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

```
git@gitlab.cs.fau.de:ezs/vezs22-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.

1 Aufgabenstellung

Vermerken Sie Ihre Antworten zu den Fragen der einzelnen Aufgaben an den vorgesehenen Stellen in der vorgegebenen `answers.md`*. Bitte erstellen Sie, um die Abgabe durch Mergerequests zu vereinfachen,* ***pro Aufgabe einen eigenen Branch****. Um einen konsistenten Zustand zu gewährleisten, benutzen Sie dazu bitte folgenden Befehl:*
`git fetch git@gitlab.cs.fau.de:ezs/vezs22-vorgabe.git aufgabe4 && git checkout -b aufgabe4 FETCH_HEAD`

Aufgabe 1 *Definition des Fehlerraumes*

In dieser Aufgabe fokussieren wir uns auf durch transiente Fehler verursachtes Fehlverhalten. *Woraus setzt sich der Fehlerraum eines solchen Systems unter diesen Bedingungen zusammen? Wie kann dieser für unser System (betrachten Sie vornehmlich Systemkomponenten von der Prozessorebene aufwärts) bestimmt werden?*

Antwort:

Referenz

Aufgabe 2 *Basissystem*

Die Untersuchung einer Fehlertoleranzmaßnahme kann nur differentiell erfolgen: Durch den Vergleich einer ungehärteten mit einer gehärteten Variante. In dieser Teilaufgabe schaffen Sie zunächst diese Vergleichsgrundlage. Übernehmen Sie hierzu Ihre Implementierung der Ihnen bekannten Signalverarbeitungskette aus dem vorherigen Aufgabenblatt in die Datei `app_tmr.c`. Kopieren Sie diese Lösung ferner in die Datei `app_baseline.c` und entfernen Sie dort sämtliche Redundanzmaßnahmen. Achten Sie dabei auf eine möglichst einfache Implementierung, die dennoch die gleiche Funktionalität erfüllt. Stellen Sie dies durch Vergleich der Ergebnisse der ersten Filterschritte sicher. *Auf welche Abläufe und Betriebssystemmechanismen kann in der „Baseline"-Version verzichtet werden?*

Antwort:

Aufgabe 3 *Laufzeitmessung*

Unter der Annahme einer konstanten Fehlerrate steigt die Fehleranzahl mit zunehmender Laufzeit. Deshalb ist die Laufzeit eine Komponente des (potentiellen) Fehlerraumes. In der Vorgabe findet sich bereits eine Schnittstelle zum Abrufen der Hardwareuhr (des Simulators). Nutzen Sie diese Funktionalität, um die Ausführungszeit einer einzelnen Ausführung der Signalverarbeitungskette zu vermessen. *Was ist bei der Umsetzung der Messung zu beachten? Bedenken Sie dabei den Einsatzzweck: Die approximative Bestimmung eines Fehlerraums. Wie verhalten sich die beiden Implementierungsvarianten im Vergleich zueinander?*

☞ `ezs_counter_get()`
☞ `ezs_counter_resolution_us`

Hinweis: Aufgrund des internen Aufbaus des hier verwendeten Simulators „QEMU" kommt es während der ersten Ausführungen der Signalverarbeitungen zu Verzögerungen, welche sich in Form sehr kurzer Antwortzeiten in den Messergebnissen widerspiegeln. Bitte berücksichtigen Sie diesen Umstand in Ihrem Messaufbau (bspw. durch Verwerfen dieser Werte oder statistische Methoden). Bitte beachten Sie ferner, dass es sich hierbei um einen reinen Lehraufbau handelt, der ausreicht, um bestimmte Effekte zu demonstrieren. In der Praxis sollten solche Messungen möglichst auf der Referenzhardware, oder wenn nicht anders möglich zumindest

in zyklengenauen Simulatoren durchgeführt werden, da andernfalls der Simulator (wie hier bei Qemu) das Zeitverhalten verfälschen könnte.

Hinweis: Sollten Sie im Lauf der Aufgabenbearbeitung feststellen, dass die von uns vorgegebene Reglerperiode für ihr System zu gering ist, so dürfen Sie diese bei Bedarf selbstständig erhöhen. Beachten Sie, dass zu hohe Werte jedoch zu langen Messzeiten führen können.

Antwort:

Aufgabe 4 *Ort (I)*

Eine weitere Komponente des Fehlerraumes stellt der Ort des Fehlers da. Bestimmen Sie zunächst, wo dieser Parameter sich zwischen Ihren beiden Systemvarianten *nur für ihren Anwendungscode (ohne Betriebssystem) allein* unterscheidet.

Hinweis: Beachten Sie, dass nicht benötigte Funktionen selbstverständlich keine Angriffsfläche darstellen.

Hinweis: Zur Beantwortung dieser Frage sind möglicherweise die Programme `size(1)`, `nm(1)`, `objdump(1)` sowie ggf. `diff(1)` hilfreich. Ferner bieten wir Ihnen durch ein entsprechendes Target die Möglichkeit, einen Aufrufgraphen Ihres Programms zu erstellen. Die Objektdateien ihres Kompilats finden Sie nach dem Übersetzen in `build/CMakeFiles/{baseline,tmr,ean}.dir/src/.../*.o` (je nach Anwendungsvariante); die vollständig gebundene Anwendung (ELF) liegt jeweils unter `build/{baseline,tmr,ean}.elf`.

☞ `make callgraph_{baseline,tmr,ean}`

Hinweis: Da es im gebundenen ELF schwierig fällt, Daten eindeutig Betriebssystem bzw. Anwendung zuzuordnen, lohnt es sich für die Speicherplatzanalyse von Anwendungen die Objektdateien zu betrachten. Zur Anzeige der Aufrufbäume in den Targets `make callgraph_*` wird ferner eine laufende graphische Oberfläche benötigt.

☞ `remote.cip.cs.fau.de`, Xpra

Dokumentieren Sie unbedingt auch Ihr Vorgehen. Falls Sie selbst erstellte Skripte zur Analyse einsetzen, so können Sie diese auch als Dokumentation Ihres Vorgehens Ihrer Abgabe beifügen.

Antwort:

Aufgabe 5 *Ort (Gesamtsystem)*

Welchen Anteil steuert dabei die Anwendung jeweils zum gesamten Speicherbedarf des Systems/ELFs (d.h. inklusive Betriebssystem) bei? Interpretieren Sie diese Werte auch bezüglich Fehleranfälligkeit und der Planung potentieller Härtungsvorhaben.

☞ `size(1), {baseline,tmr}.elf`

Antwort:

Aufgabe 6 *Fehlerraum (I)*

Errechnen Sie unter der Annahme eines Einbitfehlers aus Ihren Werten aus den Aufgaben 3 und 4 rein kombinatorisch eine (zugegebenermaßen sehr krude) Abschätzung des Fehlerraumes für Ihre Anwendung. *Ordnen Sie Ihre Messwerte ein: Inwiefern korrespondieren die ermittelten Größen mit Ihrer tatsächlich vermuteten Zunahme des Fehlerraums? Woraus ergeben sich Fehler in dieser approximativen Abschätzung?*

Antwort:

Hinweis: Sichern Sie ihren aktuellen Zustand für Vergleichszwecke , und kopieren Sie ihre Lösung aus `app_tmr.c` nach `app_ean.c`. Arbeiten sie fortan auf dieser Datei.

☞ `git commit`

Kombinierter Ansatz

Aufgabe 7 *Redundanzbereich*

Wie Sie bereits im letzten Aufgabenblatt festgestellt haben, deckt der durch TMR aufgespannte Redundanzbereich Ihre Anwendung nicht vollständig ab. *An welchen Stellen Ihrer Anwendung ist die strukturelle Redundanz nicht mehr gegeben?*

Antwort:

Aufgabe 8 *Codierung*

Sichern Sie zumindest die Ausgangsseite Ihres Systems durch den Einsatz von ANB-Codes ab. Nutzen Sie die vorgegebenen 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.

Hinweis: Beachten sie für Ihre Implementierung den vorgegebenen Header `include/cored_vote.h` und die Funktionsrümpfe in der Datei `cored_vote.c`. Gegebenenfalls müssen Sie ferner auch Datentypen und Schnittstellen zwischen Replikaten und Entscheidern entsprechend anpassen.

Aufgabe 9 *CoRed-Voter*

Implementieren Sie den in der Übung besprochenen CoRed-Voter innerhalb der Funktion `CoRedVote`. Vergessen Sie nicht die Überprüfung des berechneten Wertes und der *dynamischen Sprungsignaturen* in der Funktion `perform_encoded_voting`. Markieren Sie durch passende Rückgabewerte im Feld `output_t.vote` fehlerhafte sowie erfolgreiche Berechnungen. Im Fehlerfall sind mögliche Reparaturen am Replikatszustand nur in dem auf dem letzten Aufgabenblatt geforderten Umfang nötig (Datenfehler eines einzelnen Replikats), andere Fehlerszenarien müssen lediglich toleriert und maskiert, jedoch nicht dauerhaft behoben werden.

Weshalb ist die equals-„Funktion“ (in der Datei `src/cored_vote.c`*) 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. ☞ objdump

Sie können mittels der Funktion `ezs_ean_test` die grundlegende Funktionalität ihrer Lösung überprüfen. Rufen Sie anschließend `perform_encoded_voting` an geeigneter Stelle in Ihrer bisherigen Implementierung auf.

Antwort:

Aufgabe 10 *Fehlerraum (II)*

Bestimmen Sie nun wieder wie schon in Teilaufgabe 6 eine Abschätzung des Fehlerraumes für diese Anwendungsvariante und vergleichen Sie auch hier wieder die Anwendungsvarianten. *Ordnen Sie die Messergebnisse ein.*

Antwort:

Aufgabe 11 *Analyse & Diskussion*

Mit der Einführung der Redundanzmaßnahmen ist der geschätzte Fehlerraum (hoffentlich auch in Ihrer Berechnung) beständig gestiegen. *Was muss für die Effektivität einer Maßnahme gelten, damit ihr Einsatz sinnvoll erscheint?*
Antwort:

Während regulärer Semester zeigen die Ergebnisse der Fehlerinjektionsexperimente häufig den folgenden Verlauf: Nach Einführung der TMR-Maßnahmen steigt die Fehleranfälligkeit des Systems zunächst an. Durch Codierung und begleitende Maßnahmen kann sie dann jedoch deutlich unter den Wert der ungehärteten Version gedrückt werden. *Begründen Sie dieses Verhalten. Ferner: Wieso ist der Einsatz von TMR in diesem Szenario trotz der initialen Zunahme sinnvoll?*

Antwort:

2 Erweiterte Aufgabe

Im Rahmen der erweiterten Übung soll sich nun der dynamischen Fehlerinjektion mittels des Emulators QEMU und des Debuggers gdb genährt werden. Die Untersuchungen sollen dabei auf der EAN-gehärteten Anwendungsvariante erfolgen.

Aufgabe 12 *Fehlerinjektion: Markerplatzierung*

Um die Auswirkungen eines Defektes gemäß seiner Auswirkungen auf das Gesamtsystem besser abschätzen zu können, müssen zunächst Stellen im Programmfluss identifiziert werden, an denen festgestellt werden kann, ob die Programmdurchführung erfolgreich war, der Defekt zumindest detektiert wurde oder es in der Ausführung unerkannt zu einem Fehlverhalten kam. Setzen Sie hierzu POSITVE, NEGATIVE und DETECTED Marker gemäß ihrer Bedeutung in ihr System ein. Um den NEGATIVE Marker sinnvoll aufrufen zu können, benötigen Sie ein korrektes Vergleichsergebnis des Filterschrittes den Sie injizieren. Stellen Sie deshalb deterministische Eingaben für ihre Filterung sicher und nutzen Sie einschließend den Debugger um diesen Wert für die ersten Durchläufe zu erfahren.

☞ `fail_marker_{positive, negative}`

☞ `fail_marker_detected`

☞ `make debug_ean`
`make gdb_ean`

Hinweis: Es ist akzeptabel, wenn ihr Experiment während der Injektion nur eine begrenzte Anzahl an Durchläufen (d.h. Paare aus Eingaben und überprüften Ausgaben) unterstützt und sich danach kontrolliert beendet.

Aufgabe 13 *Fehlerinjektion: Eingangsdaten*

Nehmen Sie nun eine erste händische Fehlerinjektion vor. Identifizieren Sie hierzu eine geeignete Stelle im Programmfluss um den Eingabewert des ersten Replikats zu modifizieren. Nehmen Sie dort **mittels Debugger** eine Fehlerinjektion vor. Toggeln

Sie das niederwertigste Bit. Beobachten Sie den weiteren Verlauf (step/stepi, später dann continue). Welchen Ihrer Marker wird final erreicht? Ist dies zu erwarten?

Antwort:

Aufgabe 14 *Fehlerinjektion (II)*

Suchen Sie nun eine Stelle im Programmablauf, in der ein einzelner Einbitfehler ein Systemversagen herbeiführen kann. Wo lässt sich eine solche Stelle finden? Führen Sie auch hier eine Fehlerinjektion durch und überprüfen Sie so Ihre Hypothese.

Antwort:

Aufgabe 15 *Untersuchung der Maschinencodeebene*

Untersuchen Sie nun durch Disassemblierung die Funktion `convolve_filter_step`. Wie werden hier auf die einzelnen verwendeten Variablen/Daten zugegriffen?

Antwort:

Aufgabe 16 *Register auf IA32*

Auf der IA32-Plattform nehmen einige Register aus der Gruppe der General-Purpose Register (von uns in dieser Aufgabe betrachtete GP-Register: `eax`, `ebx`, `ecx`, `edx`, `esp`, `ebp`, `esi`, `edi`) somit eine Sonderrolle ein und werden nicht als bloße Datenregister genutzt. Welche sind dies? Wie werden sie im in Aufgabe 15 untersuchten Kompilat verwendet?

Hinweis: Für Spezialisten mit Hintergrundkenntnissen: Wir bzw. gcc verwenden hier die SysV-ABI auf der IA32-Architektur.

Antwort:

Aufgabe 17 *Fehlerinjektion: Registerindirekte Zugriffe*

Am Ende der Funktion (immer noch `convolve_filter_step`) wird das Register `esp` wieder auf seinen ursprünglichen Wert beim Betreten der Funktion zurückgesetzt. Dies erfolgt in der Regel mittels des Opcodes `leave`, welcher eine Abkürzung zur Befehlsfolge `mov %ebp, %esp; pop %ebp` darstellt. Identifizieren Sie die zugehörige Instruktion in der Assembly. Verfolgen Sie zunächst eine fehlerfrei Ausführung per Singlestepping. Injizieren Sie nun vor Ausführung dieser Instruktion einen Einbitfehler in die 32bit breiten Datenworte oberhalb der von `esp` beschriebenen Adresse (sprich: Das 32bit breiten Wort (oder da little-endian eben auch einfach "Byte") an der Adresse `$esp + 4` im Debugger). Toggeln Sie dafür jeweils einzeln die untersten vier Bits (insgesamt also 4 einzelne Injektionsexperimente). Verfolgen Sie durch Singlestepping den weiteren Verlauf der jeweiligen Ausführungen. Was passiert hier im Folgenden? Was ist das Endresultat dieser sowie der folgenden Berechnungs/Votingrunde? Welche Bedeutung tragen die injizierten Daten? Erklären Sie den Effekt. Wieso ist diese Fehlerklasse als (möglicherweise) besonders problematisch anzusehen?

Bonusfrage für parallele/ehemalige BS(T)-Hörer und Betriebssystemspezialisten: Betrachten Sie auch die Ausgabe von `info registers all`. Sehen Sie hier ähnlich problematische Register oder erinnern Sie sich an andere problematische Datenstrukturen im Speicher?

Antwort:

Aufgabe 18 *Fehlerinjektion: Registerindirekte Zugriffe (II)*

Injizieren Sie *zu Beginn* der Funktion (vor dem Funktionsprolog) `convolve_filter_step` nun in das Register `esp` einen Einbitfehler. Toggeln Sie dafür jeweils einzeln die untersten vier Bit des Registers (insgesamt also 4 getrennte Injektionsexperimente). Verfolgen Sie durch Singlestepping den weiteren Verlauf der jeweiligen Ausführungen dieser Funktion. Welchen Effekt hat der Fehler auf die in der Rechnung verwendeten Werte?

Antwort:

Aufgabe 19 *Härtung*

Die Anfälligkeit einzelner Komponenten gegen transiente Fehler ist sehr stark von der zugrundeliegenden Speichertechnologie, sowie ihrer Strukturbreite abhängig. In unserem System gehen wir nun von der folgenden, gängigen, Anordnung aus (von zuverlässiger hin zu weniger zuverlässig):

Konstanten/Programmcode in Read-Only Memory > Register (bspw. FlipFlops) » Arbeitsspeicher (bspw. DRAM)

Zunächst: Oben wurden unter Anderem das Register ebp injiziert. Heißt das, dass diese Form von Fehlverhalten in unserem hier angedachten System eher unwahrscheinlich ist?

Leiten Sie daraus, in Verbindung mit Ihren Betrachtungen oben, Empfehlungen ab, wie hier durch Umstrukturierungen am Code und/oder im Übersetzungsprozess die oben beschriebenen Probleme abgemildert werden könnten. Welche Maßnahmen würden Sie an Ihrer Lösung vornehmen (Implementierung nicht erforderlich)?

Antwort:

Hinweise

- Bearbeitung: Gruppenarbeit
- Abgabefrist: 06.12.2022
- Fragen bitte an i4ezs@lists.cs.fau.de