Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Triple Modular Redundancy

Tobias Klaus, Florian Schmaus, Peter Wägemann

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
https://www4.cs.fau.de

25. April 2016

Überblick

1. C-Quiz Teil III
2. Wiederholung: Grundlagen Fehlerbäume
3. Wiederholung: Triple Modular Redundancy
4. Replikation von Code

Annahmen

- C99
- x86 bzw. x86-64, d. h.
  - vorzeichenbehaftete Integer als Zweierkomplement implementiert
  - `char` hat 8 Bit
  - `short` hat 16 Bit
  - `int` hat 32 Bit
  - `long` hat 32 Bit auf x86 und 64 Bit auf x86-64

Frage 7

Zu was wird `INT_MAX + 1` ausgewertet?

1. `0`
2. `1`
3. `INT_MAX`
4. `UINT_MAX`
5. nicht definiert

Erklärung

`signed int`-Überlauf ist nicht definiert.

Frage 8

Zu was wird `-INT_MIN` ausgewertet?

1. `0`
2. `1`
3. `INT_MAX`
4. `UINT_MAX`
5. `INT_MIN`
6. nicht definiert

Erklärung

Es gibt keine Zweierkomplementdarstellung für `-INT_MIN`

Frage 9

Angenommen `x` hat Typ `int`. Ist `x << 0` ...

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x?

Erklärung

- negative Werte können nicht nach links verschoben werden
- noch nicht einmal um `0` Bit

Überblick

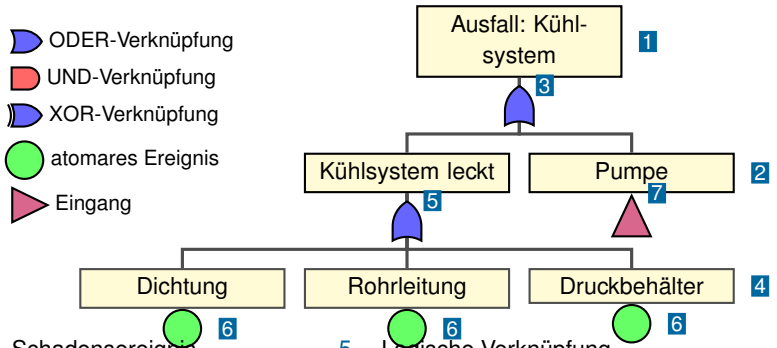
1 C-Quiz Teil III

2 Wiederholung: Grundlagen Fehlerbäume

3 Wiederholung: Triple Modular Redundancy

4 Replikation von Code

Fehlerbäume – Wiederholung



1. Schadensereignis
2. Ereignisse auf Ebene 2
3. Logische Verknüpfung
4. Ereignisse auf Ebene 3
5. Logische Verknüpfung
6. Atomare Ereignisse
7. Eingänge zerlegen den Fehlerbaum ↦ Neuer Teilbaum

Überblick

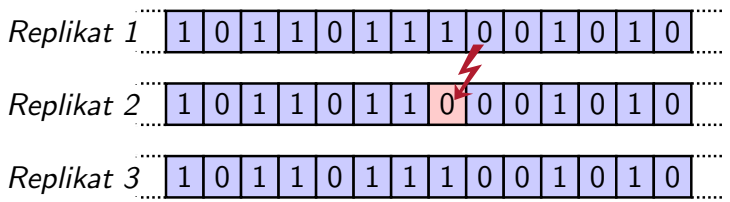
1 C-Quiz Teil III

2 Wiederholung: Grundlagen Fehlerbäume

3 Wiederholung: Triple Modular Redundancy

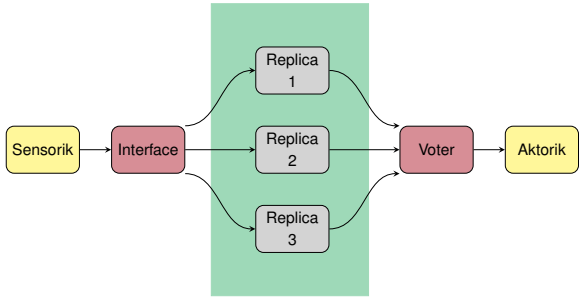
4 Replikation von Code

Fehlerhypothese



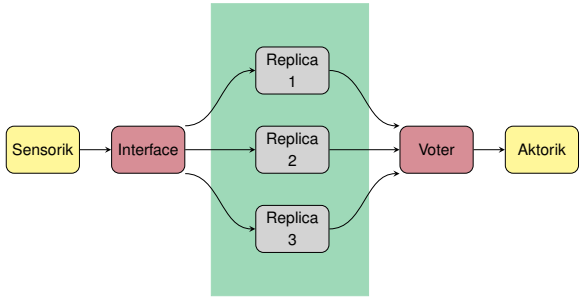
- **Wie viele Replikate** benötigt man?
- Arten des Fehlverhaltens (von n Replikaten sind f fehlerhaft)
  1. fail-silent $\mapsto$ Anzahl Replikate: $n = f + 1$
  2. fail-consistent $\mapsto$ Anzahl Replikate: $n = 2f + 1$
  3. malicious $\mapsto$ Anzahl Replikate: $n = 3f + 1$

Triple Modular Redundancy



- Schnittstelle sammelt Eingangsdaten (Replikdeterminismus)
- Verteilt Daten und aktiviert Replikate
- Mehrheitsentscheider (Voter) wählt Ergebnis
- Ergebnis wird an Aktuator versendet

Triple Modular Redundancy



Redundanzbereich

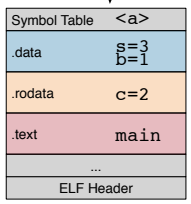
Ausschließlich Replikatausführung

- Erweiterung der Ausgangsseite mit Informationsredundanz
- Mehrheitsentscheid über Berechnungsergebnisse

Speicherorganisation auf einem Mikrocontroller

```c
int a;                    // a: global, uninitialized
int b = 1;                // b: global, initialized
const int c = 2;          // c: global, const

void main() {
  static int s = 3;       // s: local, static, initialized
  int x, y;               // x: local, auto; y: local, auto
  char* p = malloc( 100 );  // p: local, auto; *p: heap (100 byte)
}
```

compile / link — Quellprogramm

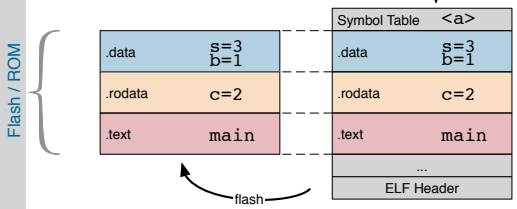


ELF-Binary

Speicherorganisation auf einem Mikrocontroller

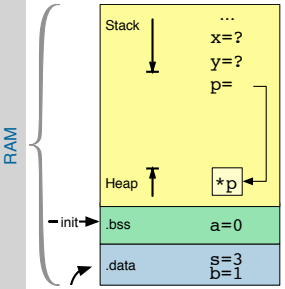
```c
int a;                    // a: global, uninitialized
int b = 1;                // b: global, initialized
const int c = 2;          // c: global, const

void main() {
  static int s = 3;       // s: local, static, initialized
  int x, y;               // x: local, auto; y: local, auto
  char* p = malloc( 100 );  // p: local, auto; *p: heap (100 byte)
}
```

compile / link — Quellprogramm



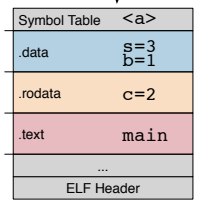
μ-Controller ELF-Binary

Speicherorganisation auf einem Mikrocontroller




```c
int a;                    // a: global, uninitialized
int b = 1;                // b: global, initialized
const int c = 2;          // c: global, const

void main() {
  static int s = 3;       // s: local, static, initialized
  int x, y;               // x: local, auto; y: local, auto
  char* p = malloc( 100 );  // p: local, auto; *p: heap (100 byte)
}
```

compile / link — Quellprogramm



μ-Controller ELF-Binary

C-Code vs. Assembler-Code

C-Code

```c
int a;
int b = 1;
const int c = 2;
  void main() {
  static int s = 3;
  int x, y;
  char* p = malloc( 100 );
}
```

Assembler-Code

```
4004f0 <main>:
  4004f0: push   %rbp
  4004f1: mov    %rsp,%rbp
  4004f4: sub    $0x10,%rsp
  4004f8: movabs $0x64,%rdi
  4004ff: 00 00 00
  400502: callq  4003e0 <malloc@plt>
  400507: mov    %rax,-0x10(%rbp)
  40050b: add    $0x10,%rsp
  40050f: pop    %rbp
  400510: retq
```

Wo können Datenfehler auftreten?

1. RAM: `-0x10(%rbp)`
2. Allgemeine CPU-Register: `%rsp`
3. Sonstige CPU-Register: `%rip`, `%rflags`

Replikdeterminismus

Replikat 1

```
void repl_2(void* p){
  ticks_t time =
    ezs_get_time();
  ...
}
```

Replikat 2

```
void repl_2(void* p){
  ticks_t time =
    ezs_get_time();
  ...
}
```

Replikat 3

```
void repl_3(void* p){
  ticks_t time =
    ezs_get_time();
  ...
}
```

Sicherstellung Replikdeterminismus

- Globale diskrete Zeitbasis
- Einigung über Eingabewerte
- Statische Kontrollstruktur der Replikate
- Deterministische Algorithmen

☞ Sicherstellung, dass Replikat *innerhalb Zeitspanne* Ergebnis liefert

Überblick

1 C-Quiz Teil III

2 Wiederholung: Grundlagen Fehlerbäume

3 Wiederholung: Triple Modular Redundancy

4 Replikation von Code

Code-Replikation: Stringification

Stringification von CPP

```
#define MAX(repl, a, b) int max##repl(int a, int b){ \
                          return a > b ? a : b;      \
                        }
int main() {
  MAX(1, 23, 42);
  MAX(2, 23, 42)
}
```

- Verwendung des C-Präprozessors (CPP)
- ##: „Token Pasting Operator“
- Konkatenieren zweier Token zu einem

☞ Es geht eleganter ...

C++ Templates

C++ Template

```
template <typename T>
T max(T x, T y)
{
  T value;
  if (x < y)
    value = y;
  else
    value = x;
  return value;
}
...
double md = max<double>(2.3, 4.2);
auto   mi = max<int>(23U, 42);
```

- Templates ermöglichen generische Programmierung
- Wiederverwendung durch **Parametrisierung**
- Unterscheidung von Funktions- & Klassen-Templates
- Expansion zur Compilezeit $\rightsquigarrow$ Quelltext muss verfügbar sein (im Header)
- „Code Bloat” beim Compilieren $\rightarrow$ **nutzbar für Replikation von Code**

C++ Templates

C++ Template Spezialisierung

```
template <float T>
T my_func(T x, T y) // specialized template for T == float
{
  T a = x;
  ...
}
```

- Spezialisierungen von Templates möglich
  - Effizientere Implementierung für bestimmte Typen
- Nutzbar zum „Zählen” von Templates
- Mehrere Template-Parameter möglich:

Mehrere Template-Parameter

```
template <typename A, typename B, typename C>
T my_other_func(A x, B y, C z) {
  ...
}
my_other_func<Dog, Cat, Mouse>(m, n, o);
```

Umgang mit Assembly-Code

Assembly-Code gemischt mit Source-Code

```
int main(){
  4007cd: 55                    push   %rbp
  4007ce: 48 89 e5              mov    %rsp,%rbp
  4007d1: 48 83 ec 10           sub    $0x10,%rsp
int a = max<int>(23U, 42);
  4007d5: be 2a 00 00 00        mov    $0x2a,%esi
  4007da: bf 17 00 00 00        mov    $0x17,%edi
  4007df: e8 04 01 00 00        callq  4008e8 <_Z3maxIiET_S0_S0_>
  4007e4: 89 45 fc              mov    %eax,-0x4(%rbp)
std::cout << a << "\n";
  4007e7: mov    -0x4(%rbp),%eax
  ...
```

- `objdump`: Ausgabe von Informationen von Objektdateien
- Nützliche Optionen
  - `-S`: Ausgabe von Quell-Code im Assembly-Code (Debug-Symbole notwendig)
  - `-D`: alle Sektionen disassemblieren

C-Code in C++ Einbinden

C Linkage

```
// C++ code
extern "C" void f(int); // one way
extern "C" {            // another way
  int g(double);
  double h();
};
void code(int i, double d)
{
  f(i);
  int ii = g(d);
  double dd = h();
  // ...
}
```

- Name mangling von C++ verhindern

⇒ C++-Code aus C-Code aufrufen