1)]#;
C:
| 00d  #C_ACT1($1)#
| 0000 #[C_ACT2() ]# ;
Validation

MINJAG
- Uses a context-free grammar describing the architecture
- Grammar generated from architecture documentation through simple transformations
- Generates the code for decoder and encoder from the same grammar
- Functional tool used in a production context
- Tested over Intel 64, Intel Xeon Phi coprocessor and ARM
Characteristics of implemented architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Intel 64</th>
<th>Intel Xeon Phi</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines in instruction list</td>
<td>2,398</td>
<td>1,194</td>
<td>1,512</td>
</tr>
<tr>
<td>Lines in grammar</td>
<td>6,082</td>
<td>3,082</td>
<td>1,491</td>
</tr>
<tr>
<td>Reduction states</td>
<td>5,950</td>
<td>2,406</td>
<td>1,625</td>
</tr>
<tr>
<td>Shift states</td>
<td>4,019</td>
<td>1,468</td>
<td>2,916</td>
</tr>
<tr>
<td>Shift/reduce states</td>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>Total states</td>
<td>9,971</td>
<td>3,876</td>
<td>4,547</td>
</tr>
</tbody>
</table>
1 Introduction

2 Multi architecture support

3 Disassembly of binary files

4 Binary rewriting

5 Conclusion
Challenges of disassembly

Binary code is not intended to be read
- No constraints on the code as long as the program can be executed
- No separation between instructions
- Instructions may be of varying sizes

Specific examples
- Interleaved foreign bytes
- Overlapping instructions
- Obfuscated code or binary format
- Self rewriting code
Example: Interleaved foreign bytes

<table>
<thead>
<tr>
<th>Binary code</th>
<th>Corresponding assembly instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>67 4C 89 4C 90 20</td>
<td>mov %r9,0x20(%eax,%edx,4)</td>
</tr>
<tr>
<td>EB 04</td>
<td>jmp &lt;+4 bytes&gt;</td>
</tr>
<tr>
<td>80</td>
<td><strong>Alignment byte, never executed</strong></td>
</tr>
<tr>
<td>4C 89 C8</td>
<td>mov %r9,%rax</td>
</tr>
<tr>
<td>F2 0F 10 EE</td>
<td>movsd %xmm6, %xmm5</td>
</tr>
<tr>
<td>F2 0F 10 F4</td>
<td>movsd %xmm4, %xmm6</td>
</tr>
</tbody>
</table>

Disassembly:

- Correct disassembly up to that point:
  - 67 4C 89 4C 90 20
  - EB 04
  - 80 4C 89 C8 F2
  - 0F 10 EE
  - F2 0F 10 F4
  - mov %r9,0x20(%eax,%edx,4)
  - jmp <+4 bytes>
  - or $0xf2,-0x38(%rcx,%rcx,4)
  - movups %xmm6, %xmm5
  - movsd %xmm4, %xmm6

- Mistaking the alignment byte for the beginning of the next instruction:
  - 67 4C 89 4C 90 20
  - EB 04
  - 80 4C 89 C8 F2
  - 0F 10 EE
  - F2 0F 10 F4

- Realignement of the parser on a valid boundary:
  - Instructions correctly disassembled
  - Errorneous instructions
  - Instructions correctly disassembled
## Disassembly algorithms

### Linear sweep
- Decoding one instruction after another
- Errors when encountering interleaved foreign bytes
- Vulnerability to obfuscation methods
- Faster disassembly

### Recursive traversal
- Decoding following the actual execution of the program
- Resists to some obfuscation techniques
- Finding the destination of a branch can be difficult
- Slower disassembly
### Linear Sweep vs Recursive Traversal

<table>
<thead>
<tr>
<th>Linear sweep</th>
<th>Recursive traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>01100111010011001000100101001100</td>
<td>01100111010011001000100101001100</td>
</tr>
<tr>
<td>100100000010000011101011100000011</td>
<td>100100000010000011101011100000011</td>
</tr>
<tr>
<td>0100100110001001111001110 10010000</td>
<td>0100100110001001111001110 10010000</td>
</tr>
<tr>
<td>11110010000011110001000011011100</td>
<td>11110010000011110001000011011100</td>
</tr>
<tr>
<td>11110010000011110001000011011100</td>
<td>11110010000011110001000011011100</td>
</tr>
</tbody>
</table>
Linear Sweep vs Recursive Traversal

**Linear sweep**

01100111010011001000100101001100
1001000000100000 1110101110000011
010010011100110110110110 10010000
11110010000011110001000011101110

00: mov %r9,0x20(%eax,%edx,4)

**Recursive traversal**

01100111010011001000100101001100
1001000000100000 1110101110000011
010010011100110110110110 10010000
11110010000011110001000011101110

00: mov %r9,0x20(%eax,%edx,4)
Linear Sweep vs Recursive Traversal

Linear sweep

```
01100111010011001000100101001100 00: mov %r9,0x20(%eax,%edx,4)
1001000000100000 1110101100000011 06: jmp <0C>  +#4 bytes
01001001110011011000111011001110
11110010000001110001000011101110
```

Recursive traversal

```
01100111010011001000100101001100 00: mov %r9,0x20(%eax,%edx,4)
1001000000100000 1110101100000011 06: jmp <0C>  +#4 bytes
01001001110011011000111011001110
11110010000001110001000011101110
```
Linear Sweep vs Recursive Traversal

**Linear sweep**

```
01100111010011001000100101001100
1001000000100000 1110101100000011
010010011000100111001110
10010000000111000100001101110
```

00: mov %r9,0x20(%eax,%edx,4)
06: jmp <0C> #+#4 bytes
08: mov %rcx,%r14

**Recursive traversal**

```
01100111010011001000100101001100
1001000000100000 1110101100000011
010010011000100111001110
11100100000111000100001101110
```

00: mov %r9,0x20(%eax,%edx,4)
06: jmp <0C> #+#4 bytes
0C: movsd %xmm6,%xmm5
Our constraints

**Disassembler intended to be used by analysis tools**

- Retrieve all possible available information from the file
- Architecture independent output format
- Possibility to add customisable additional information
- Acceptable performance in terms of speed and accuracy
Our disassembling algorithm

<table>
<thead>
<tr>
<th>General execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Linear sweep parsing</td>
</tr>
<tr>
<td>- Extraction of executable code from binary format</td>
</tr>
<tr>
<td>- Retrieval of labels and debug information if present</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Additional processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Resolving destination of direct branches</td>
</tr>
<tr>
<td>- Associating labels and debug information to instructions</td>
</tr>
<tr>
<td>- Post parsing actions to fill additional information</td>
</tr>
<tr>
<td>- Detection of unreachable instructions</td>
</tr>
<tr>
<td>- Identification of dubious disassembled data</td>
</tr>
</tbody>
</table>
Implementation: the MADRAS disassembler

Multi Architecture Disassembler, Rewriter and ASsembler

- Relies on MINJAG for source code of decoder
- Processes binaries using the ELF format used by Unix and Linux
- Disassembler available for Intel 64, Xeon Phi coprocessor and ARM
- Base component of the MAQAO framework
Performance tests

Protocol

- Comparison between MADRAS and hard coded disassemblers
- Disassembling SPEC benchmarks and test files
  - Size of executable code varying between 1 and 23 MBytes
  - Executables compiled for Intel 64 and Xeon Phi coprocessor
- Speed measured as disassembled instructions per second
Disassembler performance on Intel 64 files

![Graph showing disassembler performance on Intel 64 files.](image-url)
Disassembler performance on Xeon Phi files
Parallel disassembler performance

![Graphs showing performance comparison between Intel 64 files and Xeon Phi files.]

- **Intel 64 files**
  - madras 1 thread
  - madras 2 threads
  - madras 4 threads
  - madras 8 threads

- **Xeon Phi files**
  - madras 1 thread
  - madras 2 threads
  - madras 4 threads
  - madras 8 threads

Instructions per second are measured for various benchmarks, including Small, fma3d, calculix, gcc, dealii, Xalan, tonto, wrf, gamess, Large 1, Large 2, equake, art, ammp, swim, wupwise, mgrid, applu, apsi, galgel, and fma3d.
Disassembler accuracy

![Disassembler accuracy chart](chart.png)
1. Introduction
2. Multi architecture support
3. Disassembly of binary files
4. Binary rewriting
5. Conclusion
## Instrumentation

<table>
<thead>
<tr>
<th>Retrieving information during execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Monitoring memory usage</td>
</tr>
<tr>
<td>• Value profiling</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic: Performed during execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Monitoring code execution using a supervising thread</td>
</tr>
<tr>
<td>• Invoking functions under specified conditions</td>
</tr>
<tr>
<td>• Modifying the image loaded in memory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Static: Modifying the executable file</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Probe insertion</td>
</tr>
<tr>
<td>• Instructions modification</td>
</tr>
</tbody>
</table>
Binary rewriting

Static instrumentation
- No recompilation needed
- No overhead from instrumentation process
- No additional requirements for execution

Binary rewriting allows other modifications to the program
- Deleting or adding instructions to test their overall impact
- Modifying variables defined in the file
Challenges of binary rewriting

- Patched file must remain valid
  - Preservation of the structure of the binary file
  - Preservation of the control flow
  - Preservation of data environment

- Executables are not intended to be modified
  - All references are fixed
  - No relocation tables
  - Addresses can appear as immediate operands
Example of patching pitfalls

00: 48 8B 04 25 0E 00 00 00  mov $0x14, %rax
08: FF E0  jmp *%rax
0A: 83 45 F8 01  add $1, -8(%rbp)
0E: 83 7D F8 01  cmp $1, -8(%rbp)
12: 7E F6  jle 0A
14: B8 00 00 00 00  mov $0, %eax
Example of patching pitfalls

\[ E8 \text{ xx xx xx xx} \text{ callq } <\text{myfunc}> \]

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: 48 8B 04 25 0E 00 00 00</td>
<td>mov $0x14, %rax</td>
<td>move register %rax to memory location $0x14</td>
</tr>
<tr>
<td>08: FF E0</td>
<td>jmp *%rax</td>
<td>jump to memory location pointed by %rax</td>
</tr>
<tr>
<td>0A: 83 45 F8 01</td>
<td>add $1, -8(%rbp)</td>
<td>add memory location -8 from %rbp to memory location $1</td>
</tr>
<tr>
<td>0E: 83 7D F8 01</td>
<td>cmp $1, -8(%rbp)</td>
<td>compare memory location -8 from %rbp to memory location $1</td>
</tr>
<tr>
<td>12: 7E F6</td>
<td>jle 0A</td>
<td>jump if less than or equal to 0A</td>
</tr>
<tr>
<td>14: B8 00 00 00 00</td>
<td>mov $0, %eax</td>
<td>move memory location $0 to register %eax</td>
</tr>
</tbody>
</table>
Example of patching pitfalls

```
00: 48 8B 04 25 0E 00 00 00  mov $0x14, %rax

08: FF E0  jmp *%rax

0A: 83 45 F8 01   add $1, -8(%rbp)

0E: E8 xx xx xx xx  callq <myfunc>

13: 83 7D F8 01   cmpl $1, -8(%rbp)

17: 7E F6  jle 0A

19: B8 00 00 00 00  mov $0, %eax
```
Binary rewriting algorithm

Block relocation

- The code to be modified is moved in a new section in the executable
- Code moved at the basic block level
- Use of trampolines if the patching site is too small
Code relocation

Original code

```
...
...
...
...
...
...
...
...
...

```


Code relocation

Original code

Modification site
Code relocation

Original code

Basic block surrounding the site
Code relocation

Original code

Branch instruction
Code relocation

Original code

Added code section

Relocated block
Code relocation

Original code

Added code section

Modifications
Code relocation

Original code

Added code section

Return branch
Trampolines

Original code

Modification site
Trampolines

Original code

Basic block surrounding the site
Trampolines

Original code

Trampoline block
Trampolines

Original code

Added code section

Moved trampoline block
Trampolines

- Original code
- Added code section
- Trampoline branch
Trampolines

Original code

Added code section

Branch
Trampolines

Original code

Added code section

Moved block
Trampolines

Original code

Added code section

Modifications
Trampolines

Original code

Added code section

Return branch
Implementation: the MADRAS patcher

Multi Architecture Disassembler, **Rewriter and ASsembler**
- Relies on MINJAG for source code of assembler
- Processes binaries under the ELF format used by Unix and Linux
- Available for Intel 64 and Xeon Phi coprocessor

A production tool
- C API
- Back end of the MAQAO Instrumentation Language (MIL)
- Used by the DECAN module
Patcher features

<table>
<thead>
<tr>
<th>Code insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion of calls to functions from external or static libraries</td>
</tr>
<tr>
<td>Insertion of lists of assembly instructions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Possibility to set conditions on the execution of an inserted code</td>
</tr>
<tr>
<td>Possibility to specify code to execute if such a condition is not met</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modification or deletion of instructions</td>
</tr>
<tr>
<td>Insertion of global variables usable by inserted code</td>
</tr>
</tbody>
</table>
Example: Using MADRAS API to insert a function call

```c
void insert(char* in, char* lib, char* fct, uint addr, char* out) {
    // Disassemble the file and inits the modifications
    elfdis_t* madras = madras_disass_file(in);
    madras_modifs_init(madras, STACK_SHIFT, 512);
    // Adds a function call at the given address
    insert_t* ifct = madras_fctcall_new(madras, fct, lib, addr, 0);
    // Adds the given address as an immediate parameter
    madras_fctcall_addparam_imm(madras, ifct, addr, 0);
    // Commit changes
    madras_modifs_commit(madras, out);
    // Terminates the madras structure
    madras_terminate(madras);
}
```
Interface with the MAQAO Instrumentation Language

![Diagram showing the process flow from Instrumentation File to Instrumented Binary(ies)]
Performance of code patched by MIL

![Graph showing performance times for different patched files.

- **Green bars** represent MIL.
- **Blue bars** represent DynInst.
- **Red bars** represent PEBIL.

**Axes:**
- **Y-axis:** Patched Execution Time (s)
## Contributions

**Generic representation of binary encoding rules**
- Unified format
- Use of the same grammar for encoder and decoder generation
- Validated for the Intel and ARM architectures
- Implemented as the functional tool MINJAG

**Disassembly**
- Easier updates of architecture specific code
- Performance comparable to existing hard coded tools
- Customisable output
Contributions

Patching
- Fine granularity offering wide range of options
- Patched code has similar or better performance than existing tools

MADRAS
- Functional tool
- Standalone implementation of the whole disassembly and instrumentation chain
- Handling of multiple architectures from a single executable
- Integral component of the MAQAO framework
- Used by the DECAN module
## Future work

### General
- Implement additional architectures
- Support additional binary file formats

### Generic encoder and decoder
- Generic meta language for representing instruction lists
- Extensions allowing to specialise generated parser
Future work

Disassembler
- Improve accuracy through use of recursive traversal
- Detection of switch tables
- Improve speed
- Parallel disassembly
- Application to domains outside performance analysis

Patcher
- Improve safety of patching
- Update of indirect branch destinations
Thank you for your attention!
Additional slides
Example: Encoding of an ARM instruction

\[ \text{0x15 2D 40 04} \iff \text{pushne \{r4\}} \]

\[
\begin{array}{cccccccc}
00010101 & 00101101 & 01000000 & 00000100 \\
\end{array}
\]
Example of grammar for binary definition

```plaintext

%token <3,b> reg

%%

Start: template ;
template: Legacy3 Insn #[ FULLINSN_L3PREFIX($1,$2) ]# |
        Insn #[ FULLINSN($1) ]# ;
MemModRM: 00 reg RMSIB_00 #[ OPRS_REG_MEM($1,$2) ]# |
          01 reg RMSIB_01 #[ OPRS_REG_MEM($1,$2) ]# |
          10 reg RMSIB_10 #[ OPRS_REG_MEM($1,$2) ]# ;
RegModRM: 11 reg RMSIB_11 #[ OPRS_REG_REG($1,$2) ]# ;
Insn: 00010000 RegModRM #[ INSN(ADC, 
   REG(GEN8b,R,$1),REG(GEN8b,RW,$1)) ]# |
       REX 00010000 MemModRM #[ INSN(ADC, 
   REG(GEN8b,R,$1,$2),MEM(MEM8b,RW,$1,$2)) ]# ;
```
Overlapping instructions

Destination of the branch instruction

F3 AB
48 FF C1
48 83 F9 7F
75 F6

rep stos
inc %rcx
cmp $127,%rcx
jne <-10 bytes>

The first iteration of the loop will execute instruction rep stos. Later iterations will skip the F3 (rep) prefix and execute only the stos instruction.
Obfuscated code

```
83 C0 42
EB 02
89 81 49 83 E9 06
```

- mov %eax, 0x6e98349(%rcx)
- Instruction incorrectly identified by a linear sweep disassembler
- add $0x42, %eax
- jmp <+2 bytes>
- Destination of the branch instruction
- sub $0x6, %r9
- Instruction actually executed by the processor
Performance tests

Disassemblers
- objdump
- XED
- udis86
- distorm
- ndisasm

Disassembly modes
- Print only mode for comparison against objdump and XED
- Without parsing of the binary file against udis86 and distorm
## Intel 64 files used for the disassembler performance tests

<table>
<thead>
<tr>
<th>File</th>
<th>File size (MByte)</th>
<th>Code size (MByte)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small</td>
<td>0.96</td>
<td>0.96</td>
<td>Test file</td>
</tr>
<tr>
<td>fma3d</td>
<td>3.78</td>
<td>1.75</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>calculix</td>
<td>5</td>
<td>2.31</td>
<td>SPEC2006</td>
</tr>
<tr>
<td>gcc</td>
<td>9.02</td>
<td>2.56</td>
<td>SPEC2006</td>
</tr>
<tr>
<td>dealII</td>
<td>60.94</td>
<td>2.83</td>
<td>SPEC2006</td>
</tr>
<tr>
<td>Xalan</td>
<td>130.64</td>
<td>3.46</td>
<td>SPEC2006</td>
</tr>
<tr>
<td>tonto</td>
<td>33.27</td>
<td>5.81</td>
<td>SPEC2006</td>
</tr>
<tr>
<td>wrf</td>
<td>19.52</td>
<td>6.83</td>
<td>SPEC2006</td>
</tr>
<tr>
<td>gamess</td>
<td>18.2</td>
<td>10.55</td>
<td>SPEC2006</td>
</tr>
<tr>
<td>Large 1</td>
<td>11.95</td>
<td>11.94</td>
<td>Test file</td>
</tr>
<tr>
<td>Large 2</td>
<td>23.22</td>
<td>23.22</td>
<td>Test file</td>
</tr>
</tbody>
</table>
### Xeon Phi files used for the disassembler performance tests

<table>
<thead>
<tr>
<th>File</th>
<th>File size (Mb)</th>
<th>Code size (Mb)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equake</td>
<td>0.12</td>
<td>0.05</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>art</td>
<td>0.21</td>
<td>0.12</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>ammp</td>
<td>0.84</td>
<td>0.44</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>swim</td>
<td>0.96</td>
<td>0.57</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>wupwise</td>
<td>0.96</td>
<td>0.66</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>mgrid</td>
<td>0.95</td>
<td>0.68</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>applu</td>
<td>1.03</td>
<td>0.71</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>apsi</td>
<td>2.61</td>
<td>1.72</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>galgel</td>
<td>2.84</td>
<td>2.08</td>
<td>SPEC2001</td>
</tr>
<tr>
<td>fma3d</td>
<td>4.62</td>
<td>2.35</td>
<td>SPEC2001</td>
</tr>
</tbody>
</table>
Disassembler performance on Intel 64 files
Parallel disassembler performance

- Parallel disassembler performance graphs showing instructions per second for different benchmarks and thread counts.

- Lower graph: Instructions per second for benchmarks such as equake, art, ammp, swim, wupwise, mgrid, applu, apsi, galgel, and fma3d.

- Upper graph: Instructions per second for benchmarks like small, fma3d, calculix, gcc, dealii, Xalan, tonto, wrf, gamess, Large 1, and Large 2, with thread counts of 2, 4, and 8.
Performance of patched code

![Graph showing execution time comparison between original and instrumented code for various benchmarks. The x-axis represents the benchmarks: ammp, applu, apsi, art, equake, fma3d, galgel, mgrid, swim, wupwise. The y-axis represents execution time in seconds, ranging from 0 to 400. The graph compares the execution times of original and instrumented codes.]
Performance of instrumentation

![Graph showing the relationship between the number of function insertions and instrumentation time.](image)
Performance of patched code

![Graph showing Performance of patched code]

- MIL
- DynInst
- PEBIL

Patched Execution Time (s)

- ua.A
- sp.A
- mg.A
- lu.A
- is.A
- ft.A
- ep.A
- dc.A
- cg.A
- bt.A

69 / 55
MADRAS overall architecture

- **Binary file**
  - 0111000
  - 1100111
  - 0010010
  - 0000001
  - 0000111

- **Disassembler**
  - Instruction: `mov`
  - Parameter 1: Register `rax` size 64b
  - Parameter 2: Memory address
  - Address: 0x400400
  - Instruction: `add`

- **Patcher**
  - Instruction: `add`
  - Possible operands: reg, mem

- **Instrumentation results**

- **Architecture description**
  - Instruction `mov`: Possible operands: reg, mem
  - Coding: 0x60 0x7
  - Instruction `add`: Possible operands: reg, mem

- **MINJAG**
  - Generation of architecture specific source code

- **MAQAO**
  - Other analysis tools
MAQAO Framework

Binary Manipulation Layer (MADRAS)
- Disassembler Generator
- Disassemble
- Re-assemble
- Patch/Rewrite

Structuration and Abstraction layer
- Functions
- Loops
- Instructions
- Demangling
- Basic blocks
- Debug symbols
- Other code abstraction algorithms

MAQAO Lua Plugins
- API bindings to Abstraction And Binary layers
- DECAN
- CQA
- MIL
- Profiler
- MTL