Compiler-Assisted Automatic Error Detection

Seminar on Software-Based Fault-Tolerance

Jan Bessai

Outline

Why use a Compiler?

No need to change existing code:

- implement and test protection methods only once
- create variants by switching compile flags
- separate *aspect* of protection from program functionality

MASK

Example

```
mov ecx, 0 ; init counter
and ecx, 0xFFFE
again:
  push ecx
  call innerLoopFun ; with param ecx
  add ecx, 2 ; increase counter by 2
  and ecx, 0xFFFE
  cmp ecx, 10 ; loop if ecx < 10
  jl again
```

Observation:

- last bit of ecx is always zero
- invariant enforced by bitmask `0xFFFE`

Explaination

Containment of errors

- static analysis to find invariants
  - e.g. Bitvector analysis on single bits (known from constant propagation)
- invariants enforced with bitmasks throughout code
- only bits really involved in computations can flip

What MASK can detect/prevent

- transient and permanent faults in masked bits
- flipped operands (if constraints are violated)
- invalid jump addresses (known from NX-Bit)

What MASK cannot do

- protect bits which are expected to change
- ensure application of bitmasks does not fail
- strength of protection depends on capabilities of the compiler

Implementation considerations

- tight coupling with compiler analysis code
- intimate knowledge of target architecture required
- consideration of side effects
  - e.g. if register indirect commands are involved

AN-Encoding

Checking with Modulo

To protect the arithmetic operations $+$ and $*$:

- choose a big constant A
- multiply unencoded values x, y with A:

$$A \cdot x + A \cdot y \mod A = A \cdot (x + y) \mod A \quad = 0$$
$$(A \cdot x) \cdot (A \cdot y) \mod A = A^2 \cdot (x \cdot y) \mod A \quad = 0$$

Protect logic by replacement with arithmetic

- $\neg x = 1 - x$, $x \wedge y = x \cdot y$, ...

Properties

Hamming distance of valid words is spread:

- $A = 127$
- unencoded: $0000\ 0000 \xrightarrow{\text{1 Bit flipped}} 0000\ 0001$
- encoded: $0000\ 0000 \xrightarrow{\text{7 Bits flipped}} 0111\ 1111$

$\Rightarrow$ Datatypes have to grow by a factor of 4:

$$\underbrace{\overbrace{A \cdot x}^{\times 2} \cdot \overbrace{A \cdot y}^{\times 2}}_{\times 4}$$

Choice of A

Mersenne primes $2^n - 1$ are optimal for A

- off by $|(2^n - 1) - 2^k| > 0$, if bit k flips
- multiplication shortcut: $x \cdot (2^n - 1) = (x << n) - x$
- no way to create A from factors

Weaving in AN-Encoding

Implementation can be done using AspectC++

- provide a way to annotate values to protect
- aspect to change datatypes
- aspect to perform encoding
- available on GitHub:
  https://github.com/JanBessai/anencoding

Transparent container-based annotation

Create a container `Protected<T>`:

- constructor takeing value of type T
- member `value` of type `Encoded<T>::EncodedType`
- arithmetic operations
- cast back to type T

Type encoding:

```
template <class T>
class Encoded {
 public:
  typedef T EncodedType;
};
```

Aspect header to change types

Provide template specializations of `Encoded<T>`:

```
template <>
class Encoded<char> {
  typedef int EncodedType;
};

template <>
class Encoded<int> {
  typedef mpz_class EncodedType;
};
```

Note: GNU Multiple Precision Arithmetic Library used for 128 bit integers (`mpz_class`)

AN-Encoding Aspect (Encoding, Decoding)

```
aspect ANEncode {
  static const int A = 127;

  pointcut encode() =
    call("% protect<...>(...)");
  advice encode() : after() {
    tjp->result()->value *= A;
  }

  pointcut decode() =
    call("% Protected<...>::operator %()")
    && !negation()
  advice decode() : after() {
    *(tjp->result()) /= A;
  }
  ...
};
```

AN-Encoding Aspect (Operators)

```
pointcut addition() =
  call("% Protected<...>::operator+(...)");
pointcut multiplication() =
  call("% Protected<...>::operator*(...)");
...
pointcut checkable() =
  addition() || multiplication() || ...;
advice checkable() : after() {
  if (tjp->result()->value % A != 0) {
    cout << "Error detected!" << endl;
    cout << tjp->result()->value << endl;
  }
}

advice multiplication() : after() {
  tjp->result()->value /= A;
}
```

What AN-Encoding can detect

Redundancy in encoding

- transient and permanent faults in arithmetic/logic operations (if modulo works..)
- extensions (ANB, ANBD) to find exchanged operands and faulty jumps:

$$
\begin{aligned}
A \cdot x + A \cdot y + B_x + B_y + D \mod A &= \\
A \cdot (x + y) + B_x + B_y + D \mod A &= B_x + B_y + D \\
(A \cdot x) \cdot (A \cdot y) + B_x + B_y + D \mod A &= \\
A^2 \cdot (x \cdot y) + B_x + B_y + D \mod A &= B_x + B_y + D
\end{aligned}
$$

- B_x, B_y: operand identifiers
- D: identifier for expected source position

SWIFT

Example

```
push eax ; safe eax
add eax, ebx ; compute
push eax ; safe result
mov eax, [esp + 4] ; restore eax
add eax, ebx ; compute again
push ebx ; safe ebx
mov ebx, [esp + 4] ; load previous result
cmp eax, ebx ; compare results
jeq ok
call errorHandler ; on mismatch
ok: ...
```

Explaination

Most intuitive form of error detection

- repeat operations, compare results

Errors can even be repaired:

- compute three times (or more) and use most frequent result

What SWIFT can(not) prevent

Redundancy of operations:

- protects from **transient** faults **only**
- applicable for (almost) every operation
  - if no side effects are present
- no way to deal with faulty jumps

Implementation considerations

- intimate knowledge of target architecture is required (as with MASK)
- performance not only decreased by factor of two
  - additional register pressure
  - commands to safe and restore redundant results
  - code to compare (and vote)

Evaluation

Things to consider

Empirical evaluation based on 3 decisions:

- programs under test
- fault model (which kinds of errors?)
- performance measure for
  - errors prevented
  - overhead introduced

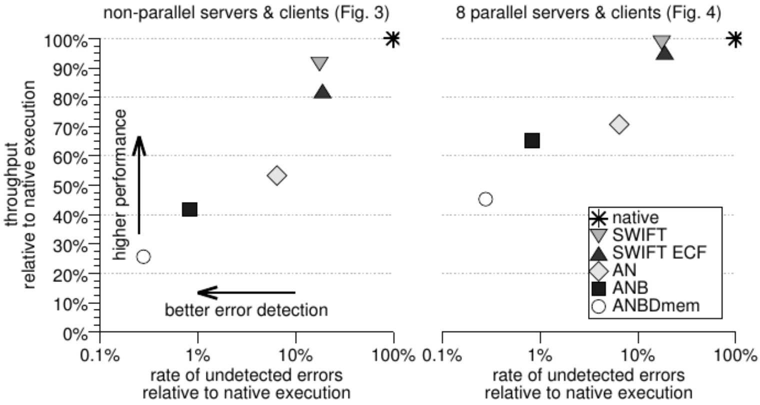
Next the two most complete studies are presented

Study by Schiffel, Schmitt, Süßkraut and Fetzer

"Software-Implemented Hardware Error Detection: Costs and Gains" [2]:

- compared SWIFT and AN-Encoding
- tested 6 standard programs (e.g. md5, primes)
- considered faulty operands, lost operations, permanent and burst faults
- analyzed cases of Silent Data Corruption (SDC)
- measured overhead in a client-server scenario as network throughput
  - included parallel execution

Study by Schiffel, Schmitt, Süßkraut and Fetzer (Results)



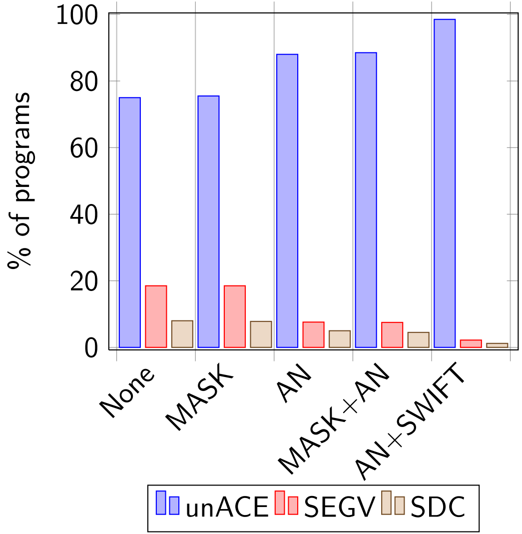
- EFC: Ehanced Controflow Checking (redundant jump targets)
- ANBDmem: D as memory operation counter

Study by Reis, Chung and August

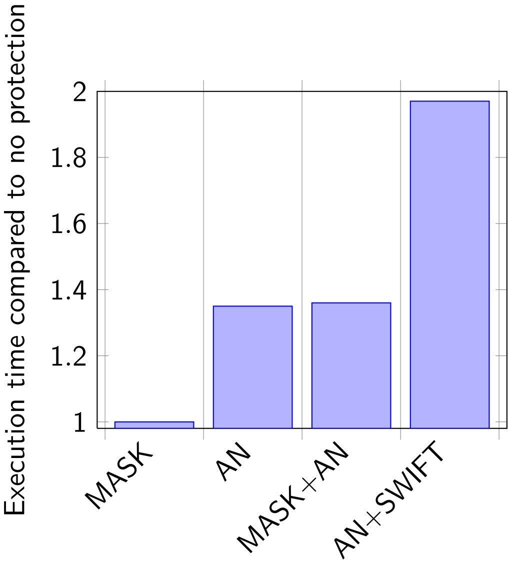
"Automatic Instruction-Level Software-Only Recovery" [1]:

- compared SWIFT, Trump (AN-Encoding + unencoded copy for recovery) and MASK
- tested 21 different programs
- considered Single Event Upset (SEU) faults involving 1 flipped Bit only
- analyzed SDC, Segfaults (SEGV) and errors in data unnecessary for architecutally correct execution (unACE)
- measured overhead in execution time

Study by Reis, Chung and August (Results)



Study by Reis, Chung and August (Results)



Conclusion and discussion

Methods seen

- MASK: Static analysis for containment of errors
- AN-Encoding: Value level redundancy by multiplication
  - Fine grain Aspect-Oriented implementation
- SWIFT: Operation level redundancy by repetition

Evaluation results

- compilers can reduce error rates up to 90%
- runtime degrades by a factor of about 2
- no method can eliminate all errors
- all methods add code which can fail

Further research topics

- standardized production grade compilers
- better control over which parts of a program need protection
- Multi Strategy Beta Reduction (see seminar paper)
- standardized benchmarks

Feedback or Questions?

- Thank you :) !

References

[1] Reis, G.A., Chang, J., August, D.I.: Automatic Instruction-level Software-Only Recovery. IEEE Micro 27, 36–47 (2007), http://doi.ieeecomputersociety.org/10.1109/MM.2007.4

[2] Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: Software-Implemented Hardware Error Detection: Costs and Gains. In: The Third International Conference on Dependability (DEPEND 2010). pp. 51–57. IEEE Computer Society, Los Alamitos, CA, USA (2010)