embeddedworld2016
Exhibition&Conference
...it's a smarter world

Software-Qualität und -Sicherheit:

C/C++-Code absichern

(Bild: vdimage – Shutterstock)

Software, die in industriellen Anwendungen und sicherheitskritischen Bereichen eingesetzt wird, muss nicht nur zuverlässig funktionieren, sondern auch robust gegenüber Eindringlingen sein. Wer ein paar Regeln beachtet, macht seine Software sicherer und verbessert damit auch die Software-Qualität.

Von Greg Davis

Mit unzuverlässigen Geräten und Maschinen kennen wir uns aus. Zuhause betreibe ich einen gemieteten Digital-Videorekorder von meinem Kabelanbieter. Er stürzt oft ab, wenn ich schnell vorspulen und Werbung überspringen möchte. Ich habe herausgefunden, dass man eine andere Taste drücken muss, die stattdessen schnell vorspult. Der Rekorder verhält sich dann korrekt. Steht der Rekorder kurz vor der Markteinführung, befindet sich der Hersteller unter hohem Zeitdruck. Dann werden nur noch die kritischsten Fehler hinsichtlich der Bedienbarkeit beseitigt. Ein Unternehmen, das sich über Sicherheit keine Gedanken macht, betrachtet dies als „gut genug".

Wenn Produkte jedoch sicher vor Hacker-Angriffen gemacht werden sollen, müssen strengere Standards befolgt werden. Hacker verwenden bekanntermaßen extreme Tests, Fuzzing und statische Analyse, um Zuverlässigkeitsprobleme in Produkten zu finden. Werden diese gefunden, analysieren Hacker die Probleme, um zu sehen, ob sie häufig auftreten, wie z.B. ein Pufferüberlauf. Die vielversprechendsten Fehler werden weiter genutzt, da Hacker nach einem Exploit suchen, das ihnen die Kontrolle über das System gibt. Dabei ist Quellcode für den Hacker nicht erforderlich, um diese Angriffe auszuführen. Quellcode erleichtert zwar die Arbeit, wird aber dafür nicht benötigt. Das Gleiche gilt für Sicherheitsmechanismen wie ASLR (Address Space Layout Randomization, Adressverwürfelung), die Ausführung von Bits in der MMU oder für Stack Canaries; ihre Anwesenheit macht die Aufgabe des Hackers nur unwesentlich schwieriger. „Gut genug" ist also nicht gut genug, um einen Hacker fernzuhalten. Ein wesentlich höheres Maß an Zuverlässigkeit muss erreicht werden.

Der Aufbau der Software ist auf jeden Fall wichtig. Geht es aber um die Sicherheit, kommt es auch darauf an,

wie der Code geschrieben ist. Betrachtet man die jüngsten Sicherheitslücken in einem Browser, basieren nach meiner Zählung 75 % der kritischen Probleme auf typischen C/C++-Programm-Mängeln (wie Array-Überschreitungen, Use after Free) anstatt auf architektonischen Mängeln (wie Privilege Escalation oder Security Bypasses). Dieser Beitrag konzentriert sich deshalb auf Tools und Techniken, mit denen sich solche Codierungsfehler vermeiden lassen.

Codierungsstandards einführen

Ein wirksames Instrument ist, C und C++ so weit zu beschränken, dass problematische Bereiche der Sprache vermieden werden. Codierungsstandards machen genau das. Eine Reihe von Standards sind etabliert: MISRA C, MISRA C++, „The Power of Ten", der C++-Codierungsstandard Joint Strike Fighter und der CERT-Standard. Es folgen einige Beispiele, welche Art von Regeln diese Standards nutzen.

Können Sie das Problem in **Listing 1** ausfindig machen? – Das Problem ist, dass in C und C++ jede Konstante, die mit einer 0 beginnt, eine oktale Konstante ist. Damit gilt: 64 == 0x40 und ein valides Abbild von Bit 6, 064 == 52 oder 0x34. Viele Codierungsstandards vermeiden dieses Problem, indem sie oktale Konstanten nicht erlauben. Die letzte Zeile muss daher entweder als

```
line_c |= 64; /* set bit 6 */
```

oder als

```
line_c |= 0x40;
          /* set bit 6 */
```

ausgedrückt werden.

Ein weiteres Beispiel: Erkennen Sie den Fehler in **Listing 2**? Der Code wird als „OK" ausgeführt, aber nicht in der Weise, wie es sich der Programmierer gedacht hat. Das Problem ist, dass die Konstanten 0.0 und 0.5 in doppelter Genauigkeit ausgedrückt werden, während die Eingangsvariable nur einfache Genauigkeit aufweist. Die gesamte Gleitkommazahl der Funktion muss daher in doppelte Genauigkeit umgewandelt werden, bevor sie verarbeitet werden kann. Dieses Missverständnis zwischen Programmierer und Compiler kann später zu Zuverlässigkeits- und Performance-Problemen führen. Codierungsstandards können dies verhindern, indem implizite Type Casts verhindert werden.

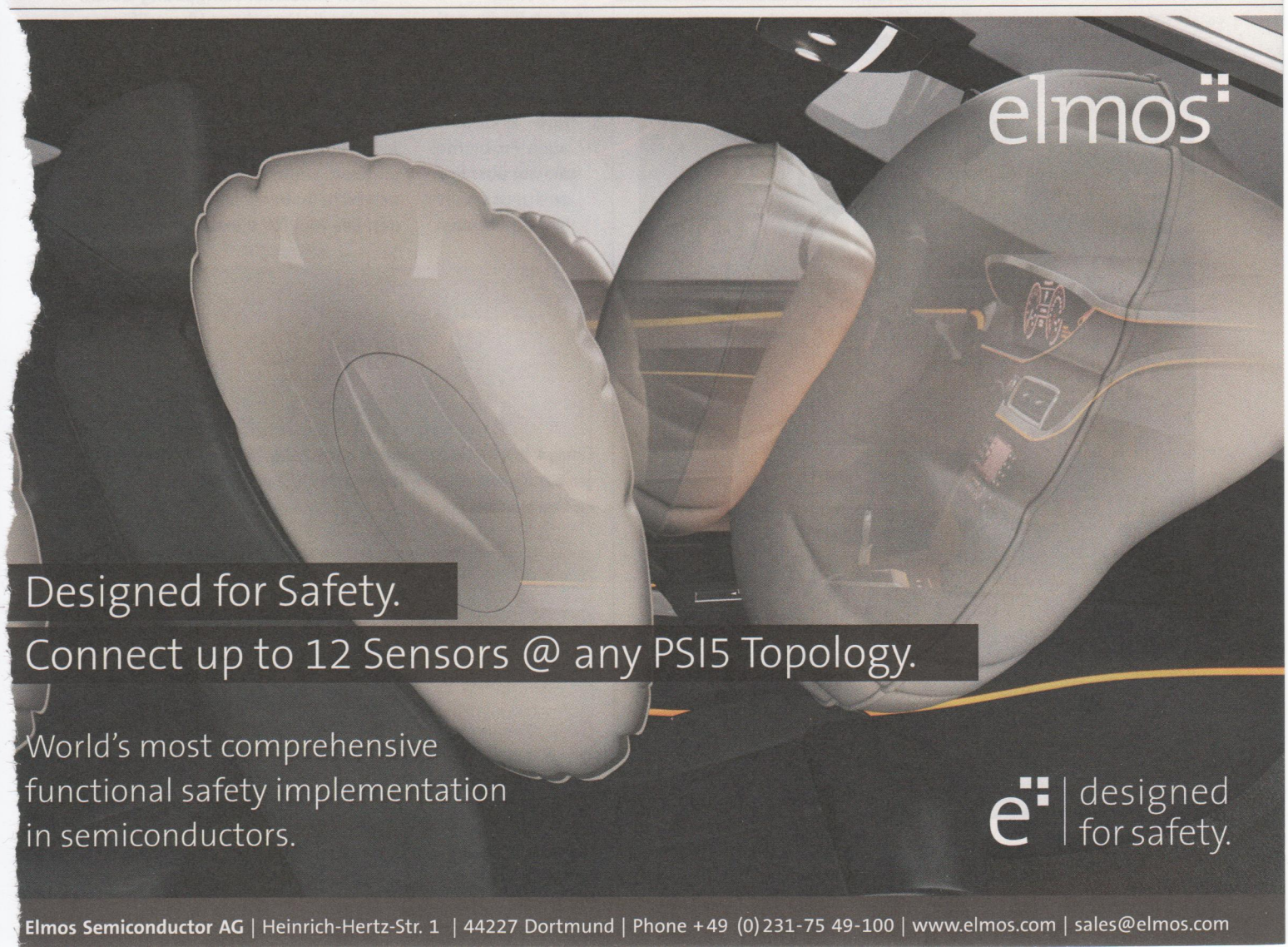
Für Codierungsstandards gebe ich zwei Empfehlungen. Erstens sollte man sich mit einigen der oben genannten Standards vertraut machen. Sie bieten Hintergrundinformationen und wecken das Interesse. Dann sollte man sich den zur Verfügung stehenden Tools widmen.

```
line_a |= 256;          /* set bit 8 */
line_b |= 128;          /* set bit 7 */
line_c |= 064;          /* set bit 6 */
```

Listing 1. Welches Problem steckt wohl in diesen drei Zeilen Code?

```
int round_to_nearest(float num)
{
    if (num >= 0.0) {
        return (int)(num + 0.5);
    } else {
        return (int)(num - 0.5);
    }
}
```

Listing 2. Auch hier steckt ein Fehler drin. Der Code wird zwar ausgeführt, aber nicht so, wie sich der Programmierer das gedacht hat.



```
int write_it(int dest /*fd*/, uintptr_t srcAddr, size_t len)
{
     unsigned char *buf = (unsigned char *)srcAddr;
     int ret;

     while (len > 0 && (ret = my_write(dest, buf,
                                       len)) > 0)
     {
          buf += ret;
          len -= ret;
     }
     return ret;
}
```

Listing 3. Hier macht „len" ein Problem. Welches, das erkennt z.B. eine statische Analyse.

Einige sind kostenlos erhältlich, andere kosten hunderte bis tausende Euro. Die Tools sollten so konfiguriert werden, dass man die Regeln so wählen kann, wie sie Sinn machen. Beginnen Sie dort, wo Sie können, und erweitern Sie Ihr Wissen im Laufe der Zeit.

Statische Analyse

Während Codierungsstandards den Code aus syntaktischer Sicht betrachten, arbeitet statische Analyse während der Compilierung oder Build-Phase. Sie simuliert die Auswirkungen, die sich bei der Codeausführung ergeben. Als Beispiel, was eine statische Analyse erkennen kann, dient der Code von **Listing 3**. Erkennen Sie das Problem? – Die Lösung: Ist „len" kleiner oder gleich Null, wird der Rückgabewert von „ret" nicht initialisiert.

Die statische Analyse bietet gegenüber Codierungsstandards eine Reihe von Vorteilen:

- Statische Analyse verbietet keine Codierungskonstrukte. Man kann den bestehenden Code nutzen.
- Die statische Analyse weist auf Fehler hin, die wahrscheinlich in der Praxis entstehen, während Codierungsstandards viele Änderungen erfordern, wo eigentlich gar kein Problem im Code vorlag.
- Tools für die statische Code-Analyse betrachten den Code global und erkennen auch Probleme, die nur über Prozedurgrenzen hinweg auftreten.

Die statische Analyse weist allerdings einige Einschränkungen auf.

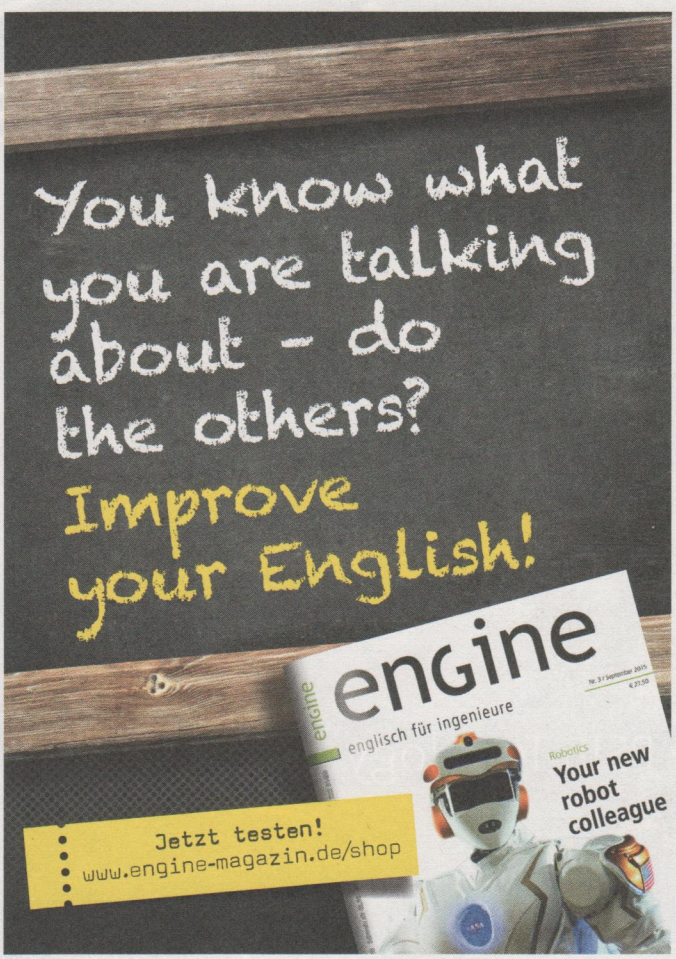
- Die Tools verwenden eine relativ begrenzte Zahl an Regeln, die sie prüfen. Viele Fehlerquellen werden durch diese Regeln nicht abgedeckt.
- Während ein False Positive laut Codierungsstandard ein Umschreiben des Codes nahe legt, was die Diagnostik beendet, kann ein False Positive für die statische Analyse umständlicher zu bearbeiten sein.

Automatische Laufzeitfehler-Überprüfung

Die statische Analyse wird „statisch" genannt, weil sie während der Compilierung zum Einsatz kommt. Die automatische Laufzeitfehler-Überprüfung (RTEC; Run-Time Error Checking) arbeitet völlig anders. RTEC sucht nach bestimmten Problemen während der Codeausführung. RTEC lässt sich auf verschiedene Arten durchführen:

- Ein Compiler kann die Überprüfungen automatisch hinzufügen.
- Nicht instrumentierter Code kann in einer Simulationsumgebung laufen, in der die Umgebung Überprüfungen durchführt. Das Open-Source-Projekt „Valgrind" ist ein Beispiel dafür.

Den Unterschied zwischen statischer Analyse und RTEC beschreibt **Listing 4**: Jeder Aufruf von get_and_set1() wird ungültig, wenn das „index"-Argument sich nicht im Bereich von 0 bis 4 befindet. Ein Tool für die statische Analyse




```
int array[5];
int get_and_set1(int index, int value)
{
     int ret = array[index];
     array[index] = value;
     return ret;
}
```

Listing 4. Hier wird nicht überprüft, ob das Index-Argument sich im zulässigen Wertebereich zwischen 0 und 4 befindet. Statische Analyse wird dieses Problem nicht melden.

```
int array[5];
int get_and_set1(int index, int value)
{
     int ret;
     if (index < 0 || index > 4) {
          report_error();
     }
     ret = array[index];
     array[index] = value;
     return ret;
}
```

Listing 5. Die Laufzeit-Überprüfung behandelt den Code von Listing 4 so, als wäre er so wie hier geschrieben, und findet unzulässige Index-Argumente.

```
// Die folgende Funktion setzt voraus, dass ein Zeiger
// und ein int die gleiche Größe aufweisen.
static_assert(sizeof(int) == sizeof(void *), "");
void do_sketchy_pointer_arithmetic(void)
{
    // ...
```

Listing 6. Eine statische Assertion ist eine Assertion, die während der Compilierung überprüft werden muss. Sie kann für defensive Programmierung und explizite Annahmen im Code verwendet werden.

```
static_assert(sizeof(header) <= 64, "header");      // ... OK
int myvar;
static_assert(myvar == 0, "myvar");                 // ERROR
```

Listing 7. Eine statische Assertion kann nur zur Überprüfung der Compilierzeit-Konstanten verwendet werden.

```
#include <assert.h>
err_t discrete_fft(data_t *p, size_t nelem)
{
    assert(nelem > 64);
    // ...
```

Listing 8. Laufzeit-Assertions können Bedingungen überprüfen, die während der Compilierung nicht evaluiert werden können.

wird kein Problem melden, solange es sich nicht sicher ist, dass ein Wert außerhalb dieses Bereichs genutzt wird. RTEC rätselt nicht. Es behandelt den Code so, als wäre er wie in **Listing 5** geschrieben.

RTEC hat den Vorteil, Fehler zu erkennen, die bei der Compilierung nicht erkannt werden. Andererseits erfordert es wesentlich größere Laufzeitressourcen, während die statische Analyse nur während der Compilierung läuft.

Assertions

Ein weiterer Garant für zuverlässige Software ist der regelmäßige Einsatz von Assertions. Eine statische Assertion ist eine, die während der Compilierung überprüft werden muss. Sie kann für defensive Programmierung und explizite Annahmen im Code verwendet werden (**Listing 6**). Eine statische Assertion kann nur zur Überprüfung der Compilierzeit-Konstanten verwendet werden. **Listing 7** zeigt ein Beispiel.

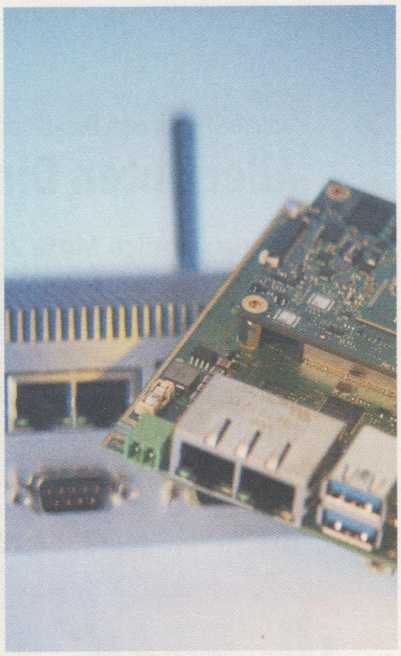
Statische Assertions sollten genauso wie Kommentare regelmäßig eingesetzt werden. Der eigentliche Vorteil einer statischen Assertion ergibt sich, wenn sich etwas ändert und man über einen Compiler-Fehler benachrichtigt wird. Damit entfällt die zeitaufwändige Fehlerfindung mittels Debugger. Statische Assertions werden vollständig während der Compilierung evaluiert, sodass sie die Leistungsfähigkeit des Systems nicht beeinträchtigen.

Laufzeit-Assertions können Bedingungen überprüfen, die während der Compilierung nicht evaluiert werden können. Die Header-Datei <assert.h> in **Listing 8** ist ein Beispiel dafür. Diese Art von Laufzeit-Assertion erzeugt Code, der die Bedingung während der Programmausführung überprüft. Assertions werden während der Entwicklung aktiviert, um möglichst viele Probleme aufzufinden. In der Testphase werden die Laufzeit-Assertions deaktiviert, damit das Produkt normal schnell läuft. Statische Asserts können jedoch während der Serien-Builds aktiv bleiben, da sie den ausführbaren Code nicht beeinflussen.

Hacker werden auch weiterhin nach Schwachstellen im Code suchen, die sie ausnutzen können, um Systeme unter ihre Kontrolle zu bringen. Werden die hier beschriebenen Standards, Tools und Techniken angewendet, lässt sich dieses Risiko mindern. *jk*

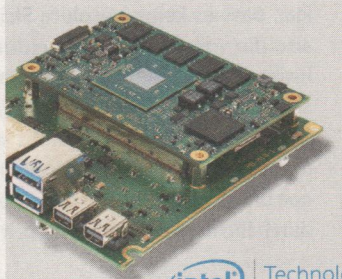
Greg Davis
leitet die Abteilung für die Entwicklung von Compilern bei Green Hills. Er hat sich intensiv mit Optimierung von Code, Schwächen von Programmiersprachen, Code-Generierung und den Erweiterungen für die Embedded-Entwicklung befasst. Er graduierte 1995 am California Institute of Technology.
mktg-europe@ghs.com

Halle 4, Stand 325



Sie haben die Wahl und wir haben die Lösung

- COM Express™ Module und Mainboards
- Hardwarekits und ODM Plattformen
- Entwicklung und Konstruktion
- Fertigung, Test und Montage
- Obsolesence Management für langfristige Verfügbarkeit



intel Technology Provider Platinum 2016

Skalierbar in Embedded:

- Intel® Atom™
- Intel® Core™ i7 / i5 / i3
- Intel® Xeon®

embeddedworld2016 Exhibition&Conference ...it's a smarter world
>> Hall 1, Stand 578

TQ-Group I Tel. 08153 9308-0
Mühlstraße 2 I 82229 Seefeld
info@tq-group.com
www.tq-group.com/intel

Technologie in Qualität