AUFGABE 4: ERWEITERTE ARITHMETISCHE CODIERUNG

In dieser Aufgabe werden Sie die gegen Bitfehler abgesicherten Bereiche Ihrer dreifach redundanten Filterausführung durch den Einsatz von arithmetischer Codierung erweitern. Die tatsächliche Fehlertoleranz Ihrer Implementierung soll anschließend durch systematisches Injizieren von Einbitfehlern überprüft, bewertet und verbessert werden.

Die Vorgabe befindet sich im Ordner `04_EAN` des Vorgaben-Repositories:

`https://gitlab.cs.fau.de/ezs/vezs19-vorgabe.git`

Starten Sie die Anwendung mit `make run` im Build-Verzeichnis, nachdem sie mittels

```
source ./ecosenv.sh && mkdir build && cd build && cmake ..
```

dieses erstellt haben.

Zusätzlich zu den gewohnten Befehlen können Sie nun mit `make trace` und `make inject` eine Fehlerinjektion durchführen. Eine Kurzübersicht wird anschließend in die Datei `ean_result.csv` geschrieben. Die Ergebnisse Ihrer jeweils letzten Injektion können Sie mittels `make resultbrowser` und einem Webbrowser genauer untersuchen.

Hinweis: Die Fehlerinjektionen dauern sehr lange! Fangen Sie frühzeitig mit der Bearbeitung an, und überdenken Sie Ihre Lösung genau, bevor Sie eine Injektion auslösen. Um die Injektionszeit zu reduzieren, können Sie Lösungsansätze in Isolation in `app_playground.cpp` prototypisch erproben und mittels `make trace_playground` und `make inject_playground` evaluieren.

1 Aufgabenstellung

Vorbereitung

Aufgabe 1 *Datenbank einrichten*

Kopieren Sie die Datei `my.cnf` als versteckte Datei in Ihr Home-Verzeichnis. Passen Sie die Datei mit den in der Tafelübung ausgeteilten Zugangsdaten Ihrer Gruppendatenbank an. Beachten Sie, dass immer nur *eine Injektion* pro Gruppe gleichzeitig stattfinden kann.

☞ `~/.my.cnf`

Aufgabe 2 *Anpassungen für die Ausführung mit Fail*

Leider ist die Injektion der Betriebssystemfunktionen, die wir in Aufgabe *03_TMR* zur Isolation der einzelnen Replikate und Ausführungseinheiten verwendet haben, zu aufwändig, um interaktiv daran zu arbeiten. Übertragen Sie deshalb Ihre TMR-Implementierung in das Vorgabeverzeichnis dieser Aufgabe. Bilden Sie Sensorabfrage, Filterung und Maskierer diesmal nicht auf Fäden, sondern auf Funktionen innerhalb der Funktion `cyg_user_start` ab. Da wir nur eine Ausführungsrunde mit Bitfehlern injizieren werden, genügt es, wenn Sie auch nur eine Runde implementieren. *Welche Fehler können Sie durch diese Einschränkungen nicht mehr maskieren oder erkennen?*

Um konsistentes Verhalten zu gewährleisten, stellen Sie sicher, dass sie als Filterparameter die in Aufgabenblatt 02 vorgegebenen Werte `1, 2, -1`.

Antwort:

Referenz

Aufgabe 3 *Erste Injektion*

Bestimmen Sie den Bereich Ihrer Applikation, den Sie mit TMR gegen Bitkipper abgesichert haben. Stellen Sie sicher, dass die Injektion direkt nach dem Auslesen und Skalieren der Inputwerte (also noch vor deren Kodierung) beginnt und nach der Einigung auf den Ausgabewert endet. Markieren Sie diese Stellen mit den in der Tafelübung besprochenen Markern. Setzen Sie auch POSITVE, NEGATIVE und DETECTED Marker gemäß ihrer Bedeutung in Ihrem Voter ein. Um den NEGATIVE Marker sinnvoll aufrufen zu können, benötigen Sie ein korrektes Ergebnis des Filterschrittes den Sie injizieren. Nutzen Sie entweder den Debugger oder (unschöner) **`printf`**`()` um diesen Wert zu erfahren. Prüfen Sie in allen weiteren Aufgaben das Ergebnis Ihrer Filterung gegen diesen Wert. Die Überprüfung Ihres durch den Ausgangsvoter gefundenen Wertes kann außerhalb des injizierten Codes stattfinden. Setzen Sie aber auch innerhalb des injizierten Codes die Marker, falls passend. Wenn Sie so alle Marker gesetzt haben, können Sie Ihre erste Fehlerinjektion durchführen. Nutzen Sie den Resultbrowser um sich die häufigsten Fehlerquellen anzusehen.
Was sind die häufigsten Fehlerquellen?

☞ `fail_start_trace`

☞ `fail_stop_trace`

☞ `fail_marker_{positive, negative}`

☞ `fail_marker_detected`

☞ `make ddd`

☞ `make gdb`

☞ `make trace`

☞ `make inject`

☞ `make resultbrowser`

Antwort:

Aufgabe 4 *Ungesichertes System*

Sichern Sie Ihren Stand und vergleichen Sie Ihr Ergebnisse mit einem nicht-replizierten Filterlauf. Achten Sie darauf nicht nur die redundanten Filterschritte auszukommentieren, sondern auch den Voter entsprechend anzupassen. *Wie viel sicherer ist Ihre initiale TMR Variante im Gegensatz zu Ihrer ungesicherten Version?*

☞ `git checkout -b` und `git commit`

Antwort:

Aufgabe 5

Entspricht dies Ihren Erwartungen?

Antwort:

Aufgabe 6

Stellen Sie Ihre replizierte Implementierung wieder her.

☞ `git checkout` oder `git revert`

Kombinierter Ansatz

Aufgabe 7 *Codierung*

An welchen Stellen Ihrer Anwendung ist die strukturelle Redundanz nicht mehr gegeben?

Antwort:

Aufgabe 8

Sichern Sie diese Stellen durch den Einsatz von ANB-Codes ab. Nutzen Sie die vorgegeben Funktionen um diese Stellen zu schützen. Wählen Sie die Signaturen so, dass Sie keine Überlaufe bei der Berechnung der Signaturen erhalten. Beachten Sie: Da wir nur Ein-Bit-Fehler injizieren, hat die Wahl der Konstanten keinen Einfluss auf die Fehlertoleranz Ihrer Lösung. Vermeiden Sie Berechnungen mit kodierten Werten falls strukturelle Redundanz vorliegt und de- bzw- enkodieren Sie immer nur an den Übergängen. *Weshalb ist die equals-„Funktion“ als Präprozessor-Makro und nicht als C-Funktion vorgegeben?* Achten Sie darauf, statische Werte zur Compile-Zeit vom Compiler berechnen zu lassen und dass dynamische Berechnungen nicht wegoptimiert werden.

Antwort:

Aufgabe 9 *CoRed-Voter*

Implementieren Sie den in der Tafelübung besprochenen CoRed-Voter innerhalb des injizierten Bereichs. Vergessen Sie nicht die Überprüfung des berechneten Wertes und der *dynamischen Sprungsignaturen* im nicht-injizierten Bereich. Achten Sie auch auf das Setzen der jeweiligen Marker.

☞ Kontrollfluss

Optimierung

Aufgabe 10 *Programmiermuster*

Injizieren Sie Ihre Lösung und betrachten Sie die Fehler im Resultbrowser. Nutzen Sie Ihre Erkenntnisse um die Lösung iterativ besser zu machen. *Welche Programmiermuster sind besonders anfällig?*

Antwort:

Aufgabe 11

Minimieren Sie das Auftreten des NEGATIVE-Markers weiter. Versuchen Sie, die absolute Fehlergesamtzahl (*Total*, also die Summe der Injektionen *IP, MEM, REGS*, in der Ausgabe, *Full Injection* im Dashboard) unter 1000 zu bringen. Fühlen Sie sich frei, sowohl Funktionssignaturen als auch vorgegebene Speicherstellen zu verändern. Die Gruppe, die am Besten abschneidet, erhält eine kleine Belohnung.

☞ `fail_marker_negative`

2 Erweiterte Aufgabe

Kopieren Sie Ihre aktuelle Implementierung nach `src/app_ext.cpp` und bearbeiten Sie hier die folgenden Aufgaben. Alle `make`-Ziele verhalten sich wie in der grundlegenden Übung, benötigen jedoch noch ein ein *_ext* Suffix.

☞ `make trace_ext`

☞ `make inject_ext`

☞ `make resultbrowser_ext`

Aufgabe 12 *Reduktion der DETECTED-Marker*

Beim Auftreten eines DETECTED-Markers ist die Wahrscheinlichkeit sehr hoch, dass dennoch ein valider Wert berechnet wurde und identifiziert werden kann. *Warum? Welchen Einfluss hat die Tatsache, dass unsere Fehlerhypothese nur von einmaligen Einbitfehlern ausgeht auf Ihre Überlegung?* Nutzen Sie diese Überlegung um den Fehlerfall nicht nur zu erkennen sondern auch – innerhalb des injizierten Bereichs – zu maskieren.

Antwort:

Aufgabe 13 *Interne Zustände der Replikate*

Sichern Sie abschließend auch den internen Zustand der einzelnen Replikate ab. Berechnen Sie hierfür bei jedem Filterschritt eine Prüfsumme über den internen Zustand der Replikate und führen Sie einen Test auf Gleichheit durch. Da dieser Test nicht redundant implementiert werden kann, müssen auch diese Prüfsummen kodiert werden. *Warum ist eine solche erweiterte Fehlererkennung sinnvoll?*

Antwort:

Hinweise

- Bearbeitung: Gruppenarbeit
- Abgabefrist: 18.06.2019
- Fragen bitte an `i4ezs@lists.cs.fau.de`