

Embedded Testing 2017

Risiko Binärdatei?

Einsatz von Werkzeugen zur statischen Analyse
nicht nur für den Quellcode



Royd Lüdtkke
Verifysoft Technology GmbH
luedtke@verifysoft.com
+49 781 127 8118-8



Agenda

- Statische Codeanalyse vs. dynamische Analyse
- Statische Analyse – Quellcode- oder Binärdatei?
- Anteil an extern entwickeltem Code
- Einbinden von „fremden“ Binärdateien
- Assembler-Komponenten
- Statische Binäranalyse – Mustererkennung
- “Buffer Overrun” auf dem Stack
- “Buffer Overrun” auf dem Heap
- Fazit





Dynamische Analyse

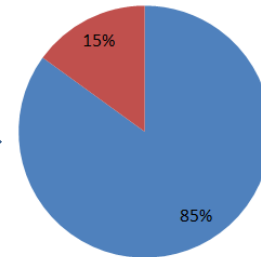
Testfälle

Test #	Funktion	Eingangsparameter	Erwartetes Ergebnis
1	init()	0	5
2	init()	200000	4
3	init()	a	0
4	init()	-1	3,14
5	getParam()	12, 16, 25	0
6			

Überprüfung zur Laufzeit

Applikation

Messung der
Testabdeckung



Erwartete Ergebnisse



?

==

15 42 77
68 24 00
56 07 22

Testergebnisse

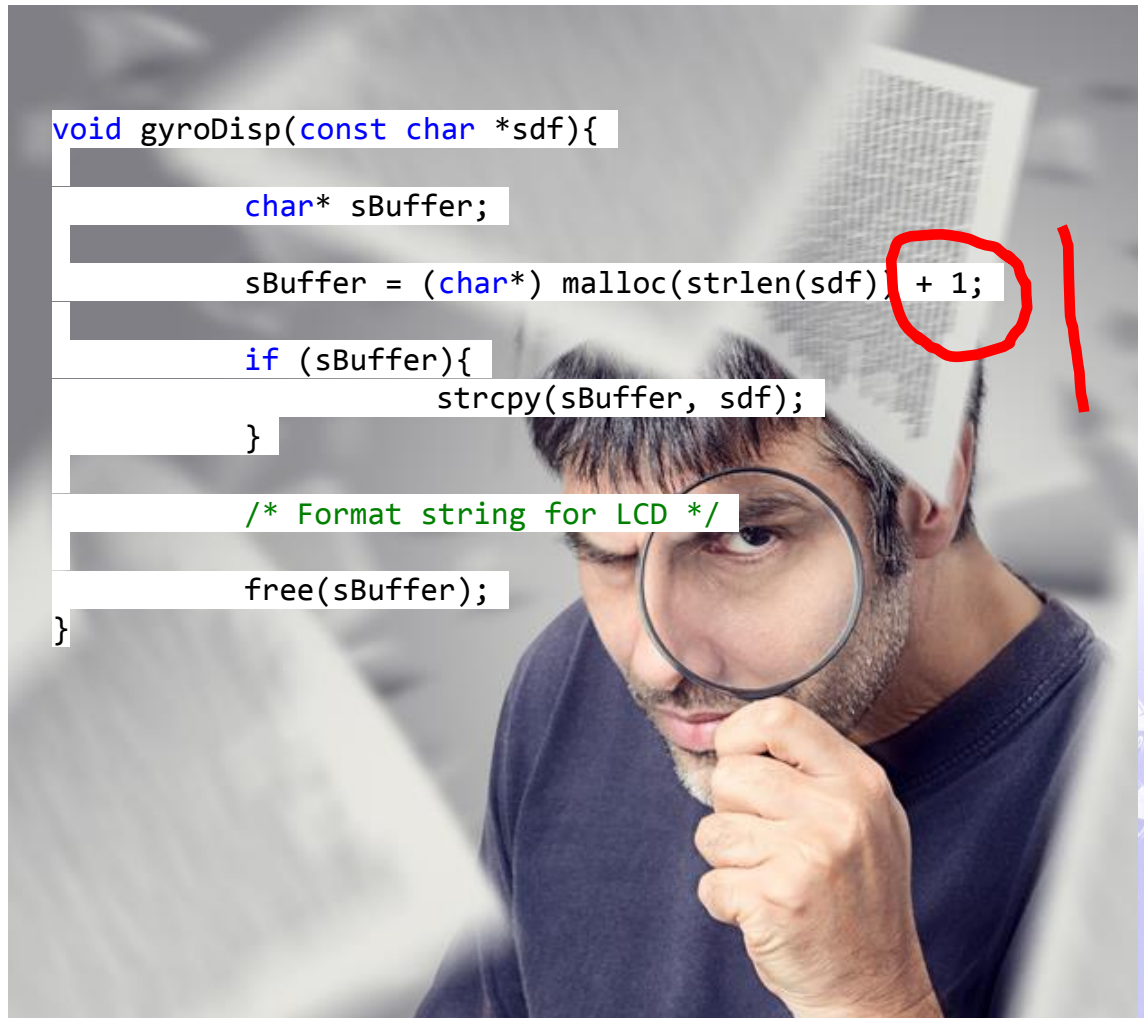


Statische Analyse

- Manuelle oder maschinelle Überprüfung zur Compile-Zeit
- Die Applikation wird nicht ausgeführt

Mögliche Überprüfungen:

- Syntax
- Semantik
- Kontrollflussanomalien
- Datenflussanomalien
- Kodierrichtlinien
- Architektur





Statische Analyse – Quellcode- oder Binärdatei?

```
int main(void){  
    printf("Hello World!\n");  
    return 0;  
}
```

Quellcode

Compiler

Optimizer

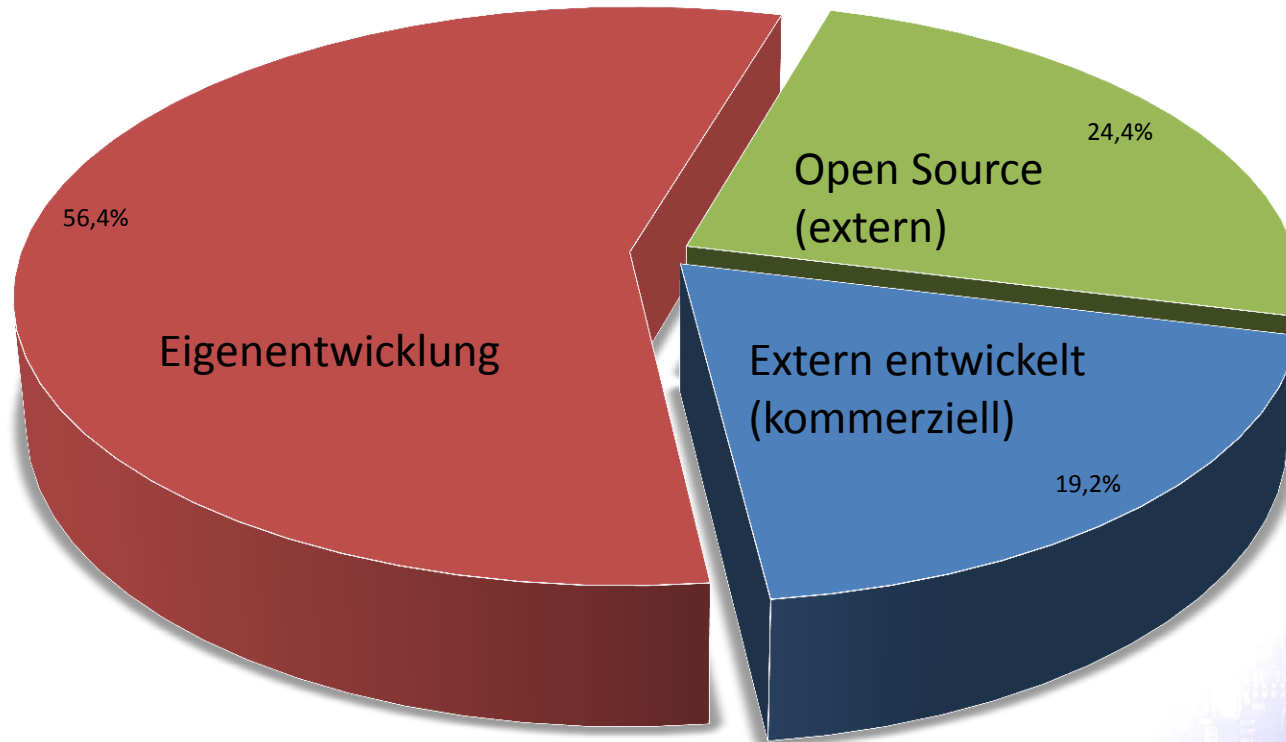
Linker

```
0100111001011101010110111000101011010001010100011101010100001010100  
010100100010101010001011110000101000011111101011110111101010000101101  
110100001110100010111101011101010001101001110111010001010011110100010  
100111101011100010100000011101010100101110010111001111010100010100010  
10010010101010011111100010100010100010000101010011110110110001010  
001100101010001111001000101111010001000011100001010101000011111000101
```

Binärdatei



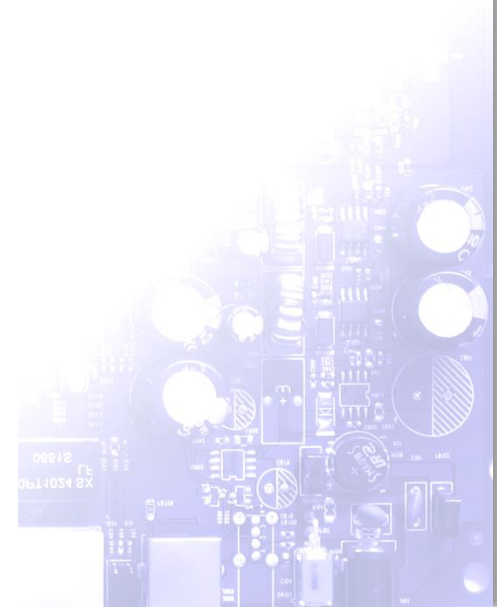
Anteil an extern entwickeltem Code



Quelle: VDC Research
2016

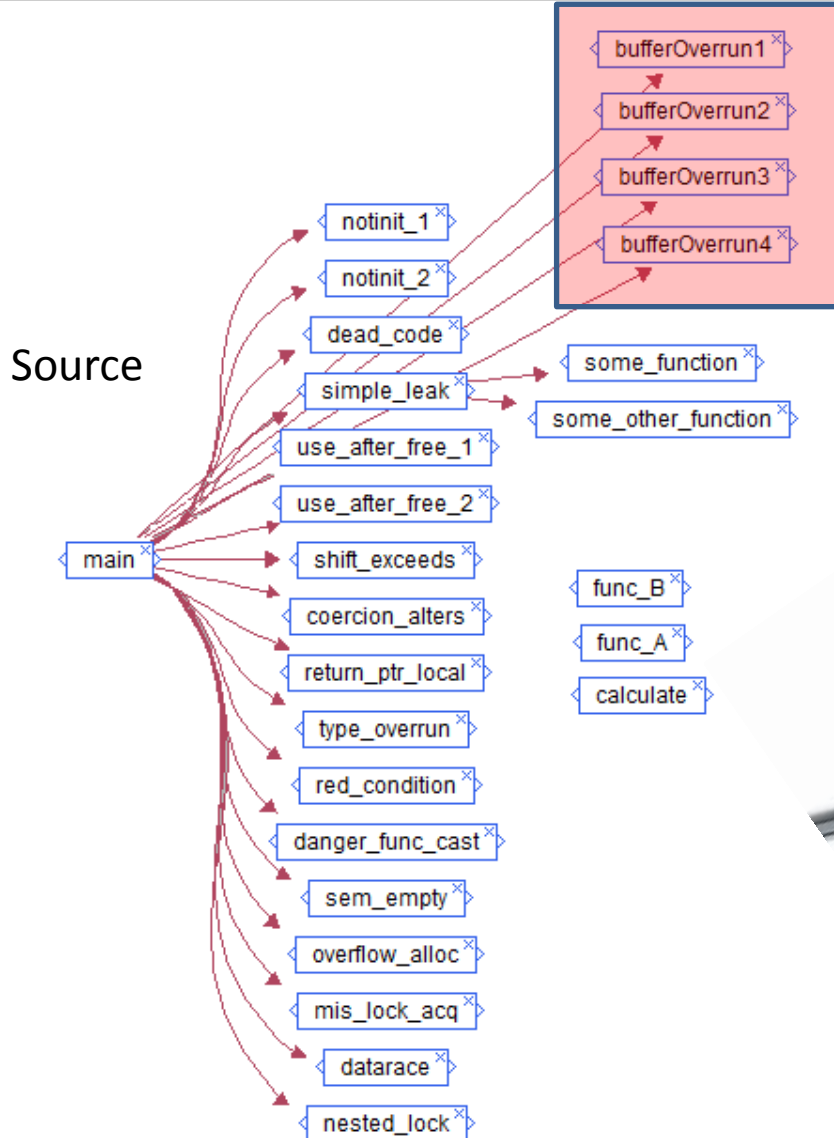


Extern entwickelte Komponenten





Einbinden von „fremden“ Binärdateien



Binary





Ein kleines Testprogramm

```
char bufferOverrun0(void){  
    char* buffer;  
    char ch;  
    buffer = (char*)malloc(5 * sizeof(char));  
  
    *(buffer + 0) = '0';  
    *(buffer + 1) = '1';  
    *(buffer + 2) = '2';  
    *(buffer + 3) = '3';  
    *(buffer + 4) = '4';  
    *(buffer + 5) = '5';    /* Ueberlauf */  
  
    ch = *(buffer + 5);  
  
    free(buffer);  
  
    return ch;  
}
```



Ein kleines Testprogramm (Fortsetzung)

```
char bufferOvrrun1(void){  
    char buffer[5];  
    char ch;  
    ch = 'c';  
    buffer[0] = '0';  
    buffer[1] = '1';  
    buffer[2] = '2';  
    buffer[3] = '3';  
    buffer[4] = '4';  
    buffer[5] = '5';    /* Ueberlauf */  
  
    return buffer[5];  
  
}
```





Ein kleines Testprogramm (Fortsetzung)

```
int some_function(){
    return 1;
}

int some_other_function(){
    return 0;
}

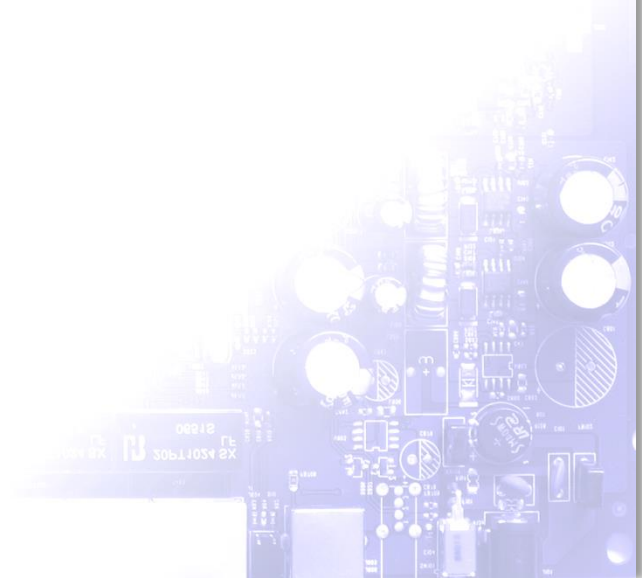
void callFunc(void){
    char *p;
    p = (char*)malloc(12);
    if (!p)
        return;
    if (!some_function()){
        free(p);
        return;
    }
    if (!some_other_function())
        return;
    free(p);
    return;
}
```

/* Speicherleck */



Ein kleines Testprogramm (Fortsetzung)

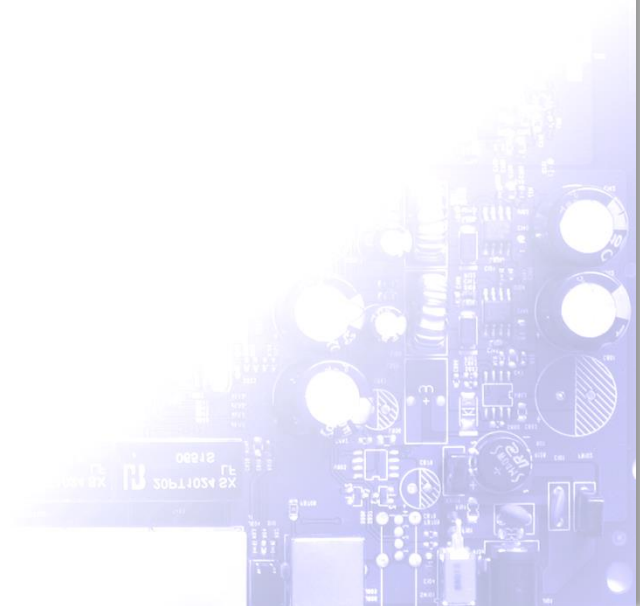
```
int shiftValue(void){  
    int a = 50;  
    int b = 0;  
    b = a << 200;    /* Shift ueberschreitet Laenge der Variablen */  
    return b;  
}
```





Ein kleines Testprogramm (Fortsetzung)

```
__asm__ (  
    ".globl _createArray\n\t"  
    ".def _createArray\n\t"  
    ".sc12\n\t"  
    ".type32\n\t"  
    ".endef\n\t"  
    "_createArray:\n\t"  
    "LFB30:\n\t"  
    ".cfi_startproc\n\t"  
    "pushl%ebp\n\t"  
    ".cfi_def_cfa_offset 8\n\t"  
    ".cfi_offset 5, -8\n\t"  
    "movl%esp, %ebp\n\t"  
    ".cfi_def_cfa_register 5\n\t"  
    "subl$40, %esp\n\t"  
    "movl$10, (%esp)\n\t"  
    "call_malloc\n\t"  
    "movl%eax, -12(%ebp)\n\t"  
    "cmpl$0, -12(%ebp)\n\t"  
    "jeL33\n\t"  
    "movl-12(%ebp), %eax\n\t"  
    "movl%eax, -16(%ebp)\n\t"  
    "movl-12(%ebp), %eax\n\t"  
    "movl%eax, (%esp)\n\t"  
    "call_free\n\t"  
    "movl-16(%ebp), %eax\n\t"  
    "movb$98, (%eax)\n\t" /* Verwendung nach "free()" */  
    "L33:\n\t"  
    "leave\n\t"  
    ".cfi_restore 5\n\t"  
    ".cfi_def_cfa 4, 4\n\t"  
    "ret\n\t"  
    ".cfi_endproc\n\t"  
);
```





Quellcode mit Inline Assembler

```
int add(int val1, int val2){  
  
    int a = val1;  
  
    __asm__ (  
        "add %1, %0\n\t"  
  
        : "+r" (a)  
        : "g" (val2)  
        : "cc"  
  
    );  
  
    return a;  
}
```





Optimierungsfehler

Originalcode

```
volatile int x = 0;

int condInc(){
    x = 5;
    if (x < 0){
        x++;
    }
    return x;
}
```

Fehlerhafte Optimierung

```
int x = 0;

int condInc(){
    x = 5;
    return x;
}
```



Optimierungsfehler

Sicherheitsproblem: Passwort verbleibt lesbar im Stack!

Original

```
int checkPermission(void){  
  
    char password[MAXLEN];  
    int len, valid = 0;  
  
    len = readPassword(password);  
  
    valid = checkPassword(len, password);  
  
    memset(password, '\\0', len);  
  
    return valid;  
}
```

Optimiert

```
int checkPermission(void){  
  
    char password[MAXLEN];  
    int len, valid = 0;  
  
    len = readPassword(password);  
  
    valid = checkPassword(len, password);  
  
    return valid;  
}
```

Problem aufgedeckt durch Microsoft Security Review 2002



Statische Binäranalyse – Mustererkennung



```
/* Shift Exceeds */  
int shiftValue(void){  
    int a = 50;  
    int b = 0;  
    b = a << 200;  
    return b;  
}
```

```
/* Dead Code */  
void checkValue(void){  
    int x = 1;  
    int y = 12;  
    if (x){  
        if (y > 10 && !x){  
            x++;  
            return;  
        }  
    }  
}
```

Programmiersprache?

Bitness?

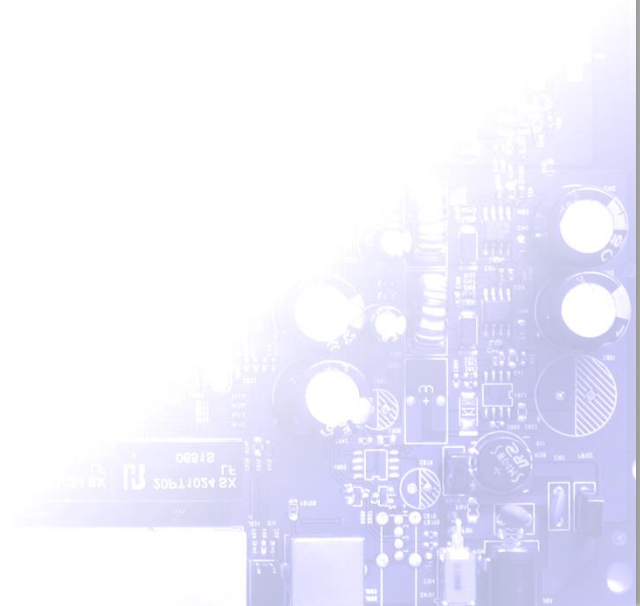
Compiler und Zielarchitektur?

Optimierungsgrad?



“Buffer Overrun” auf dem Stack

```
char bufferOverrun1(void){  
    char buffer[5];  
    char ch;  
  
    ch = 'c';  
  
    buffer[0] = '0';  
    buffer[1] = '1';  
    buffer[2] = '2';  
    buffer[3] = '3';  
    buffer[4] = '4';  
    buffer[5] = '5';  
  
    return buffer[5];  
}
```





Übersetzung in Assembler

```
.file      "bintest.c"
.text
.globl     _bufferOverrun1
.def       _bufferOverrun1;          .scl      2;          .type     32;
.undef
_bufferOverrun1:
LFB0:
.cfi_startproc
pushl      %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl       %esp, %ebp
.cfi_def_cfa_register 5
subl       $16, %esp
movb       $99, -1(%ebp)
movb       $48, -6(%ebp)
movb       $49, -5(%ebp)
movb       $50, -4(%ebp)
movb       $51, -3(%ebp)
movb       $52, -2(%ebp)
movb       $53, -1(%ebp)
movzbl     -1(%ebp), %eax
leave
.cfi_restore 5
.cfi_def_cfa 4, 4
ret
.cfi_endproc
```



Betrachtung Stack Frame

```
pushl    %ebp
movl     %esp, %ebp
subl     $16, %esp
movb     $99, -1(%ebp)
movb     $48, -6(%ebp)
movb     $49, -5(%ebp)
movb     $50, -4(%ebp)
movb     $51, -3(%ebp)
movb     $52, -2(%ebp)
movb     $53, -1(%ebp)
movzbl   -1(%ebp), %eax
leave
```



00007FF0h	
00007FF1h	
00007FF2h	
00007FF3h	
00007FF4h	
00007FF5h	
00007FF6h	
00007FF7h	
00007FF8h	
00007FF9h	'0'
00007FFAh	'1'
00007FFBh	'2'
00007FFCh	'3'
00007FFDh	'4'
00007FFEh	'5'
00008000h	04h
00008001h	80h
00008002h	00h
00008003h	00h
00008004h	3
00008005h	'B'
00008006h	6
00008007h	0





“Buffer Overrun” auf dem Heap

```
.file      "bintest.c"
.text
.globl     _bufferOverrun0
.def       _bufferOverrun0; .scl      2;
.type      32; .endif
_bufferOverrun0:
LFB0:
.cfi_startproc
pushl      %ebp
.cfi_def_cfa_offset 8
.cfi_offset 5, -8
movl       %esp, %ebp
.cfi_def_cfa_register 5
subl       $40, %esp
movl       $5, (%esp)
call       _malloc
movl       %eax, -12(%ebp)
movl       -12(%ebp), %eax
movb       $48, (%eax)
movl       -12(%ebp), %eax
addl       $1, %eax
...
```



Fazit

Die statische Binäranalyse ist sehr leistungsfähig, kann aber naturgemäß nicht die gleiche Fehlerrate wie die statische Quellcodeanalyse erreichen.

Herausforderung:

- Unterscheidung zwischen Code und Daten
- Abgrenzung von Variablen
- Abgrenzung von Instruktionen und Prozeduren
- Unterscheidung von globalen und lokalen Variablen

Ist kein Quellcode zu einer Binärdatei vorhanden, kann die statische Binäranalyse als wertvolle Ergänzung zum dynamischen „Black Box“ Test die Softwarequalität entscheidend verbessern.



VIELEN DANK !

Royd Lüdtke
Verifysoft Technology GmbH
luedtke@verifysoft.com
+49 781 127 8118-8

