Slice Your Bug:
Debugging Error Detection Mechanisms using Error Injection Slicing

Ute Schiffel, André Schmitt, Martin Süßkraut and Christof Fetzer
• Error Injection with Slicing (EIS):

• Software based, hardware independent error injector

• Triggers symptoms of hardware errors during the application runtime

• Uses an flexible and comprehensive error model

• Debugging support by creating a forward slice of modified data and control flow to understand effects of injected errors
• Expected exponential growth of soft error rate + faster aging of transistors [1]

• Systems Engineering group at TU-Dresden tries to build dependable systems (=trustworthy systems)

• Mission critical systems must be able to detect and handle transient and permanent hardware errors as bitflips, stuck bits in memory, bus, processor, …
  – Fault model can be very complex
  – Large number of possible error sources in hardware and software

• We try is to build systems that can detect and handle hardware errors using unreliable/commodity hardware

Software Encoded Processing

- C\C++ programs are automatically transformed into a safe version which is with a very high probability not vulnerable to hardware execution errors
- SEP targets mission-critical systems
- Data is continuously protected by Arithmetic Codes during storage, transportation, and processing
- Many possible fields of application: automotive, avionics, ...

- We needed a flexible and fast testing framework for evaluation.
Injection Tools

- **Physical injection**: using heat, radiation, heavy ions, inject into circuit pins
  - realistic, but expensive, difficult to control, reproduce, and to debug

- **Simulation based**: simulate hardware - processor and memory with e.g. stuck-at-0/1, bridging and timing errors
  - Realistic, but high overhead, high masking[2], hardware dependent

- **Symbolic injection**: complete coverage based on system and error model
  - Detailed, but complex, runtime increases very fast with larger models

- **SIFI**: inject data or control flow errors directly
  - Symptom based injection, dynamic/static instrumentation

Software Implemented Fault Injection

• Less masking of errors then with direct hardware injection
  – Inject symptoms of hardware errors

• Hardware independence
  – No special test hardware needed, error model is hardware independent

• Good controllability and reproducibility of injected errors
  – Combine different types, complete deterministic injection

• Fast and easy of use
  – Transform and compile application once, trigger individual errors during runtime

• Slicing: Important debugging feature to understand the effects of injected errors e.g. for undetected silent data corruption
Error Model

• Interested in hardware errors
• The error model was first used by Forin[3] for safety evaluation of encoded programs

• **Modify operands**: bitflip(s) in operands
• **Faulty operation**: bitflip(s) in operation result
• **Exchange operands**: use different but unmodified data
• **Exchange operators**: use different operator (+ → -)
• **Lost Stores**: Store operation in omitted

Injection Tool

- Implemented as compiler passes, instrumentation during compilation
- Error injection only possible into available source code
- Uses LLVM framework (= ease of use and completeness)
- Insert error trigger points into binary
- Which inserted error point is triggered at what point of the execution is decided during runtime
Execution Modes

- **Deterministic**: activate exactly one error point per run

- **Probabilistic**: combination of all error models, each error point is triggered with given probability
  - Likelihood of error activation depends on the applications runtime
  - Multiple errors can be triggered

- **Permanent**: activated error depends on the original value
  - Simulate permanent hardware errors
  - If specific bit is set in original value it is flipped
  - Implemented for arithmetic integer operations and integer loads/stores
Analysis Motivation

- Goal is to evaluate several error detection techniques
  - e.g.: Encoded Processing, Swift
- Interesting are errors that are **not masked** but also **not detected** → silent data corruption

- Filtered by comparing injected programs output with output from golden run
  - Output is different but error not detected

- Result: Some (few) errors were not detected.
- Effects of error are hard to understand if you just look at the code.
- We needed an automated approach to help us understand what happened.
Analysis

• For error $e$ EIS computes the slice $S_e$ containing all instructions operating on values influenced by $e$

• Slice contains only data influenced by the error and modified control flow $\rightarrow$ much smaller than complete trace

• Slice starts with error injection and end with external function e.g. printf()

• Slice is presented as browse-able XML or HTML document

• Goal: let us find missing checks and improve coverage of error detection
Example Slice

• Slice consists of sequence of LLVM instruction
• Effected registers are marked and their values displayed (omitted here)
• bitflip_i32 is the error injection point triggered
Example Slice

1  [+] @abs:
2  [+] bb1:
3      %reg1 = call i32 @bitflip_i32 (i32 %tmp0)
4      store i32 %reg1, i32* %addr1
5
6  [+] bb2:
7      %reg2 = load i32* %addr1
8      store i32 %reg2, i32* %addr2
9
10  [+] return:
11     %reg3 = load i32* %addr2
12     ret %reg3
13
14  [+] @main:
15  [+] bb4:
16      %reg1 = call i32 @abs (i32 %tmp0)
17      %reg2 = add i32 %reg1, %tmp1
18      store i32 %reg2, i32* %addr3
19      ;; loop start (repeated 350 times)
20     %reg3 = load i32* %addr3
21     %reg4 = add i32 %tmp2, %reg3
22     store i32 %reg4, i32* %addr3
23      ;; loop end
24
25  [+] bb15:
26      %reg5 = load i32* %addr3
27      call void @print_3_int(i8* %msg, i32 %tmp3, i32 %reg5, i32 %tmp4)
Injection Tool + Slice

- Insert shadow instructions to trace error
  - Shadow LLVM registers with shadow register
  - Instruction value are disjunction of operands
  - Function get additional return value and additional parameter
  - Memory with Shadow memory
Summary

• Slicing is faster than full log because it reduces IO
• Slice is much smaller than full trace because it omits unmodified values

• Fast, easy and cost effective error injection tool
• We found slicing very helpful for understanding the results of error injection and evaluating/improve the protection mechanism
Error Model

- Not limited to data segment, support for multiple bitflips
- Combination of injected symptoms implicitly include other possible symptoms of hardware errors
  - Exchanged operator + exchanged operands = control flow error

- Model is extensible:
  - Software errors e.g. out-of-memory errors = ~100 lines of additional code
Deterministic Reexecution

• During execution multiple encounters of the same error point are possible (loops)
• Each encountered error point increments global counter
• Injection log contains this counter value, the unique id of the triggered error together with the error type and the original and modified values
• Parameters needed for re-execution of injection with slicing