Test-Generator entdeckt Compiler-Fehler

# Schwachstellensucher

**Christian Lindig**

C-Compiler müssen Funktionsaufrufe in Maschinencode übersetzen, der komplizierten Aufrufkonventionen folgt. Dabei unterlaufen selbst ausgereiften Compilern Fehler, wie der Testfallgenerator Quest aufdeckt.

Die Chance ist klein, in einem C-Compiler auf einen Fehler zu treffen. Folglich konzentrieren sich Tests von C-Compilern auf den Sprachumfang, Codeoptimierung und Bibliotheken [1 bis 4]. Trotzdem kann ein gezielter Test Schwachstellen aufdecken. Eine solche ist die Übersetzung der Parameterübergabe an Funktionen. Was als Funktionsaufruf im C-Quelltext ganz einfach aussieht, erfordert vom Compiler eine Übersetzung in Maschinencode, der komplizierte Aufrufkonventionen einhalten muss.

Der Code in Listing 1 demonstriert einen Fehler der GNU Compiler Collection (GCC) auf Mac OS X 10.3. Wenn man den Code übersetzt und ausführt, schlägt die Assertion in Zeile 20 fehl:

```
$ gcc -O2 -o bug bug.c
$ ./bug
bug:20: failed assertion "y.f == i.f"
Abort trap
```

Die Assertion prüft, ob *main* das Argument vom Typ *union C* richtig an die Funktion *f* übergibt. Dazu vergleicht sie den Wert des Parameters *y.f* mit dem Wert der globalen Variable *i.f*, die *main* als vierten Parameter verwendet.

Eine Erklärung für den beobachteten Fehler lautet: der ANSI-Standard verlangt, dass der Compiler ein *short int* bei der Übergabe als variables Argument zu einem *int* ausweitet. Das darf jedoch nicht bei einem *short int* als Teil einer *union* geschehen. Da der Typ *union C* dieselbe Größe hat wie *short int*, versucht der GNU-Compiler wohl, *union C* zu *int* aufzuweiten – was falsch wäre.

Der Code in Listing 1 enthält C-Sprachelemente, die in üblichen Quellen selten anzutreffen sind: Strukturen, die das Hauptprogramm als Werte an Funktionen übergibt, Funktionen mit variabler Anzahl von Argumenten und Unions. Die zufällige Kombination dieser an sich schon selten verwendeten Sprachmittel provoziert Fehler in

der Übersetzung von Funktionen und deren Aufrufen.

## Chipschmieden legen Konventionen fest

Die Übersetzung von Funktionsaufrufen in C-Compilern ist vor allem deshalb fehlerträchtig, weil der generierte Maschinencode aufrufkompatibel sein muss zu dem Code anderer Compiler und bestehender Bibliotheken. Dazu spezifizieren Hardwarehersteller wie Intel in Prozessorhandbüchern eine Aufrufkonvention [6], die alle Compiler einer Plattform respektieren sollen. Die Empfehlung legt fest, welche Prozessorregister (abhängig vom Datentyp) für die Übergabe von Argumenten dienen, welche Argumente auf dem Stack übergeben werden, welches Alignment sie dabei beachten müssen und wie Funktionen Ergebnisse zurückgeben.

Da dies plattformspezifisch ist, summiert sich der Aufwand für Implementierung und Test von Aufrufkonventionen bei Compilern wie GCC oder LCC (A Retargetable C Compiler) [5], da sie Code für mehrere Plattformen generieren können. Sie sind deshalb besonders anfällig für Fehler. Zudem wuchs die Komplexität der Spezifikation vor allem für C, weil die Sprache im Laufe der Zeit neue Konstrukte aufgenommen hat: Übergabe von Strukturen an Funktionen, Einführung von Funktionsprototypen, Standardisierung des Zugriffs auf Extraargumente in Funktionen mit variabler Anzahl von Argumenten, neue Datentypen wie *long long* oder *long double*. Gleichzeitig mussten die Erweiterungen die Binärkompatibilität mit bestehenden Konventionen einhalten.

Native Compiler für Java und andere moderne Sprachen sind von den Schwierigkeiten weit weniger betroffen als C. Obwohl ihre Typstruktur oft reichhaltiger ist, implementieren sie einfachere Aufrufkonventionen. Zudem sind sie auch durch ihre Jugend weniger vom Zwang zur Kompatibilität mit alten Versionen betroffen.

**iX-TRACT**

- Prozessorhersteller legen Konventionen fest, um die Kompatibilität zwischen Compilern und Bibliotheken sicherzustellen.
- Aufgrund der Kompliziertheit implementieren Compilerentwickler die Konventionen meist nicht vollständig.
- So kann es bei der Übergabe von Parametern selbst bei gestandenen Compilern zu Fehlern kommen.

## Quest stellt die Fragen

Code zum Test der Parameterübergabe kann man (wie in Listing 1) mit „Quest“ zufällig generieren; er enthält Assertions, die bei einem Fehler im Compiler unwahr sind. Der Test eines Compilers kann einfach in einer Endlosschleife laufen, die Code mit Quest generiert, übersetzt und ausführt. Die folgende Schleife für die Unix Bourne-Shell (*/bin/sh*) bricht über *break* ab, wenn entweder der Compiler oder das Testprogramm einen Fehler meldet; *echo* zeigt nur das Fortschreiten des Tests an.

```
while true;
   do quest > bug.c
   gcc -O2 -o bug bug.c || break
   ./bug || break
   echo -n .
done
```

In der Tabelle „Mit Quest gefundene …“ sind die Fehler aufgeführt, die dabei durch Quest provoziert wurden. Detaillierte Beschreibungen der einzelnen Fälle liefert die offizielle Nummer des Bugreports für den jeweiligen Compiler. Obwohl die Untersuchung der C-Compiler nur unter Unix stattfand, sind ähnliche Fehler auf anderen Plattformen zu erwarten.

Tests sind nicht nur für Compiler-Entwickler nützlich, Anwender können sie auch zur Qualitätskontrolle durchführen. In der Regel begrüßen die Entwickler von Compilern Fehlerberichte und bieten für deren Einreichung eine Webschnittstelle an. Die Entwickler des GCC stellen auf Unix-Systemen zusätzlich das Kommando *gccbug* für die formgerechte Übermittlung per E-Mail bereit.

Anders als in Listing 1 gezeigt, enthält eine von Quest generierte Datei in der Standardeinstellung nicht einen, sondern 20 Testfälle. Der generierte Code verwendet die Header *assert.h* und *stdio.h*, ist konform zum ANSI-C-Standard von 1989 und sollte sich daher auf den meisten Plattformen übersetzen lassen.

Quest steht als Quelltext und als ausführbares Programm für Linux/Debian, Windows, und Mac OS X zum Download bereit. Für andere Plattformen kann man Testcode zunächst auf einer der unterstützten Plattformen erzeugen und dann auf der fremden übersetzen und analysieren.

**Listing 1**

```
 1  #include <stdarg.h>
 2  #include <assert.h>
 3  struct B {double d; int e;}
 4 h = { 78.01, 834 };
 5  union  C {short int f; char g;}
 6 i = { 68 };
 7  struct D {char j; double k;}
 8 n = { 'c', 31.01 };
 9  struct E {long long l; double m;}
10 o = { 167L, 17.2 };
11
12  void f(struct D a, struct E b, ...)
13  {
14 va_list ap;
15 struct B x;
16 union  C y;
17 va_start (ap, b);
18 x = va_arg(ap, struct B); /*3*/
19 y = va_arg(ap, union C);  /*4*/
20 assert (y.f == i.f);  /* fails */
21 va_end (ap);
22 return c;
23  }
24  int main(int argc, char *argv[]) {
25 f(n, o, h, i);
26 return 0;
27  }
```

**Missglückt: *gcc* 3.3 auf Mac OS X 10.3 übergibt das vierte Argument *union C i* nicht richtig an Funktion *f*: die Assertion in Zeile 20 schlägt fehl.**

## Prinzipielle Unversehrtheit der Parameter

Solche Testfälle sind um ein einfaches Prinzip konstruiert: Ein als aktueller Parameter an eine Funktion übergebener Wert muss diese unverändert als formalen Parameter erreichen. Der generierte Code übergibt dazu einen bekannten Wert an eine Funktion, die per Assertion prüft, ob der

**Listing 2**

```
int    x = 6362;   /* zufällig */
short *y = (short*) 6328282U;
char   z = 'q';
char f(int a, short* b) {
    assert(a == x);
    assert(b == y);
    return z;
}
void g(void) {
    char c;
    c = f(x,y);
    assert(c == z);
}
```

**Schema der Code-Generierung für eine Funktion *char f(int, short*)*: Funktion *g* ruft *f* und übergibt die Werte in globalen Variablen, die *f* prüft. Analog dazu überprüft *g* die Rückgabewerte.**

**Listing 3**

```
ANSI.mems    = R.choose(1,3)
ANSI.argc    = R.choose(1,10)
ANSI.vargc   = R.oneof
  { R.unit(0)     -- zero var args
  , R.choose(1,4) -- 1-4  var args
  }
ANSI.simple = R.oneof
  { R.any_int   -- char, short, int, long
  , R.any_float -- float, double
  }
ANSI.array_size = R.freq
  { 2,R.unit(1) -- more frequent: size 1
  , 1,R.unit(2) -- less frequent: size 2
  , 1,R.unit(3) -- less frequent: size 3
  }
ANSI.bitfields = R.list
  ( R.choose(0,4) -- number of bitfields
  , R.bitfield(R.choose(2,12)) -- size
  )
function ANSI.arg_(issimple)
  if issimple then return ANSI.simple
  else return R.smaller
    { ANSI.simple
    , R.pointer(ANSI.arg)
    , R.array(ANSI.arg ,ANSI.array_size)
    , R.struct(R.concat
      { R.list(ANSI.mems,ANSI.arg)
      , ANSI.bitfields
      , R.list(ANSI.mems,ANSI.arg)
      })
    , R.union(R.list(ANSI.mems,ANSI.arg))
    }
  end
end
ANSI.arg = R.bind(R.iszero,ANSI.arg_)
function ANSI.test () return
  { args    = R.list(ANSI.argc,ANSI.arg)
  , varargs = R.list(ANSI.vargc,ANSI.varg)
  , result  = ANSI.result -- result type
  , static  = R.flip      -- static func?
  }
end
```

**Ausschnitt aus der Definition des Generators für ANSI-C in Lua. *ANSI.arg* ist ein Generator für C-Typen, die als Funktionsparameter verwendet werden.**

empfangene Wert tatsächlich mit dem zuvor übergebenen übereinstimmt.

Die Konstruktion eines Testfalls ist typgetrieben. Quest generiert zufällig einen Funktionsprototypen, wie

```
char f(int, short*)
```

und konstruiert daraus zwei Funktionen *f* und *g* wie in Listing 2. Für jeden Parameter von *f* legt Quest eine globale Variable an und initialisiert sie. Funktion *g* übergibt ihre Werte an *f*, die mit Hilfe von Assertions prüft, ob die aktuellen Parameter die Werte der korrespondierenden globalen Variablen tragen. Schließlich gibt *f* einen Wert (vom Typ *char*) zurück, den *g* analog zu *f* prüft.

Bei einem Struct als Parameter generiert Quest eine Kette von Assertions, die jede Komponente des *struct* einzeln vergleichen. Es behandelt Zeiger wie vorzeichenlose Zahlen und vergleicht nur ihre Werte – nicht worauf sie zeigen.

Jeder Testfall stellt fest, ob ein Compiler die Parameterübergabe konsistent implementiert. Er testet nicht, ob ein Compiler tatsächlich eine Aufrufkonvention korrekt implementiert. Dies kann man zusätzlich untersuchen, wenn ein Referenzcompiler vorliegt, der eine Konvention korrekt umsetzt.

Quest kann dazu die Funktionen *f* und *g* in zwei unabhängige Dateien schreiben. Übersetzt etwa ein Referenzcompiler *g*, während der fragliche Compiler sich *f* vornimmt, und treten bei den gebundenen Objektdateien Fehler auf, deuten dies auf eine Verletzung der Aufrufkonvention durch den getesteten Compiler hin. In jedem Fall lässt sich so die Binärkompatibilität von Compilern überprüfen: GCC und LCC zum Beispiel sind unter Linux nicht binärkompatibel bei der Übergabe von Structs.

## Tuning tiefer gelegt

Quest generiert ANSI-C89-Code, aber manche C-Compiler können mehr. Deswegen bietet Quest speziell einen Generator für den GCC (*quest -test gcc*), der Arrays der Länge null und leere *struct*-Deklarationen akzeptiert. Wem das nicht reicht, kann vorhandene Generatoren verändern oder seine eigenen definieren.

In Quest steuert das Generieren eines Testfalls die eingebaute Skript-Sprache „Lua“, in der auch alle vordefinierten Generatoren implementiert sind. Lua ähnelt Pascal mit einem mächtigem Datentyp *table*, der zur Modellierung von Listen und Records dient. Damit man Testgeneratoren kompakt aufbauen kann, erhielt Lua in Quest eine Erweiterung durch einen speziellen Datentyp für Generatoren.

Da das Generieren des Codes typgetrieben ist, reicht es, wenn ein Generator den Prototyp einer Funktion definiert – alles Weitere ergibt sich automatisch. Es würde zu weit führen, die Details des Verfahrens zu erklären, aber Listing 3 dürfte einen Eindruck vermitteln. Ein typischer Generator besteht aus weniger als hundert Zeilen Lua, von denen die Abbildung einen Ausschnitt zeigt.

Die Lua-Funktionen *ANSI.arg* und *ANSI.arg_* definieren gemeinsam den Generator für Funktionsparameter. In *ANSI.test* entsteht damit eine Argumentliste der Länge eins bis zehn (*ANSI.argc*). Ein Parameter ist ein einfacher Typ (*ANSI.simple*), ein Array, Pointer, Struct oder *union*-Typ. Sie enthalten rekursiv wieder einen *ANSI.arg*-generierten Typ. Strukturen bestehen aus bis zu vier Bitfield-Deklarationen, die zwei bis zwölf Bits breit sind.

Der einfachste Weg zu einem eigenen Generator führt über die Veränderung der Lua-Quelle. Dazu kann man sie mit *quest -lua* ausgeben, verändern und als Datei wieder an Quest übergeben. Einfache Modifikationen wären zum Beispiel, die Verwendung von Fließkommatypen zu unterdrücken,

### Fundorte im Web

| | |
|---|---|
| Quest | www.St.Cs.Uni-Sb.De/~Lindig/Src/Quest/ |
| Lua | www.lua.org/, 2005 |

### Mit Quest gefundene Fehler in Unix-C-Compilern

| Compiler und Optionen | Plattform | Kommentar |
|---|---|---|
| SGI MipsPro 7.3.1.3m, -O3 | Irix 6.5/MIPS | fehlerhafte Übergabe von *struct* |
| GCC 2.95.3, -O2 | SunOS 5.8/Sparc | fehlerhafte Übergabe von *double* als *var arg* |
| GCC 2.95.4, -O | Linux/x86 | Compilerabsturz, Bug #16819, auch in 3.2.2 und 3.3.3, behoben in GCC 3.4 |
| GCC 3.3 | Irix 6.5/MIPS | fehlerhafte Übergabe von *union* als *var arg*, Bug #19268, behoben in GCC 3.4 |
| GCC 3.3 | MacOS X 10.3 | siehe Abbildung 1, Bug #18742 |
| LCC 4.2 | Linux/x86 | fehlerhafte Übergabe von *double* als *var arg* |
| PathCC 1.4, -O2 -m32 | Linux/x86 | fehlerhafte Übergabe von *float*, behoben in Release 2.0 |
| PathCC 1.4, -O2 -m32 | Linux/x86 | fehlerhafte Übergabe von *union* mit *struct*, behoben in Release 2.0 |
| PathCC 2.0, -Ofast | Linux/x86 | Compilerabsturz mit floating-point exception, Bug #5273, behoben in Release 2.1 |
| Intel ICC 8.1 | Linux/x86 | fehlerhafte Übergabe von *var arg*, Bug #292019 |

oder die Anzahl der Argumente *ANSI. argc* auf ein Intervall von 5 bis 8 mit *choose (5,8)* einzuschränken. Das Quest-Manual (*quest -man*) erläutert die Komposition von komplexen Generatoren aus den gegebenen Primitiven.

## Fazit

Hinter einem Funktionsaufruf in C verbirgt sich viel Arbeit für den Compiler beim Übersetzen in Maschinencode, denn er muss die plattformspezifische Aufrufkonvention einhalten, um die Kompatibilität mit anderen Compilern und bestehenden Bibliotheken zu garantieren. Die Konventionen sind für C häufig so kompliziert, dass Compiler sie nur unvollständig implementieren. Quest ist ein Werkzeug zum zufallsgesteuerten Erzeugen von Testcodes, die bei ihrer Ausführung die Konsistenz von Funktionsaufrufen überprüfen: Übergebene Parameter müssen unverändert in einer Funktion ankommen. Untersuchungen zeigten, dass selbst ausgereifte Compiler fehlerhaften Code erzeugen können. Durch die in Quest eingebaute Skriptsprache kann man Testgeneratoren modifizieren oder ersetzen. (rh)

CHRISTIAN LINDIG

ist wissenschaftlicher Mitarbeiter am Lehrstuhl für Softwaretechnik an der Universität des Saarlandes in Saarbrücken.

## Literatur

[1] Michael Riepe; Compiler; Aller guten Dinge; GCC 3.0; *iX* 9/2001, S. 60

[2] Herbert Schmid; Linux-Sprinter; Intel C++ Compiler für Linux; c't 23/01, S. 222

[3] Andreas Stiller; Ein weites Feld; SPEC und die AMD64-Linux-Compiler von GNU, PathScale, PGI und Intel; c't 13/04, S. 162

[4] Ralph Hülsenbusch; Compiler; Verteilte Vorlage; Version 4 der Gnu Compiler Collection 3; *iX* 7/2004, S. 62

[5] Chris W. Fraser and David R. Hanson; A Retargetable C Compiler: Design and Implementation; Benjamin/Cummings Pub. Co., 1995 www.cs.princeton.edu/software/lcc/.

[6] Intel. IA-32 Intel Architecture Software Developers's Manual, Vol. 1.; Intel Corporation, 2003

[7] Andreas Bauer; Compiler; Übersetzerbau; GCC-Interna im Detail; *iX* 11/2004, S. 124

### Quest-Optionen

| Option | Effekt |
|---|---|
| -test x | verwende Testgenerator x |
| -list | zeige vorhandene Testgeneratoren |
| -n 123 | 123 Testfälle pro Datei |
| -s 3 | Komplexität generierter Typen |
| -1 | generiere Testdatei nach stdout |
| -2 | generiere zwei Dateien: quest-main.c, quest-callee.c |
| -o name | verwende name statt quest |
| -lua | zeige eingebauten Lua Code |
| file.lua | lade file.lua in den Interpreter |
| -h | zeige Kommandoübersicht |
| -man | zeige Manual |