Software Countermeasures in the LLVM RISC-V compiler

François de Ferrière

STMicroelectronics

March 30, 2021

Agenda

life.augmented

Introduction

life.augmented

Introduction

- Security extension to LLVM
  - Named SecSwift for Secure Swift
    - Based on a research paper

      SWIFT : Software Implemented Fault Tolerance
      G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, D.J. August – CGO 2005

  - Single Fault model

  - Work started four years ago

  - Internal development on our ports of the LLVM compiler for RISC-V, ARM, and on our proprietary processors

Introduction

- The overall objective is to
  - Replace hand-written countermeasures by automatic generation in the compiler
    - Let the user control what protections to activate and where
    - Let the compiler do the tedious work
  - Provide a report of which transformations have been done and where
    - For verification
    - For certification
    - For debugging and patches
- Full integration with LLVM compiler
  - No constraints on compilation options
    - -Oz, -O2, -O3, -flto levels are fully supported
  - Security code is guaranteed to be preserved by the compiler
  - Security code is efficiently compiled and mixed with application code

life.augmented

Countermeasures

life.augmented

Control-Flow Integrity

- Control-flow integrity checking
  - IDs are assigned to basic blocks and functions
  - A variable is used to duplicate the Program Counter
    - GSR : Global Signature Register
  - A second variable is used on transfers between basic blocks
    - RTS : Runtime Transfer Signature
  - GSR properties
    - Initialized at function entry
    - Updated as a function of its previous value
      - GSR = GSR ^ RTS
    - To be verified at safety critical points only
      - Unprotected call and return instructions
      - At entry of basic blocks with memory write

```
BBx: // ID = SigBBx
     GSR = GSR^RTS;
     assert(GSR == SigBBx); // Optional
     <Body BBx>
     RTS = SigBBx^((x>0?SigBBy:SigBBz);
     if (x>0)
        goto BBy;
     goto BBz;
```

life.augmented

Data-flow Integrity

- Computation-flow integrity
  - Duplication of local scalar variables
  - Duplication of function's parameters and return values
  - Duplicated computations are performed on duplicated variables
  - Checks are inserted at the end of a duplicated data-flow path
    - Before unprotected call and return instructions
    - Before memory operations

```
<int, int> DFI(int x, int _x, int y, int _y){
int z; int _z;
  z = GV + (x - y) * (x + y);
  _z = GV + (_x - _y) * (_x + _y);
return <z, _z>;
}
```

Memory Integrity

- Memory protection of global variables and fields of aggregates
  - Duplication of memory location
    - New global storage for global variables
    - New field next to the original one for aggregates
    - On scalar and array types
  - Checks are inserted before every memory reads
  - Writes are duplicated
    - After writes into the original memory
  - The duplicated value in memory can be :
    - A bitwise-not of the original value (the default)
    - The opposite of the original value
    - A copy of the original value

```c
#include <secswift.h>

typedef struct {
  secswift_memdup_int32_t field;
  int32_t _field; // = ~field
} Safe_t;

int32_t f(Safe_t *S, int i) {
   assert(S->field == ~S->_field);
   int32_t n = S->field + i;
   S->field = n;
   S->_field = ~n;
   return n;
}
```

SecSwift Annotations

SecSwift Annotations

- SecSwift Annotations gives feedback to the user on applied transformations
  - Location in source code
  - Variable/Field on which it applies

- SecSwift Annotations are a key element for certification by external entities
  - The source code only contains a few annotations on which transformations to apply
  - The protections are too difficult to analyses in the optimized assembly code

- SecSwift Annotations are available for
  - Memory Integrity
  - Control-Flow Integrity
  - Data-Flow Integrity is under development

life.augmented

SecSwift Annotations

- SecSwift Annotations are available under our customized Visual Studio Code
  - LLVM Annotations are displayed as diagnostics
    - YAML files generated by LLVM are analyzed to decorate the source code
  - Features under implementation
    - Provide “IntelliSense” completion for SecSwift attributes
    - Provide a disassembly view which highlights the code added by SecSwift

```
__attribute__((noinline)) struct s *use(struct s *g, int x) {
  int z = 10, w;

  z = x >> 2;

  if (x < 5)
    __builtin_secswift_assert(g->val3);
ⓘ mem.c 1 of 12 problems
Replaced by secswift_assert(->val3 == ~->_SECSWIFT_val3); (Merged with one(s) from other branch(es)) Clang(-Rpass-analysis=secswift)
  else
    __builtin_secswift_assert(g->val3);

  return g;
}
```

life.augmented

Qualification

Qualification

- Based on GDB scripts
  - Branch inversion
  - Instruction skip
  - Instruction re-execution
  - Register injection (0xffffffff & 0x0)

- Based on a symbolic execution tool : angr
  - Performs symbolic execution to find attacks on register values that modify program’s behavior
  - Operates on an intermediate representation : VEX (Valgrind’s one)
  - Multi-architecture
    - but RISC-V is still WIP

Qualification

- Classification of the result of an attack
  - No effect
    - No visible effect on the behavior and output of the execution
  - Crash
    - The attack resulted in a crash of the execution
  - Detected
    - SecSwift countermeasure code triggered a call to the secswift_abort function
  - Successful
    - The execution ended normally but produced a different output

      enter a wrong key but continue the execution as if it was correct !

life.augmented

Qualification

- Results
  - Performed on a few “small” benchmarks
    - Coremark, PStone, Stanford, …
    - Internal benchmarks
  - 0% of successful attacks on branch inversion
    - Was about 99% without protection
  - 1.5% mean (9.31% max) of successful attacks on instruction skip
    - Was about 70% without protection
  - 2% mean (4.83% max) of successful attacks on register fault injection
    - Was about 50% without protection

life.augmented

Qualification

- Code size and performance impact
  - Control-Flow Integrity
    - Applied on entire benchmark
    - Code size : ~ +75%
    - Cycle count : ~ +50%
  - Data-Flow Integrity
    - Applied on entire benchmark
    - Code size : ~ +150%
    - Cycle count : ~ +200%
  - Memory Integrity
    - Replaced handwritten source code protection by automatic SecSwift protection on a real customer application
    - Improved code size and cycle count
      - Higher optimization level could be used to compile the application
    - Easier to fine tune and revealed bugs in handwritten protections

Conclusion

life.augmented

Conclusion

- Extension of memory and data-flow duplication
  - Add support for function calls and pointers
- Continue the integration of SecSwift annotations under Visual Studio Code
- Validate more benchs with our qualification scripts
  - More analysis on symbolic execution for fault-injection
  - Analyze faults that are not currently detected by SecSwift
- Implement other countermeasures on request
  - Test duplication
  - Triplication with voting
  - Protection of peripheral registers

Thank you

life.augmented