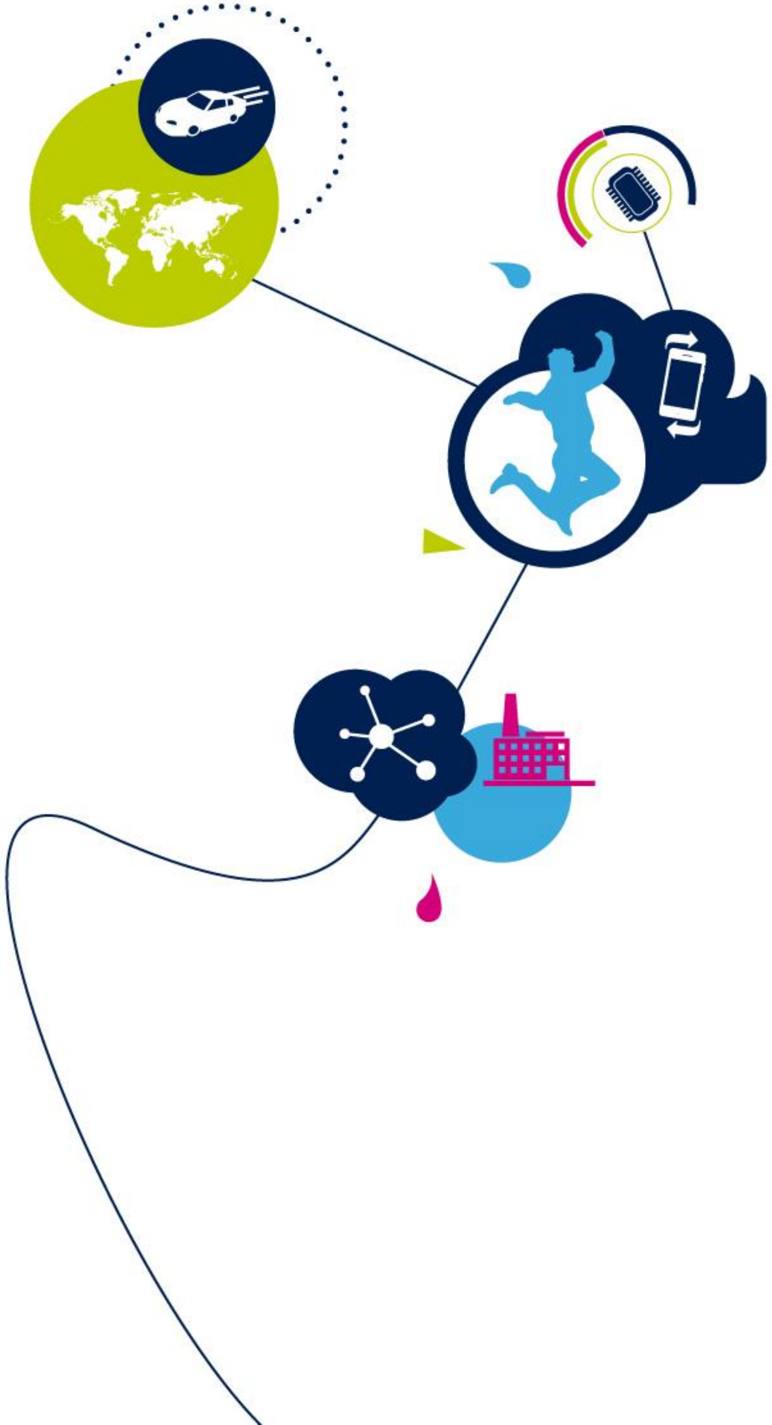
A compiler approach to Cyber-Security

François de Ferrière
Compilers Expertise Center
STMicroelectronics - Grenoble, France

EuroLLVM, April 9th 2019

life.augmented

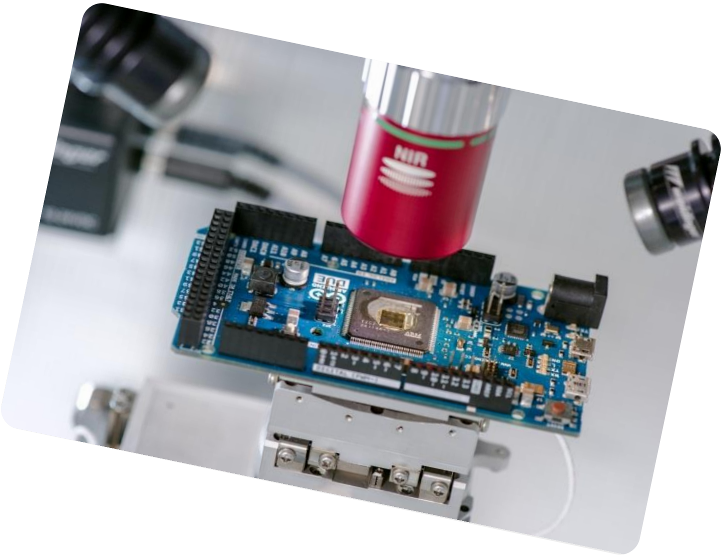


Growing Need of Security in an Open World

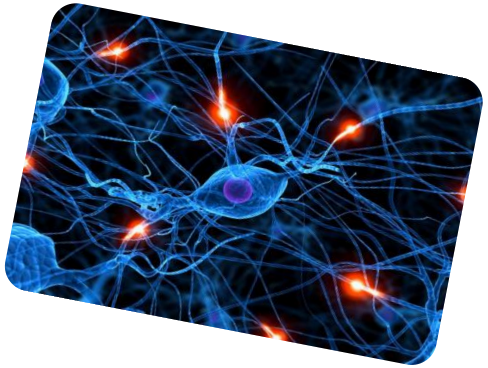
- From traditional dedicated circuits
  - Smartcards, …
  - Built-in security features
  - Short lifespan
- To IOT nodes
  - Fast cryptographic primitives for confidentiality, integrity, authenticity & privacy
  - Power and performance constraints
  - Long lifespan
  - Highly connected
  - Estimate at 20 billions in 2020
    - Smart Homes
    - Health Monitoring
    - Intelligent Transport System
    - Biometric Authentication
    - …

Physical Attacks

- Side Channel Attacks
  - Timing analysis
  - Power analysis
  - Electromagnetic analysis
- Fault injection attacks
  - Laser
  - Electromagnetic pulse
  - Power and clock signals glitches
- Aiming at
  - Obtain sensitive data
  - Bypass protection
  - Reverse engineering



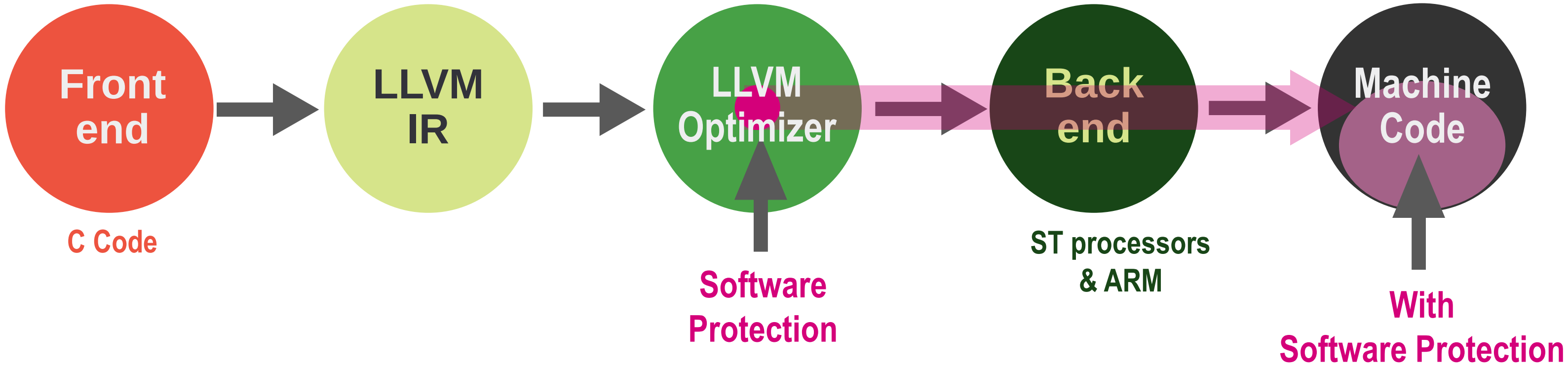
Software-Based Countermeasures



- Aims at protecting against
  - Instruction skip
  - Modification of instructions or data
- Source level protections
  - Easy to implement
  - But compiler optimizations tend to remove redundant code
  - Require some implementation tricks and may be difficult to maintain
  - Demoting compiler optimizations results in poor performance and code size
- Assembly level protections
  - The compiler made heavy transformations to reach good performance and code size
  - Difficult to map source code from assembly instructions
  - Difficult to find available resources for adding extra code after aggressive register allocation and code scheduling
  - Higher risk of introducing errors while implementing countermeasures at this level

The Idea

- A compiler approach
  - Instead of struggling against the compiler, make the compiler work for us
    - No need to modify the source code of an application
    - No need to demote compiler optimizations
  - Security code added by the compiler is part of the code to generate
    - Efficient register allocation and instruction scheduling



History

- EDDI : Error Detection by Duplicated Instructions in super-scalar processors

  N. Oh, P.P. Shirvani, E.J. McCluskey - IEEE Transactions on Reliability 2002
  - Duplicate instructions and use different registers
  - Duplicate memory locations
  - Check points at side effects
- SWIFT : Software Implemented Fault Tolerance

  G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, D.J. August – CGO 2005
  - Designed to reduce performance and code size impact
  - No duplicated storage, no duplicated loads/stores
  - Control-flow checking
- Fault Model
  - Single fault on any instruction
  - Protection is guaranteed if applied on whole program
  - Memory is protected by hardware (ECC, …)

Introducing LLVM SecSwift

- Our implementation in LLVM: Secure Swift -> SecSwift
    - Abort on fault detection

- SecSwift consists in 3 different transformations
        - Can be activated independently of each other
        - Combine and benefit from each other
    - SecSwift Duplicate
        - Duplicate the computation flow
        - Check the equality of values at synchronization points
    - SecSwift ABI (Application Binary Interface)
        - Duplicate parameters and return values
        - Check the equality of values when leaving the SecSwift perimeter
    - SecSwift Control-Flow Integrity
        - Branch instructions inside a function
        - Call and return instructions between functions
        - Propagate a signature along control-flow paths
        - Check validity at synchronization points

SecSwift Duplicate

- Duplicate all instructions
  - Done on the intermediate representation of the LLVM compiler
  - Check equality before synchronization points (store, return)
  - Counter-measure for instruction skip

- Duplicated instructions go through the backend
  - The compiler will not remove the redundant code
    - Use of an intrinsic function to hide copies of constants and variables
    - No need to demote compiler optimizations
  - Redundant code is fully integrated with original code for reg-alloc and scheduling
  - Might have pending caveats
    - Not 100% coverage for now
      - e.g prologue/epilogue expansion done after LLVM IR

```
int neq = 0, _DUP_neq = 0;
 for (int i = 0, _DUP_i = 0; i < N; i++, _DUP_i++) {
    neq |= input[i] ^ expected[i];
    _DUP_neq |= input[_DUP_i] ^ expected[_DUP_i];
 }
 secswift_trap(i == _DUP_i);
 secswift_trap(neq == _DUP_neq);
```

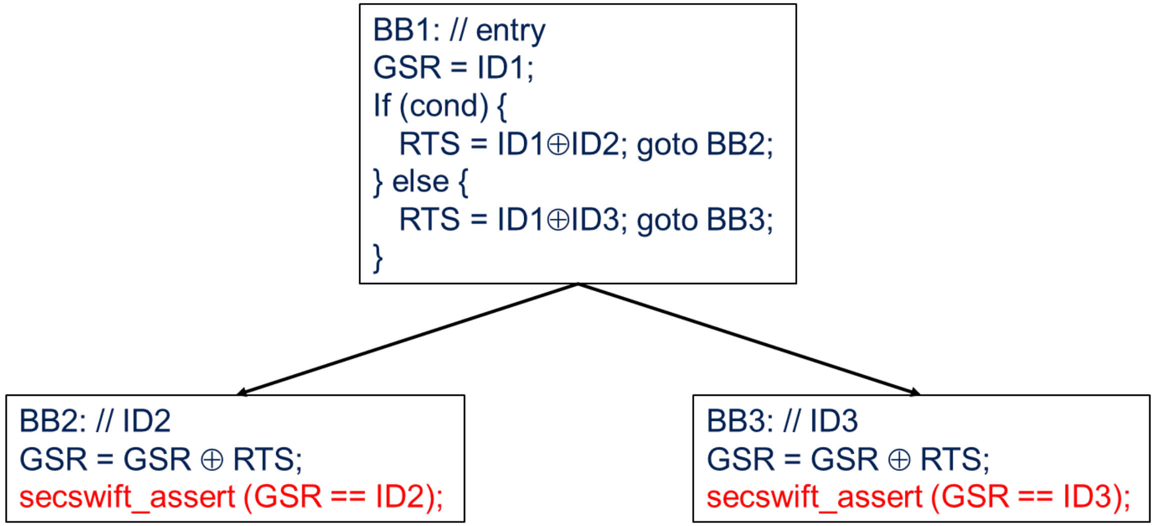
SecSwift ABI

- Change calling convention
  - Counter-measure for corruption of parameters and return values
- A new function prefixed with _SECSWIFT_ is created to use the SecSwift ABI
  - The original function is kept
  - A dead function elimination pass after SecSwift will remove unused functions

```
<int, int> _SECSWIFT_is_invalid(int *input, int *_DUP_input, size_t N, size_t _DUP_N) {
 ....
 return <neq, _DUP_neq>;
}
```

SecSwift Control-Flow

- Control-flow checking: Dynamically checks that branches reach the expected target
  - Counter-measure for fault or skip of branch instructions
  - Based on the property: `A⊕(A⊕B)=B`
  - A static signature is assigned to each basic block: `GSR (General Signature Register)`
  - A dynamic transfer signature is computed on control-flow edges: `RTS (Runtime Transfer Signature)`
  - A check on the signature is inserted at the beginning of basic blocks which have side effect instructions



```
BB1: // entry
GSR = ID1;
If (cond) {
   RTS = ID1⊕ID2; goto BB2;
} else {
   RTS = ID1⊕ID3; goto BB3;
}
```

```
BB2: // ID2
GSR = GSR ⊕ RTS;
secswift_assert (GSR == ID2);
```

```
BB3: // ID3
GSR = GSR ⊕ RTS;
secswift_assert (GSR == ID3);
```

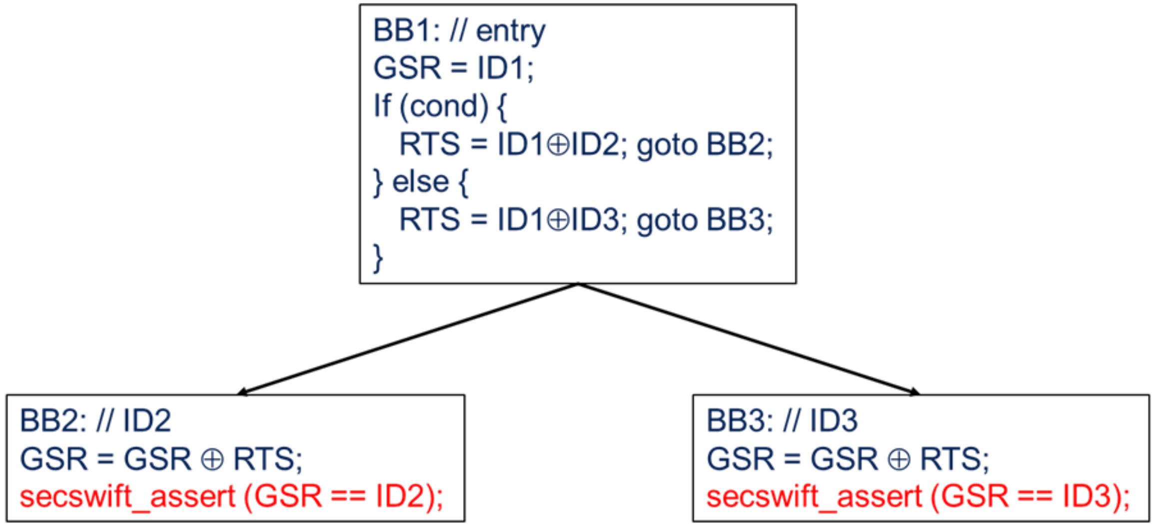
Example 1

```
int GSR = 31155, RTS = 31155 ^ 40106;
 for (int i = 0; i < N; i++) {
    GSR ^= RTS;
    neq |= input[i] ^ expected[i];
    RTS = i < N ? 0 : 40106 ^ 642;
 }
GSR ^= RTS;
secswift_assert(GSR == 642);
```

Example 2

SecSwift CFG

- Why a XOR ?
  - Mathematical properties
  - Fewer gates, compared to an add or mul
- Why a GSR and RTS ?
  - Creates a chain of updates of the GSR value
  - If one `GSR=GSR⊕RTS` is not executed correctly
    - Because of a fault on the instruction
    - Because of an incorrect control-flow transfer
    - Because of an incorrect value in GSR or RTS
  - The error will be propagated in the next computations of the GSRs
    - No need to insert many checks
      - Only before instructions that do side effects
- GSR serves as a redundant duplicate for the Program Counter



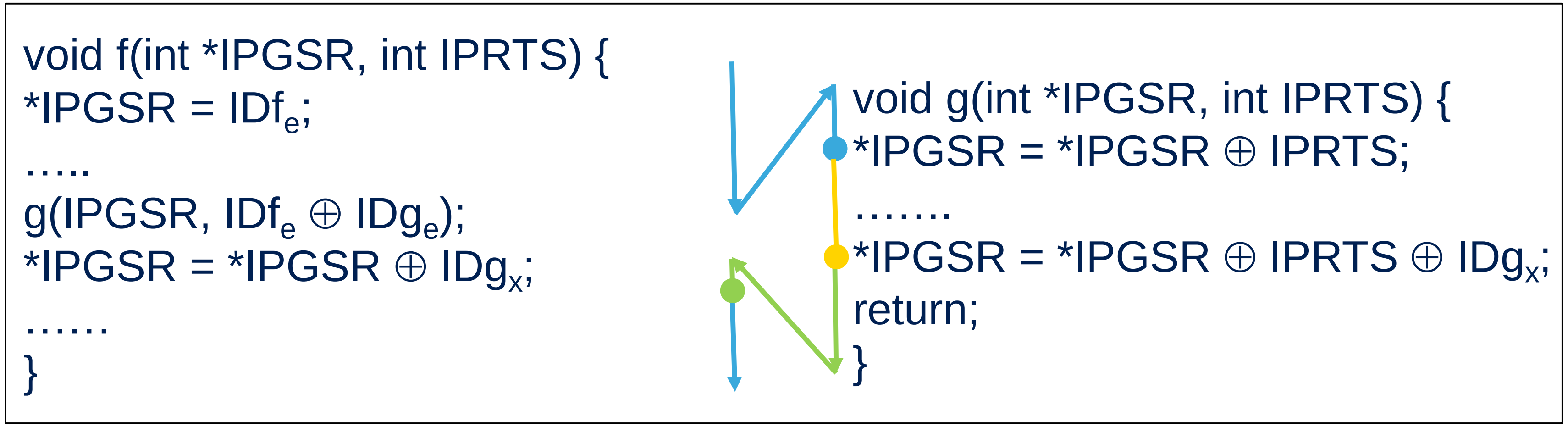
```
BB1: // entry
GSR = ID1;
If (cond) {
   RTS = ID1⊕ID2; goto BB2;
} else {
   RTS = ID1⊕ID3; goto BB3;
}
```

```
BB2: // ID2
GSR = GSR ⊕ RTS;
secswift_assert (GSR == ID2);
```

```
BB3: // ID3
GSR = GSR ⊕ RTS;
secswift_assert (GSR == ID3);
```

SecSwift Inter-Procedural CFG

- Signatures are statically assigned to functions for which IPCFG has been enabled
  - A hash of the function’s name is used to compute the signatures
  - Two signatures are assigned to each function
    - One for the entry point
    - The other one for all the exit points
- Two parameters, IPGSR and IPRTS, are added on functions protected by IPCFG
  - They replace the GSR and RTS variables for function calls and returns



void f(int *IPGSR, int IPRTS) {
*IPGSR = IDf_e;
…..
g(IPGSR, $\mathrm{IDf}_e \oplus \mathrm{IDg}_e$);
*IPGSR = *IPGSR $\oplus$ IDg_x;
……
}

void g(int *IPGSR, int IPRTS) {
*IPGSR = *IPGSR $\oplus$ IPRTS;
…….
*IPGSR = *IPGSR $\oplus$ IPRTS $\oplus$ IDg_x;
return;
}

LLVM Implementation Details

- SecSwift passes are implemented at the LLVM IR level
  - Two generic passes
    - One module pass to implement ABI and IPCFG transformations
    - One function pass to implement DUP and CFG transformations
  - Added at the very end of the LLVM middle-end passes
  - Do not interfere with general optimizations
  - The pass of Global Dead Function Elimination is run again after SecSwift
    - Eliminate dead functions after the application of SecSwift ABI and IPCFG transformations

- Very limited modifications in the target backend
  - We use intrinsic functions and pseudo instructions
    - To prevent copies from being coalesced in the early passes of the Code Generator
    - To generate target dependent code for the SecSwift checks between values
    - They are lowered to real target code before register allocation
  - Support for SecSwift ABI on return values
    - The return value of functions will be duplicated by SecSwift

LLVM Implementation Details

- Activation of SecSwift
  - Each SecSwift transformation can be enabled/disabled independently
    - dup : Duplication of the data flow at basic block level
    - cfg : Control-flow integrity checking at basic block level
    - ipcfg : Control-flow integrity checking on call and return instructions
    - abi : Duplication of function parameters and return value

- Command line options apply to all functions in a file
  - -fsecswift-…

- Function attributes
  - __attribute__((secswift(…, …)))
  - Override command line options
  - Fine tuning of functions on which SecSwift transformations will be applied

LLVM Implementation Details

- Pragma
  - #pragma secswift(…, …)
  - Override command line options and function attributes
  - Apply to the next single instruction or to the next block of instructions
    - Only ‘dup’ and ‘cfg’ are meaningful
  - Reuse the implementation of the “OpenMP Captured” feature
    - The instructions are outlined into a “captured” function
    - Function attributes are used to set the SecSwift options
    - SecSwift is run on a captured function as on the other functions
    - The captured function is inlined back into its original function at the end of the SecSwift passes
- SecSwift options are passed from CLANG to LLVM by means of LLVM function attributes
  - Fully validated and functional in LTO mode

Is the generated code more robust ?

- Historically evaluated “by hand”
  - Security experts analyze software protection implemented at source level
    - Check in generated code that protections are still there
- The compiler must now be part of the certification process
- Tools are needed to improve the evaluation process
  - Simulator with fault injection capability
  - Simple solutions currently used, based on debugger tools
- Evaluation on a simple string compare function
  - Attack is a single skip of an instruction
  - -02: 15 instructions, 13% successful attacks
  - -O2 -sec-dup: 53 instructions, 7% succesful attacks
  - -O2 -sec-cfg: 34 instructions, 3% successful attacks
  - O2 -sec-cfg-dup: 51 instructions, 0% successful attacks

```
int mcompare(unsigned char* s1, unsigned char* s2,
                       unsigned int bytelen) {
  char res = 0;
  int i;
  for (i = 0; i < bytelen; i++) {
    res |= s1[i] ^ s2[i];
  }
  return res;
}
```

How much for this ?

- Evaluation done on ARM Cortex-M0, with options –Oz –flto
  - On a set of 22 benchmarks (eembc, audio/video, dhrystone, coremark, …)
- Performance impact (QEMU instruction count)
  - About 2x slower in average, between 1.5x to 5x
    - Major contribution is -fsecswift-dup
    - -fsecswift-cfg -fsecswift-ipcfg alone is 50% slower in average, 3x at most
    - -fsecswift-abi alone has negligible impact
- Code size impact
  - About 3x larger in average, between 1.5x to 4x larger
    - -fsecswift-dup is 2.5x larger in average, 3.5x at most
    - -fsecswift-cfg -fsecswift-ipcfg is 2x larger in average, 3.5x at most
    - -fsecswift-abi alone has negligible impact
- Not the whole application code need to be protected
  - Only safety critical application parts
    - Fine scoping through pragmas and function attributes
  - SecSwift impact on performance and code size is comparable to compiling at –O0 without protection

Perspectives

- Continuous race between attacks and countermeasures
  - Fault attacks
    - More and more precise attacks
      - Timing of the attacks
      - Very precise location on a chip
    - Synchronized multiple attacks
  - Countermeasures
    - Protection against skip of multiple instructions has been proposed
    - Add some randomization
      - dead-code
      - random memory location
- No single hardware or software protection, both are needed

Conclusion

- Manually implemented software protection is too limited
  - Sophistication of attacks
  - Complexity of countermeasures
  - Risk on time-to-market
- We provide compilation tools that enable security hardening transformations
  - That would not be reasonably doable by hand – **productivity**
  - That can be local enough to stay limited in resource demand increase - **controllability**
  - That can be global enough to treat arbitrary code bases - **scalability**
  - That play well together - **composability**
  - That are semantically correct for already semantically correct code – **soundness**
- New roles for the security experts
  - Propose new or adapted software counter-measures
  - Validate the counter-measures in the compiler rather than in the final application code
  - Determine which counter-measure are needed on which part of an application

Thanks for your attention

life.augmented