Project Proposal: Decompiler Understandability Study

Jeremy Lacomis

Reverse Engineering

26 Feb 2013 | 14:00 GMT

The Real Story of Stuxnet

How Kaspersky Lab tracked down the malware that stymied Iran's nuclear-fuel enrichment program

By **David Kushner**

Computer cables snake across the floor. Cryptic flowcharts are scrawled across various whiteboards adorning the walls. A life-size Batman doll stands in the hall. This office might seem no different than any other geeky workplace, but in fact it's the front line of a war—a cyberwar, where most battles play out not in remote jungles or deserts but in

2014 IEEE International Conference on Software Maintenance and Evolution

Reverse Engineering PL/SQL Legacy Code: An Experience Report

Martin Habringer
voestalpine Stahl GmbH
4020 Linz, Austria
martin.habringer@voestalpine.com

Michael Moser and Josef Pichler
Software Analytics and Evolution
Software Competence Center Hagenberg GmbH
4232 Hagenberg, Austria
michael.moser@scch.at, josef.pichler@scch.at

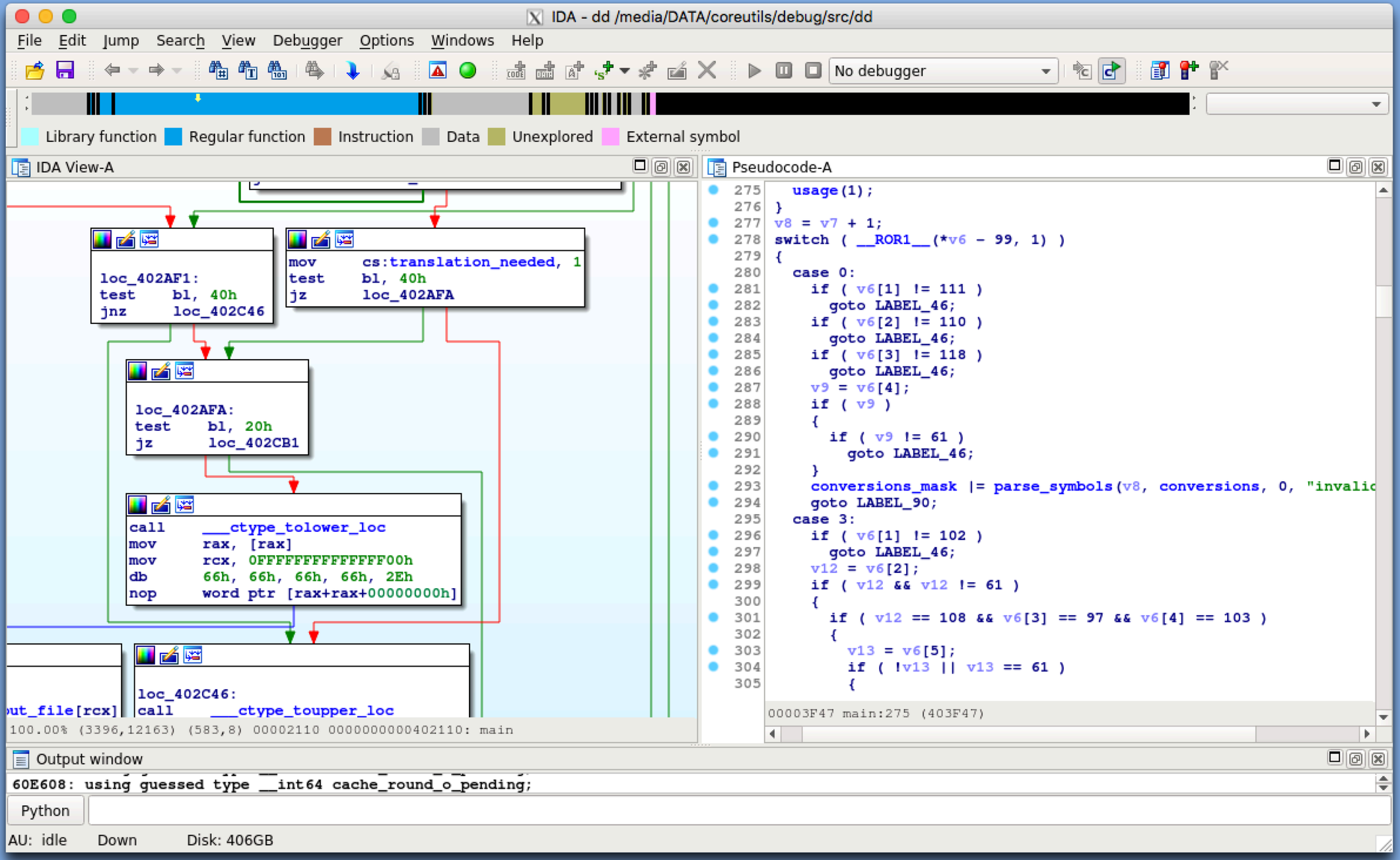
***Abstract*—The reengineering of legacy code is a tedious endeavor. Automatic transformation of legacy code from an old technology to a new one preserves potential problems in legacy code with respect to obsolete, changed, and new business cases. On the other hand, manual analysis of legacy code without assistance of original developers is time consuming and error-prone. For the purpose of reengineering PL/SQL legacy code in the steel making domain, we developed tool support for the reverse engineering of PL/SQL code into a more abstract and comprehensive representation. This representation then serves as input for stakeholders to manually analyze legacy code, to identify obsolete and missing business cases, and, finally, to support the re-implementation of a new system. In this paper we briefly introduce the tool and present results of reverse engineering PL/SQL legacy code in the steel making domain. We show how stakeholders are supported in analyzing legacy code by means of general-purpose analysis techniques combined with domain-specific representations and conclude with some of the lessons learned.**

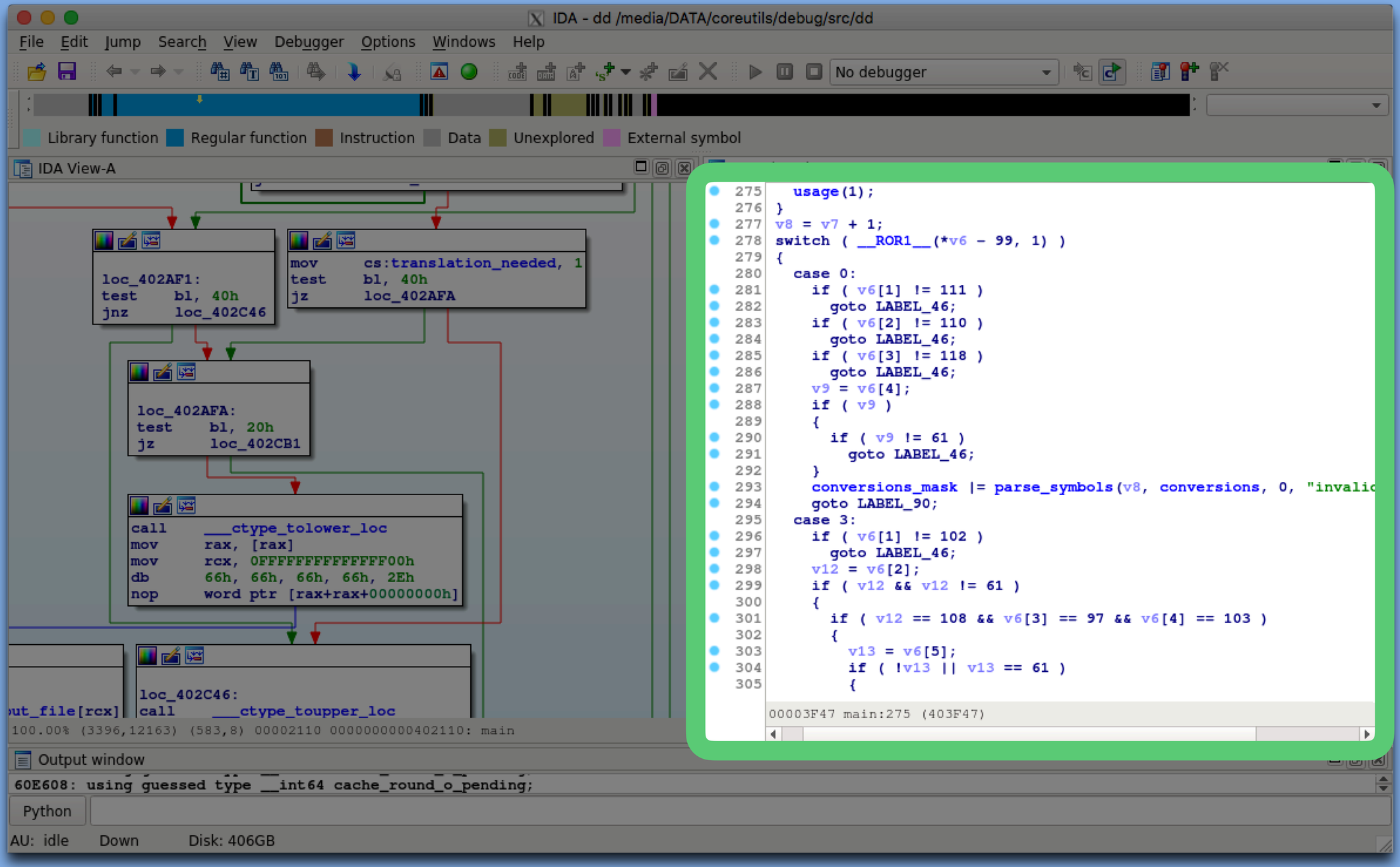
***Keywords*—reverse engineering; program comprehension; source code analysis**

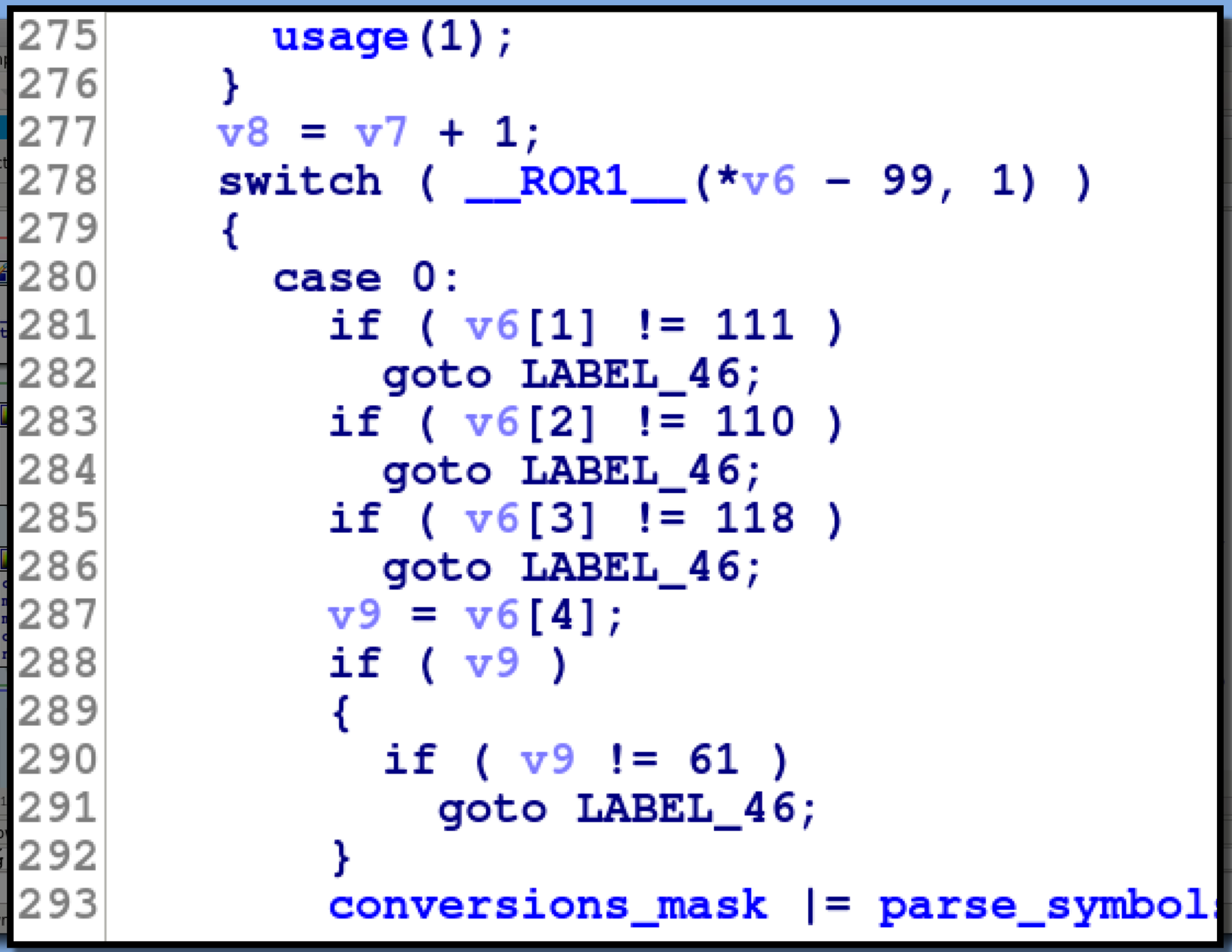
- Changes in business cases over the last years were not reflected in verification logic of the legacy code.
- For a new production plant, additional requirements must be incorporated.
- The maintenance of the legacy programs was complicated by the retirement of original developers.
- Legacy code is not extensible in a safe and reliable way.
- Stakeholders estimated high effort for manual analysis of the legacy code.

The goal for the reverse engineering tool was to support stakeholders to comprehend the verification logic implemented in the legacy programs. Whereas, comprehension requires that stakeholders can (1) *identify* the business cases currently checked by the software as well as that stakeholders are able to (2) *extend* the verification logic with respect to new requirements.

The contributions of this paper are:


IDA - dd /media/DATA/coreutils/debug/src/dd
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function
Regular function
Instruction
Data
Unexplored
External symbol
IDA View-A
loc_402AF1:
test bl, 40h
jnz loc_402C46
mov cs:translation_needed, 1
test bl, 40h
jz loc_402AFA
loc_402AFA:
test bl, 20h
jz loc_402CB1
call ___ctype_tolower_loc
mov rax, [rax]
mov rcx, 0FFFFFFFFFFFFFF00h
db 66h, 66h, 66h, 66h, 2Eh
nop word ptr [rax+rax+00000000h]
loc_402C46:
call ___ctype_toupper_loc
100.00% (3396,12163) (583,8) 00002110 0000000000402110: main
Pseudocode-A
275 usage(1);
276 }
277 v8 = v7 + 1;
278 switch (__ROR1__(*v6 - 99, 1))
279 {
280 case 0:
281 if (v6[1] != 111)
282 goto LABEL_46;
283 if (v6[2] != 110)
284 goto LABEL_46;
285 if (v6[3] != 118)
286 goto LABEL_46;
287 v9 = v6[4];
288 if (v9)
289 {
290 if (v9 != 61)
291 goto LABEL_46;
292 }
293 conversions_mask |= parse_symbols(v8, conversions, 0, "invalic
294 goto LABEL_90;
295 case 3:
296 if (v6[1] != 102)
297 goto LABEL_46;
298 v12 = v6[2];
299 if (v12 && v12 != 61)
300 {
301 if (v12 == 108 && v6[3] == 97 && v6[4] == 103)
302 {
303 v13 = v6[5];
304 if (!v13 || v13 == 61)
305 {
00003F47 main:275 (403F47)
Output window
60E608: using guessed type __int64 cache_round_o_pending;
Python
AU: idle Down Disk: 406GB


IDA - dd /media/DATA/coreutils/debug/src/dd
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Regular function Instruction Data Unexplored External symbol
IDA View-A
loc_402AF1:
test bl, 40h
jnz loc_402C46
mov cs:translation_needed, 1
test bl, 40h
jz loc_402AFA
loc_402AFA:
test bl, 20h
jz loc_402CB1
call ___ctype_tolower_loc
mov rax, [rax]
mov rcx, 0FFFFFFFFFFFFFF00h
db 66h, 66h, 66h, 66h, 2Eh
nop word ptr [rax+rax+00000000h]
loc_402C46:
call ___ctype_toupper_loc
100.00% (3396,12163) (583,8) 00002110 0000000000402110: main
Output window
60E608: using guessed type __int64 cache_round_o_pending;
Python
AU: idle Down Disk: 406GB
275 usage(1);
276 }
277 v8 = v7 + 1;
278 switch (__ROR1__(*v6 - 99, 1))
279 {
280 case 0:
281 if (v6[1] != 111)
282 goto LABEL_46;
283 if (v6[2] != 110)
284 goto LABEL_46;
285 if (v6[3] != 118)
286 goto LABEL_46;
287 v9 = v6[4];
288 if (v9)
289 {
290 if (v9 != 61)
291 goto LABEL_46;
292 }
293 conversions_mask |= parse_symbols(v8, conversions, 0, "invalic
294 goto LABEL_90;
295 case 3:
296 if (v6[1] != 102)
297 goto LABEL_46;
298 v12 = v6[2];
299 if (v12 && v12 != 61)
300 {
301 if (v12 == 108 && v6[3] == 97 && v6[4] == 103)
302 {
303 v13 = v6[5];
304 if (!v13 || v13 == 61)
305 {
00003F47 main:275 (403F47)


usage(1);
}
v8 = v7 + 1;
switch (__ROR1__(*v6 - 99, 1))
{
case 0:
if (v6[1] != 111)
goto LABEL_46;
if (v6[2] != 110)
goto LABEL_46;
if (v6[3] != 118)
goto LABEL_46;
v9 = v6[4];
if (v9)
{
if (v9 != 61)
goto LABEL_46;
}
conversions_mask |= parse_symbol

The problem:

Decompilers are good at rebuilding structure, but bad at rebuilding things that can't be computed (e.g., comments, variable names, custom types)

Compilation Loses Information

Comments:

```
/* This is the functionality
 * you're looking for! */
```

Loop Constructs:

```
while (x < 100) {...}

for (i = 0; i < 100; ++i) {...}
```

Variable Names:

```
int width, length;
double volume;
char *user_id;
```

User-Defined Types:

```
typedef struct {
  int x;
  int y;
} point;
```

Compilation Loses Information

Comments:

```
/* This is the functionality
 * you're looking for! */
```

Loop Constructs:

```
while (x < 100) {...}

for (i = 0; i < 100; ++i) {...}
```

Variable Names:

```
int width, length;
double volume;
char *user_id;
```

User-Defined Types:

```
typedef struct {
  int x;
  int y;
} point;
```

```c
typedef struct {
  int x;
  int y;
} point;

double func(point *p1, point *p2) {
  double xdist, ydist;

  xdist = pow((p1->x - p2->x), 2);
  ydist = pow((p1->y - p2->y), 2);

  return sqrt(xdist + ydist);
}
```

```c
double func(int *a1, int *a2) {
  double v1, v2;

  v1 = pow((*a1 - *a2), 2);
  v2 = pow((a1[1] - a2[1]), 2);

  return sqrt(v1 + v2);
}
```

Theory:

Variable names and types provide useful information about their purpose, making code easier to understand.

Research Questions:

Do variable names and types make code easier to understand?

Does making code easier to understand help reverse engineers be more effective?

Proposed Study Design

Provide reverse engineers at the Software Engineering Institute with implementations of our tools and conduct semi-structured interviews about their experiences using them.