

Symbolic Fault Injection

Reiner Hähnle & Daniel Larsson

KeY Symposium
Speyer, June 2006

Symbolic Fault Injection

An attempt to introduce Formal Methods into the area of Fault Injection

Objective: Evaluate dependability of (safety-critical) computer systems

Outline

- Fault Tolerance
- Fault Injection
- Weaknesses of Conventional Fault Injection
- Symbolic Fault Injection
- Case Study: CRC
- Limitations
- Future Work

Fault Tolerance

Impossible guarantee that computer system is error free

Even when using best available techniques for

- design of hardware/software
- manufacturing hardware components
- testing
- ...

the system may still contain defects.

Moreover: Cannot guarantee absence of transient errors:

- electromagnetic interference
- ionizing radiation
- ...

Fault Tolerance

How achieve fault tolerance?

- Some kind of redundancy, i.e. extra hardware and/or software used to build fault tolerance mechanisms:
 - fault detection
 - fault recovery
 - fault masking
- Symbolic fault injection:
 - A new method for evaluating fault tolerance mechanisms
 - Applicable to *software*-implemented mechanisms
 - ... that handle *hardware* faults (so far)

Fault Injection

A collection of techniques for

- deliberately introducing errors into a computer system
- examine the system's behavior in the presence of these errors

Hardware-implemented fault injection

- pin-level injection
- heavy ion radiation
- ...

Software-implemented fault injection

- Special software used to alter system state. Simulates the effects of real hardware/software faults.

Fault Injection

Fault injection techniques are based on a *fault model*, a specification of the type of faults to be injected. Example:

- Hardware faults
 - bit-flips (single/multiple)
 - “stuck-at” faults
 - ...
- Software faults
 - mutations: syntactically correct changes of original software
- Fault model for symbolic fault injection:
single and multiple bit-flips

Weaknesses of Conventional Fault Injection

Experimental evaluation

- Impossible to achieve 100% coverage
 - Example: Examine consequences for all states that result from 3 bit-flips in a 32-bit memory location
 - \Rightarrow 4960 test cases
 - Moreover: The results only hold for *one* particular input
- Large portion of injected faults are not “activated”. Are injected in
 - parts of memory that are not used
 - registers that are overwritten before they are read
- Time consuming analysis of huge amounts of monitoring (trace) data

What about a Symbolic Approach?

Can *complement*, not replace conventional methods

- Better coverage
 - symbolic input
 - symbolic faults
- More faults are activated
- Can in a natural way be combined with formal verification

Symbolic fault injection is based on *symbolic execution*

Symbolic Fault Injection

Idea:

- Inject *symbolic* faults (representing whole classes of “real” faults) during symbolic execution
- Continued execution shows the consequences of these faults

One step further:

- Combine this technique with formal verification
- Possible to prove that a program has certain properties *in the presence of faults*

Symbolic Fault Injection

How does symbolic fault injection work?

- Source code is instrumented with pseudo-instructions `inject(location);`
- KeY is extended with proper rules/taclets for these `inject` instructions

Rules for `inject`:

$$\text{boolean} \frac{\vdash \{b:=\text{true}\}\langle\omega\rangle\phi \quad \vdash \{b:=\text{false}\}\langle\omega\rangle\phi}{\vdash \langle\text{inject}(b); \omega\rangle\phi}$$

$$\text{int} \frac{\vdash \forall j : \text{int}. \{i := j\}\langle\omega\rangle\phi}{\vdash \langle\text{inject}(i); \omega\rangle\phi}$$

Symbolic Fault Injection

Example program:

```
int m(int x) {  
    int r;  
    if (x >= 0) {r = x;}  
    else {r = 0;}  
    return r;  
}
```

Property we want to prove:

m always returns a non-negative value

Symbolic Fault Injection

Example program:

```
int m(int x) {  
    int r;  
    if (x >= 0) {r = x;}  
    else {r = 0;}  
    inject(r);  
    return r;  
}
```

Property we want to prove:

m always returns a non-negative value

Symbolic Fault Injection

$$\vdash \langle \text{if}(x \geq 0) \{ r = x; \} \text{else} \{ r = 0; \} \text{ inject}(r); \text{return } r; \rangle \text{result} \geq 0$$

Symbolic Fault Injection

$$\begin{aligned} & x \geq 0 \vdash \langle r=x; \text{inject}(r); \text{return } r; \rangle \text{result} \geq 0 \\ & \vdash \langle \text{if}(x \geq 0) \{ r=x; \} \text{else} \{ r=0; \} \text{inject}(r); \text{return } r; \rangle \text{result} \geq 0 \end{aligned}$$

Symbolic Fault Injection

$$\frac{x \geq 0 \vdash \{r:=x\} \langle \text{inject}(r); \text{return } r; \rangle \text{result} \geq 0}{x \geq 0 \vdash \langle r:=x; \text{inject}(r); \text{return } r; \rangle \text{result} \geq 0}$$
$$\vdash \langle \text{if}(x \geq 0) \{r:=x; \} \text{else} \{r=0; \} \text{inject}(r); \text{return } r; \rangle \text{result} \geq 0$$

Symbolic Fault Injection

$$\frac{x \geq 0 \vdash \forall j : int. \{r:=x\} \{r:=j\} \langle \text{return } r; \rangle \text{result} \geq 0}{x \geq 0 \vdash \{r:=x\} \langle \text{inject}(r); \text{return } r; \rangle \text{result} \geq 0}$$
$$\frac{x \geq 0 \vdash \langle r:=x; \text{inject}(r); \text{return } r; \rangle \text{result} \geq 0}{\vdash \langle \text{if}(x \geq 0) \{r:=x;\} \text{else} \{r=0;\} \text{inject}(r); \text{return } r; \rangle \text{result} \geq 0}$$

Symbolic Fault Injection

$$\frac{x \geq 0 \vdash \forall j : int. j \geq 0}{x \geq 0 \vdash \forall j : int. \{result:=j\} \langle \rangle result \geq 0}$$
$$\frac{x \geq 0 \vdash \forall j : int. \{r:=j\} \{result:=r\} \langle \rangle result \geq 0}{x \geq 0 \vdash \forall j : int. \{r:=j\} \langle return r; \rangle result \geq 0}$$
$$\frac{x \geq 0 \vdash \forall j : int. \{r:=x\} \{r:=j\} \langle return r; \rangle result \geq 0}{x \geq 0 \vdash \{r:=x\} \langle inject(r); return r; \rangle result \geq 0}$$
$$\frac{x \geq 0 \vdash \{r:=x\} \langle inject(r); return r; \rangle result \geq 0}{x \geq 0 \vdash \langle r=x; inject(r); return r; \rangle result \geq 0}$$
$$\vdots$$
$$\vdash \langle \text{if}(x \geq 0) \{r=x; \} \text{else} \{r=0; \} inject(r); return r; \rangle result \geq 0$$

Symbolic Fault Injection

Another kind of rule ...

$$\text{int_single} \frac{\vdash \forall j : \text{int}. 0 \leq j \leq 31 \rightarrow \{i := i \wedge (1 \ll j)\} \langle \omega \rangle \phi}{\vdash \langle \text{injectSingle}(i); \omega \rangle \phi}$$

Case Study: CRC

- Fault detection mechanism CRC (Cyclic Redundancy Check)
- Checksum algorithm
 - Send: Calculate checksum of data, send data & checksum
 - Receive: Calculate checksum of data. Compare with appended checksum.
 - If they do *not* agree: Data has been modified
 - Otherwise: Either no modification or fault is undetected
- A good checksum algorithm minimizes the risk that several faults “even out” w.r.t. the checksum

Case Study: CRC

Short description of CRC algorithm

- Treats data blocks as binary repr. of integers
- The integer is divided with a predetermined divisor.
- The remainder of division becomes the checksum
- Polynomial division instead of the “usual” one

CRC implementation

- Checksum cannot be calculated in one step
- Data block is fed through a register while arithmetic operations are applied to its content

Case Study: CRC

```
public static byte compute(byte[] buffer) {  
    int count = buffer.length;  
    byte register = (byte)0x0;  
    while (count > 0) {  
        byte element = buffer[buffer.length - count];  
        int t = ((int)(register ^ element) & 0xff);  
        register <<= 8;  
        register ^= table[t];  
        count--;  
    }  
    return register;  
}
```

Case Study: CRC

- Proof attempt: CRC implementation detects all single bit-flips
- Not proven for arbitrary size of data, which demands non-trivial induction proof
- Proven for an array size of 5 elements

Case Study: CRC

Test harness:

```
static boolean crcTest(byte[] msg) {  
    byte crc1 = Crc.compute(msg);  
    injectSingle(msg[0]);  
    byte crc2 = Crc.compute(msg);  
    return (crc1 != crc2);  
}
```

Proof obligation:

$$\forall msg1 : \text{byte} [] . (msg1.length \doteq 5$$
$$\rightarrow \{msg := msg1\} \langle \text{Crc.crcTest}(msg); \rangle (result \doteq \text{true}))$$

Case Study: CRC

- Symbolic execution was automatic. About 800 rule applications
- Resulting sequent contained large arithmetic expressions and could not be proved using KeY
- Proved by transformation to JAVA expressions, which were exhaustively tested

Limitations

Our technique cannot ...

- simulate faults in specific parts of the hardware
- evaluate real-time properties
- handle floating point data types
- handle multi-threaded programs

Future Work

- Induction-based proof of CRC implementation
- Generalization to different fault models and fault trigger mechanisms
- Injection of *software* faults
- Focus on weaker properties: Not always meaningful to talk about functional correctness; often sufficient to prove that application will not crash or hang

Questions?