Software fault injection for SecSwift qualification

Hervé Chauvet - François de Ferrière - Thomas Bizet

ST Grenoble – Compilers and Computing Center

September 23, 2021

Summary

life.augmented

Context

life.augmented

Context

- Software security extension to LLVM
  - A software countermeasures module developed in ST ports of the LLVM compiler
  - Named SecSwift for Secure Swift

    SWIFT : Software Implemented Fault Tolerance
    G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, D.J. August – CGO 2005
  - Supported architectures
    - ARM
    - STxP5 (RISC-V): under development
    - Other proprietary processors
- The overall objective is to replace hand-written countermeasures by automatic generation in the compiler
    - Let the user control which protections to activate and where
    - Let the compiler do the tedious work

life.augmented

Context

- Full integration with the 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

- Need for a fault-injection tool
  - Validate SecSwift’s countermeasures effectiveness
  - Provide a way to qualify an application protected with SecSwift
  - Will be used for continuous integration of SecSwift developments

- Fault model
  - SecSwift protections support single-fault model

Software Faults

life.augmented

Software Faults

- Our tool can inject several kinds of software faults
  1. Conditional branch inversion
     BNE -> BEQ, BLE -> BGT, ….
  2. Instruction skip
     PC += 4
  3. Instruction re-execution
     PC -= 4
  4. Register value modification
     Reg = 0x0, 0xffffffff (-1); 0xffffff80 (-128) for alignment purposes
     Reg = Reg xor (1<<n)
     - Register value modifications on load & store operations simulate injections on memory
- Limitations
  - Can only apply to registers and instructions
  - Mimics the effects of physical faults & their consequences

life.augmented

Software Faults

- Fault types used to evaluate SecSwift countermeasures

SecSwift countermeasures	Branch inversion	Instruction skip	Instruction re-execution	Register injection
Control-flow integrity	✓	✓		✓
Data-flow duplication		✓	✓	✓
Memory duplication (global variables & aggregate members)				✓

life.augmented

Simulation & Fault Injection

Qualification steps

- The qualification process is implemented as two independent steps:

- Advantages:
  - After the collection step, the number of faults to be injected is known:
    -> The time required to perform fault injection can be easily approximated.
    -> Resources and parallelism degree for the injection step can be adjusted
  - Fault injection tasks can be easily distributed over a pool of processes or machines.
- Drawbacks:
  - Execution traces can be very large, up to tens of Gigabytes

life.augmented

Collection step

1. Execution of the program and generation of an execution trace
   - Can be reduced to a list of functions or range of addresses

2. Parsing of the execution trace to create a set of faults to be injected

   ➔ Each selected fault kinds is applied to each instruction

   Input/output registers, opcode of the instruction

   ➔ An instruction will be targeted as many time it is executed

3. Write down a JSON file that contains the list of faults to be injected

   (*Address*, *Occurrence*, *Fault*)

life.augmented

Collection step

- Number of injected faults

Fault Kind	Number of fault Injections
Branch Inversion	One fault injection for each branch instruction
Instruction Skip	One fault injection for each instruction
Instruction re-execution	One fault injection for each instruction
Register value modification	One fault injection for each input register and for each injected value

- On real applications

Program	Number of collected fault injections	Execution time
Summin	380 000	20s
Coremark	550 000	1min
Stanford	1 150 000	~ 6min

Fault Injection Step

- Processing of one fault injection
  - A GDB session is initiated and attached to a simulated execution of the program
  - A fault injection Python script is executed under GDB:
    - Set a breakpoint in the program at the fault injection **address**.
    - Start the program execution until the **occurrence** of the breakpoint is reached.
    - Perform the **fault injection** action via GDB
    - Resume the program execution
    - Classify the result of the fault injection

Classification	Effects
Successful attack	• Program's behavior or correctness has been modified • The fault has not been detected by SecSwift countermeasures
Fault detected	• The fault triggered SecSwift countermeasures
Correct execution	• Program's behavior/correctness has not been modified • The fault has not been detected
Unexpected execution	• The fault injection caused a crash or modified the execution of the program
Timeout	• The program did not terminate in time and had to be interrupted

life.augmented

Fault injection execution

One computer	Linux Computer Farm
One supervisor process distributes fault injection tasks to a pool of N fault injection processes	One supervisor process starts K distributed jobs that each execute N fault injection processes
Reduces the fault injection time by a factor of ~ N (limited by the number of cores on the computer)	Reduces the fault injection time by a factor of ~ K * N (K = 30 and N = 8 for our experiments)
Allows to qualify small to medium size applications.	Can easily be used to qualify applications of a few thousand lines

- For a program with an execution time of 0.6s:
  - Can perform about 6 000 fault injections per hour on one core
  - About 1.5 million of fault injections per hour on a farm of 30 machines with 8 cores each
  - Can qualify applications of a few hundred lines of code within a few hours
- A time budget can be given to partially qualify larger application
  - Fault injections are picked at random from the list of fault injections

life.augmented

Results

life.augmented

Results

- Number of faults of each type for a given program

Program (ARM)	Targeted instructions	Branch inversions	Skips & re-executions	Register injections
Coremark	7 256	570	7 256	33 906
Quicksort	10 625	1 665	10 625	45 627
Pstone/summin	66 203	8 197	66 203	245 481
Stanford	211 235	26 148	211 235	884 904

- Rate of successful attacks

Fault type	Successful attacks without protections	Successful attacks with SecSwift protections
Branch inversion	99 %	0 %
Instruction skip & re-execution	70 %	0.3 %
Register injection	50 %	0.5 %

life.augmented

Countermeasure improvements

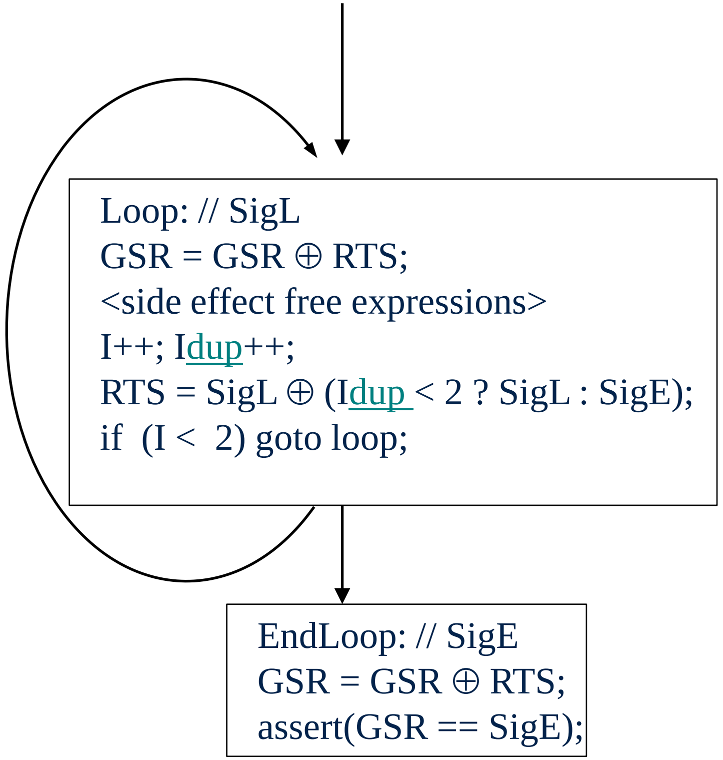
life.augmented

Countermeasure improvements

- The analysis of the “Successful Attacks” cases resulted in the identification of weaknesses in the SecSwift protections:
  - Use of the XOR operator for control-flow integrity checking
  - Missing duplication of instructions to build immediate values
  - Weakness in the checking of stored value
  - Skip of the last branch instruction of a function
- Other “Successful Attacks” cases have yet to be analyzed

Countermeasure improvements

- XOR operator for control-flow integrity checking
  - Undetected fault on a loop where the trip count was increased by an even number
    - XOR is now replaced by a combination of add/sub operations



Loop: // iteration 2
GSR = GSR ⊕ RTS;
<side effect free expressions>
I++; Idup++;
RTS = SigL ⊕ (2 < 2 ? … : SigE);
if (0 < 2) goto loop;
	GSR = SigL
	
	~~I = 2;~~I = 0; Idup = 2;
	RTS = SigL ⊕ SigE
	goto loop

Loop: // iteration 3
GSR = GSR ⊕ RTS;
<side effect free expressions>
I++; Idup++;
RTS = SigL ⊕ (3 < 2 ? … : SigE);
if (1 < 2) goto loop;
	GSR = SigE
	
	I = 1; Idup = 3;
	RTS = SigL ⊕ SigE
	goto loop

Loop: // iteration 4
GSR = GSR ⊕ RTS;
<side effect free expressions>
I++; Idup++;
RTS = SigL ⊕ (4 < 2 ? … : SigE);
if (2 < 2)
	GSR = SigL
	
	I = 2; Idup = 4;
	RTS = SigL ⊕ SigE
	fallthrough

EndLoop: GSR = GSR ⊕ RTS; assert(GSR == SigE);	GSR = SigE SigE == SigE

Countermeasure improvements

- Weaknesses in the SecSwift protection on store operations:
  - A reload of the stored value is added
  - The value must be compared against the duplicate of the stored value
  - Also reported in :
    - A compiler technique for near Zero Silent Data Corruption – M. Didehban, A. Shrivastava. DAC 2016

```
ADD R0, R0, #10   // duplicated computation-flow start
ADD R1, R1, #10   // R1 is duplicate of R0
….
CMP R0, R1
BNE trap  // Attack on R0 here would not be detected
CMP R9, R10
BNE trap
STR R0, [R9]
```

```
ADD R0, R0, #10   // duplicated computation-flow start
ADD R1, R1, #10   // R1 is duplicate of R0
….
STR R0, [R9]
LDR R0, [R10]      // R10 is duplicate of R9
CMP R0, R1
BNE trap
```

Countermeasure improvements

- Missing duplication of instructions to build an immediate value
  - An intrinsic function is used in the compiler to force the generation of a duplicated constant

```
LLVM-IR:
%3  =  add  i32  6000, %1
%4  =  add  i32  6000, %2 // duplicated instruction


ARM generated code:
MOV R0, #6000
ADD R1, R1, R0
ADD R2, R2, R0 // duplicated instruction
```

```
LLVMIR:
%3  =  add  i32  6000, %1
%copy = call  i32 @llvm.hiddencopy(i32 6000)
%4  =  add  i32  %copy, %2 // duplicated instruction

ARM generated code:
MOV R0, #6000
MOV R3, #6000 // intrinsic expansion
ADD R1, R1, R0
ADD R2, R2, R3 // duplicated instruction
```

life.augmented

Countermeasure improvements

- Missing protection at the entry of a control-flow protected region
  - IPGSR is now statically initialized at program load time
  - A check is added at the entry of the protected region
    - We are still looking for a better fix for this case

```
global IPGSR
foo:
    ….
    ….
    return
main:
    IPGSR = InitValue // IPGSR has no context yet
```

```
global IPGSR = InitValue
foo:
    ….
    ….
    return
main:
    assert (IPGSR == InitValue);
```

life.augmented

Perspectives

life.augmented

Perspectives

- Bypass GDB interface to directly inject faults via the simulator/emulator
  - Implement some hooks ?
- Enhance reporting of undetected faults to help for analysis/comparison
  - Link vulnerable instructions to source code
- Study the possibility to implement a snapshot system that copies simulation states in order to avoid re-executing the simulation from the beginning at each injection

life.augmented

Conclusion

life.augmented

Conclusion

- Fault injections scripts have reached a product level
- Used to validate SecSwift countermeasures
  - Already spotted a few weaknesses in the implementation
  - Some “successful attacks” still need to be analyzed
- Qualification should soon be performed on real applications for our internal customers

life.augmented

Thank you

life.augmented