Securing Optimized Code Against Power Side Channels

Rodothea Myrsini Tsoupidi Roberto Castañeda Lozano Elena Troubitsyna
Panagiotis Papadimitratos

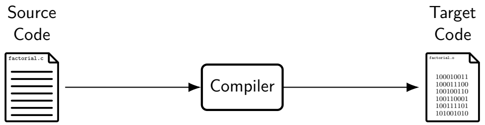
KTH, Royal Institute of Technology

Contents

Contents

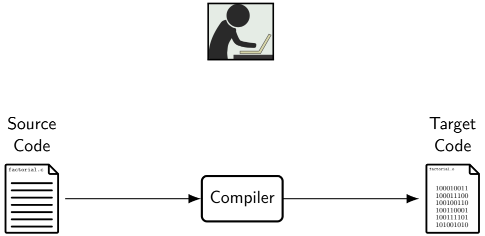
Power Side-Channel (PSC) Attacks

- Exploit properties of the **machine code** of a program



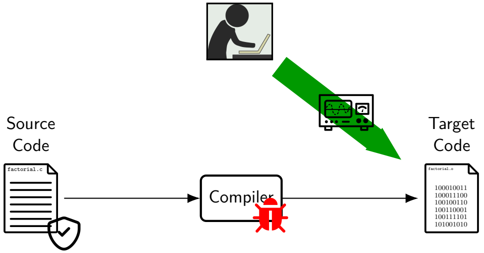
Power Side-Channel (PSC) Attacks

- Exploit properties of the **machine code** of a program



Power Side-Channel (PSC) Attacks

- Exploit properties of the **machine code** of a program
- The attacker records the **power consumption** of the running program



PSC Attacks

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

PSC Attacks

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

0x00000000

The **power traces** depend on program transition between zeros and ones (exclusive OR).

PSC Attacks

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

0x00000000

0xBAADC0DE

The **power traces** depend on program transition between zeros and ones (exclusive OR).

PSC Attacks

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

$$\begin{array}{rr} & \text{0x00000000} \\ \oplus & \text{0xBAADC0DE} \\ \hline & \text{0xBAADC0DE} \end{array}$$

The **power traces** depend on program transition between zeros and ones (exclusive OR).

PSC Attacks

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

```
   0x00000000
⊕  0xBAADC0DE
-------------
   0xBAADC0DE
```

software secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  u32 mk = mask ⊕ key;
  u32 t = mk ⊕ pub;
  return (mask, t);
}
```

PSC Attacks

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

$$\begin{array}{r} \texttt{0x00000000} \\ \oplus \quad \texttt{0xBAADC0DE} \\ \hline \texttt{0xBAADC0DE} \end{array}$$

software secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  u32 mk = mask ⊕ key;
  u32 t = mk ⊕ pub;
  return (mask, t);
}
```

binary of secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  reg1 ← mask;
  reg1 ← reg1 ⊕ key;
  reg1 ← reg1 ⊕ pub;
  return (mask, reg1);
}
```

PSC Attacks

Transitions between the **old** and the **new** values in hardware registers **leak** the difference in ones and zeros (hamming distance).

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

```
  0x00000000
⊕ 0xBAADC0DE
  ----------
  0xBAADC0DE
```

software secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  u32 mk = mask ⊕ key;
  u32 t = mk ⊕ pub;
  return (mask, t);
}
```

binary of secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  reg1 ← mask;
  reg1 ← reg1 ⊕ key;
  reg1 ← reg1 ⊕ pub;
  return (mask, reg1);
}
```

PSC Attacks

Transitions between the **old** and the **new** values in hardware registers **leak** the difference in ones and zeros (hamming distance).

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

```
  0x00000000
⊕ 0xBAADC0DE
  ----------
  0xBAADC0DE
```

software secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  u32 mk = mask ⊕ key;
  u32 t = mk ⊕ pub;
  return (mask, t);
}
```

binary of secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  reg1 ← mask;
  reg1 ← reg1 ⊕ key;
  reg1 ← reg1 ⊕ pub;
  return (mask, reg1);
}
```

0xDEADBEEF

PSC Attacks

Transitions between the **old** and the **new** values in hardware registers **leak** the difference in ones and zeros (hamming distance).

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

```
   0x00000000
⊕  0xBAADC0DE
-------------
   0xBAADC0DE
```

software secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  u32 mk = mask ⊕ key;
  u32 t = mk ⊕ pub;
  return (mask, t);
}
```

binary of secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  reg1 ← mask;                        0xDEADBEEF
  reg1 ← reg1 ⊕ key;      ⊕ 0xDEADBEEF ⊕ 0xBAADC0DE
  reg1 ← reg1 ⊕ pub;
  return (mask, reg1);
}
```

PSC Attacks

Transitions between the **old** and the **new** values in hardware registers **leak** the difference in ones and zeros (hamming distance).

insecure Xor

```
u32 Xor(u32 pub, u32 key) {
  u32 t = pub ⊕ key;
  return t;
}
```

```
  0x00000000
⊕ 0xBAADC0DE
  ----------
  0xBAADC0DE
```

software secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  u32 mk = mask ⊕ key;
  u32 t = mk ⊕ pub;
  return (mask, t);
}
```

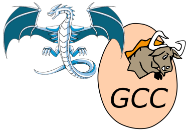
binary of secure Xor

```
u32 SecXor(u32 pub,
           u32 mask,
           u32 key) {
  reg1 ← mask;
  reg1 ← reg1 ⊕ key;
  reg1 ← reg1 ⊕ pub;
  return (mask, reg1);
}
```

```
                          0xDEADBEEF
⊕ 0xDEADBEEF ⊕ 0xBAADC0DE
  ------------------------
                          0xBAADC0DE
```

Securing Binary Code

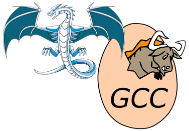
General-Purpose Compilation



- Focus on **performance**

Securing Binary Code

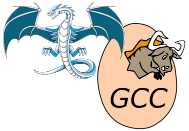
General-Purpose Compilation



- Focus on **performance**
- Generate code for **multiple targets**

Securing Binary Code

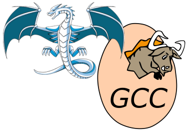
General-Purpose Compilation



- Focus on **performance**
- Generate code for **multiple targets**
- Do not consider power side channels

Securing Binary Code

General-Purpose Compilation

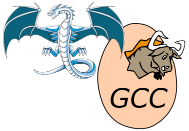


- Focus on **performance**
- Generate code for **multiple targets**
- Do not consider power side channels

Conventional Compilation (Wang et al. '19)

Securing Binary Code

General-Purpose Compilation



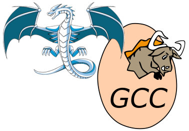
- Focus on **performance**
- Generate code for **multiple targets**
- Do not consider power side channels

Conventional Compilation (Wang et al. ’19)

- **Portable** approach

Securing Binary Code

General-Purpose Compilation



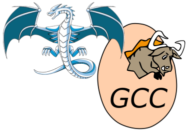
- Focus on **performance**
- Generate code for **multiple targets**
- Do not consider power side channels

Conventional Compilation (Wang et al. '19)

- **Portable** approach
- No focus on the performance of the code

Securing Binary Code

General-Purpose Compilation



- Focus on **performance**
- Generate code for **multiple targets**
- Do not consider power side channels

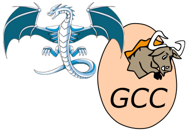
Conventional Compilation (Wang et al. '19)

- **Portable** approach
- No focus on the performance of the code

Binary-Rewriting (Shelton et al. '19)

Securing Binary Code

General-Purpose Compilation



- Focus on **performance**
- Generate code for **multiple targets**
- Do not consider power side channels

Conventional Compilation (Wang et al. '19)

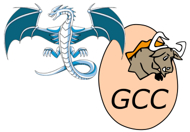
- **Portable** approach
- No focus on the performance of the code

Binary-Rewriting (Shelton et al. '19)

- Not portable, adjusted to one processor

Securing Binary Code

General-Purpose Compilation



- Focus on **performance**
- Generate code for **multiple targets**
- Do not consider power side channels

Conventional Compilation (Wang et al. '19)

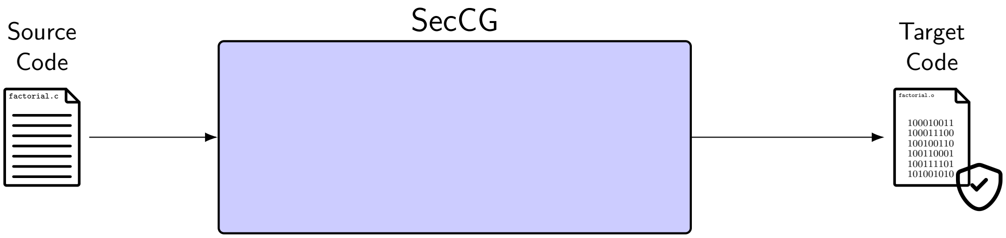
- **Portable** approach
- No focus on the performance of the code

Binary-Rewriting (Shelton et al. '19)

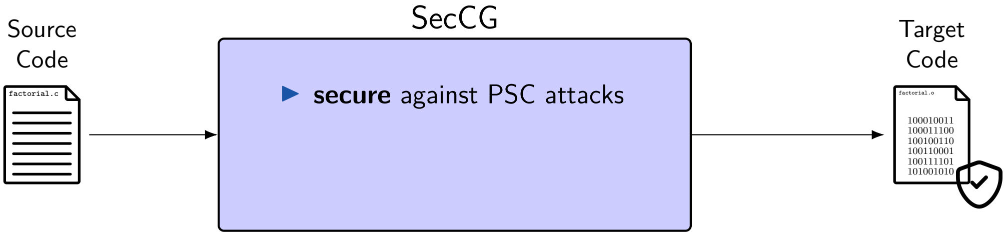
- Not portable, adjusted to one processor
- Good performance but introduces overhead

Security-Aware Code Generation (SecCG)

Security-Aware Code Generation (SecCG)

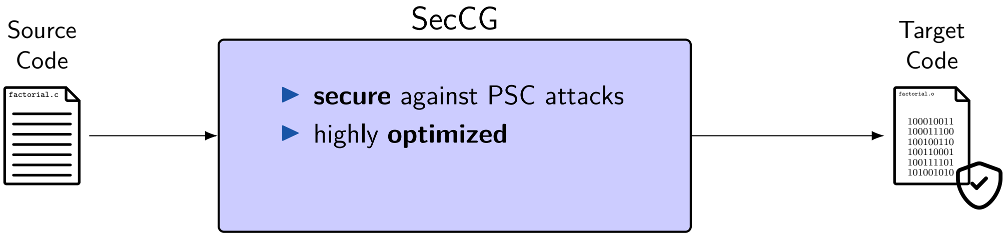


Security-Aware Code Generation (SecCG)



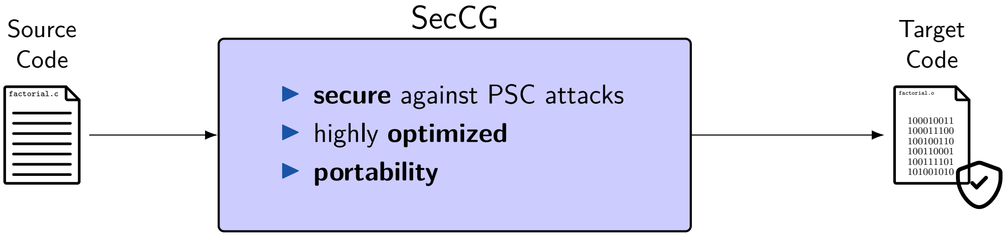
- **secure** against PSC attacks

Security-Aware Code Generation (SecCG)



- **secure** against PSC attacks
- highly **optimized**

Security-Aware Code Generation (SecCG)



- **secure** against PSC attacks
- highly **optimized**
- **portability**

Contents

Constraint Programming (CP)

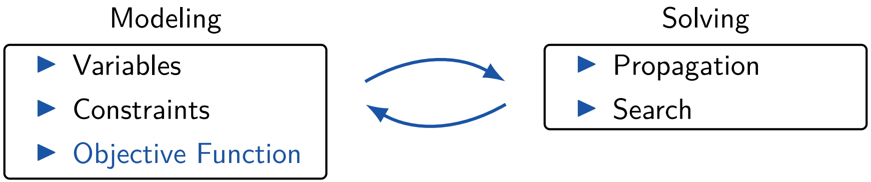
Modeling

- Variables
- Constraints
- Objective Function

Solving

- Propagation
- Search

Constraint Programming (CP)



Modeling

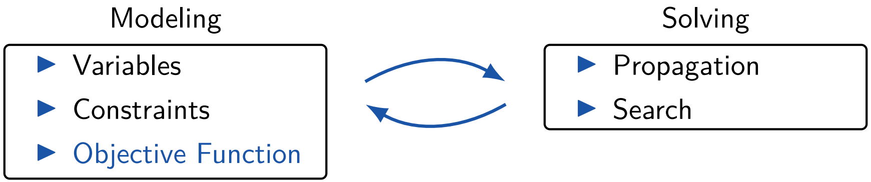
- Variables
- Constraints
- Objective Function

Solving

- Propagation
- Search

CP strengths:

Constraint Programming (CP)



CP strengths:

- Global constraints

Constraint Programming (CP)

Modeling

- Variables
- Constraints
- Objective Function

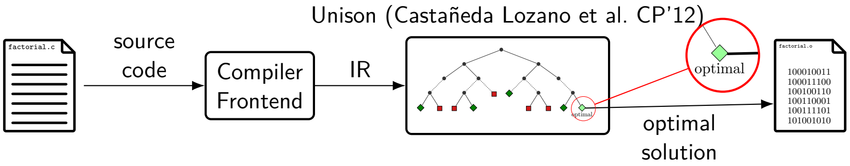
Solving

- Propagation
- Search

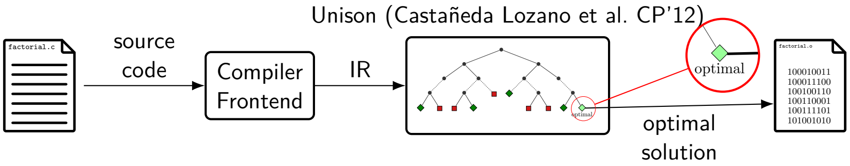
CP strengths:

- Global constraints
- Control over search (e.g. Gecode)

Constraint-Based Compiler Backend

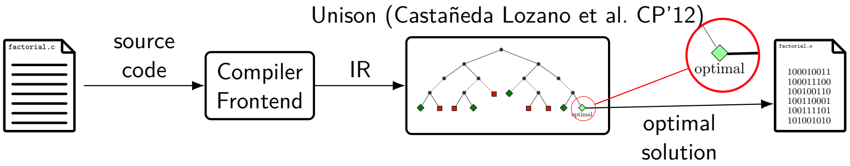


Constraint-Based Compiler Backend



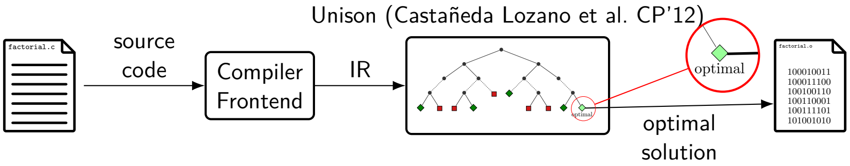
- Decision variables:

Constraint-Based Compiler Backend



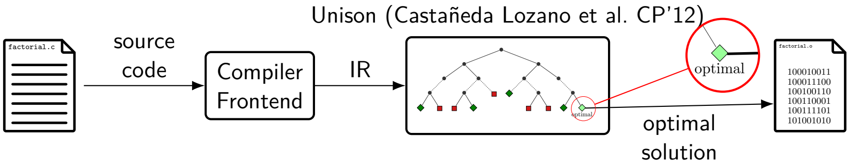
- Decision variables:
  - c: the **issue cycle** for each instruction

Constraint-Based Compiler Backend



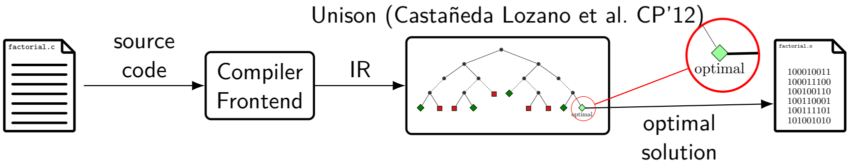
- Decision variables:
  - *c*: the **issue cycle** for each instruction
  - *m*: the **processor instruction** for each instruction

Constraint-Based Compiler Backend



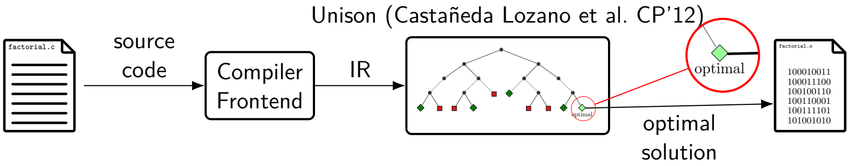
- Decision variables:
  - c: the **issue cycle** for each instruction
  - m: the **processor instruction** for each instruction
  - r: the processor **register** for each operand

Constraint-Based Compiler Backend



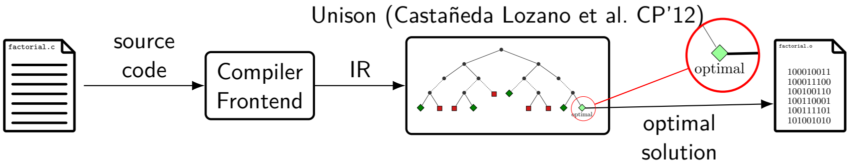
- Decision variables:
  - c: the **issue cycle** for each instruction
  - m: the **processor instruction** for each instruction
  - r: the processor **register** for each operand
- Constraints:

Constraint-Based Compiler Backend



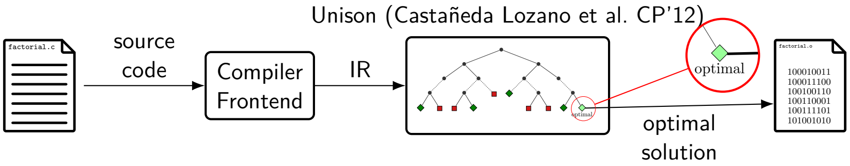
- Decision variables:
  - c: the **issue cycle** for each instruction
  - m: the **processor instruction** for each instruction
  - r: the processor **register** for each operand
- Constraints:
  - program semantics

Constraint-Based Compiler Backend



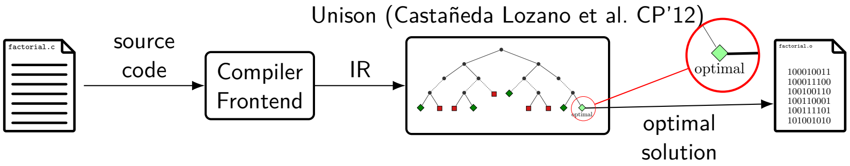
- Decision variables:
  - *c*: the **issue cycle** for each instruction
  - *m*: the **processor instruction** for each instruction
  - *r*: the processor **register** for each operand
- Constraints:
  - program semantics
  - hardware description

Constraint-Based Compiler Backend



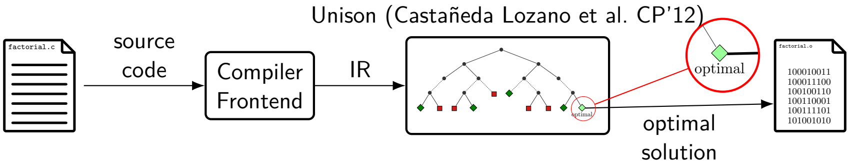
- Decision variables:
  - *c*: the **issue cycle** for each instruction
  - *m*: the **processor instruction** for each instruction
  - *r*: the processor **register** for each operand
- Constraints:
  - program semantics
  - hardware description
  - compiler transformations

Constraint-Based Compiler Backend



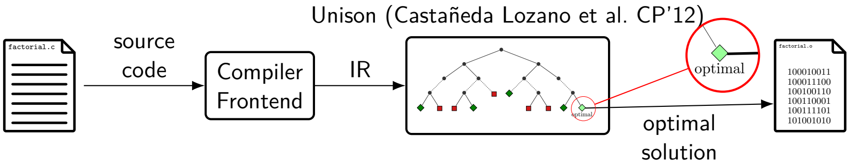
- Decision variables:
  - *c*: the **issue cycle** for each instruction
  - *m*: the **processor instruction** for each instruction
  - *r*: the processor **register** for each operand
- Constraints:
  - program semantics
  - hardware description
  - compiler transformations
- Optimization goal:

Constraint-Based Compiler Backend



- Decision variables:
  - *c*: the **issue cycle** for each instruction
  - *m*: the **processor instruction** for each instruction
  - *r*: the processor **register** for each operand
- Constraints:
  - program semantics
  - hardware description
  - compiler transformations
- Optimization goal:
  - execution time (speed)

Constraint-Based Compiler Backend



- Decision variables:
  - *c*: the **issue cycle** for each instruction
  - *m*: the **processor instruction** for each instruction
  - *r*: the processor **register** for each operand
- Constraints:
  - program semantics
  - hardware description
  - compiler transformations
- Optimization goal:
  - execution time (speed)
  - code size (size)

Example: Exclusive OR

```
uint32 xor_mem (uint32 *pub,
                uint32 *mask,
                uint32 *key) {
  uint32 sm, res;
  sm = (*sec) ^ (*mask);
  res = (*pub) ^ sm;
  return res;
}
```

Example: Exclusive OR

```c
uint32 xor_mem (uint32 *pub,
                uint32 *mask,
                uint32 *key) {
  uint32 sm, res;
  sm = (*sec) ^ (*mask);
  res = (*pub) ^ sm;
  return res;
}
```

```
9d001be8 <xor_mem>:
  lw   $a1, 0($a1)
  lw   $a2, 0($a2)
  xor  $a1, $a1, $a2
  lw   $a0, 0($a0)
  xor  $v0, $a0, $a1
  jr   $ra
  ...
```

Example: Exclusive OR

```
9d001be8 <xor_mem>:
  lw  $a1, 0($a1)
  lw  $a2, 0($a2)
  xor $a1, $a1, $a2
  lw  $a0, 0($a0)
  xor $v0, $a0, $a1
  jr  $ra
  ...
```

Example: Exclusive OR

```
9d001be8 <xor_mem>:
  lw  $a1, 0($a1)
  lw  $a2, 0($a2)
  xor $a1, $a1, $a2
  lw  $a0, 0($a0)
  xor $v0, $a0, $a1
  jr  $ra
  ...
```

```
9d001be8 <xor_mem>:
  lw  $t1, 0($a1)
  lw  $a2, 0($a2)
  xor $a1, $t1, $a2
  lw  $a0, 0($a0)
  xor $v0, $a0, $a1
  jr  $ra
  ...
```

Register allocation: `$a1` to `$t1`

Example: Exclusive OR

```
9d001be8 <xor_mem>:
  lw  $a1, 0($a1)
  lw  $a2, 0($a2)
  xor $a1, $a1, $a2
  lw  $a0, 0($a0)
  xor $v0, $a0, $a1
  jr  $ra
  ...
```

```
9d001be8 <xor_mem>:
  lw  $a2, 0($a2)
  lw  $a1, 0($a1)
  xor $a1, $a1, $a2
  lw  $a0, 0($a0)
  xor $v0, $a0, $a1
  jr  $ra
  ...
```

Instruction **issue cycle**: Swap instructions `lw $a1, 0($a1)` with `lw $a2, 0($a2)`

Example: Exclusive OR

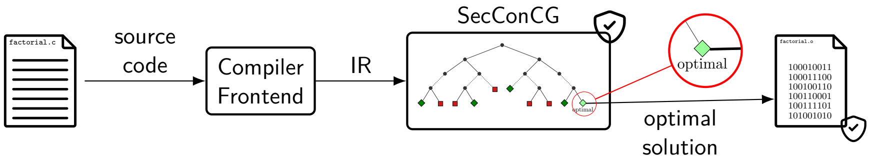
Allows generating **secure** solutions!

```
9d001be8 <xor_mem>:
  lw  $a1, 0($a1)
  lw  $a2, 0($a2)
  xor $a1, $a1, $a2
  lw  $a0, 0($a0)
  xor $v0, $a0, $a1
  jr  $ra
  ...
```

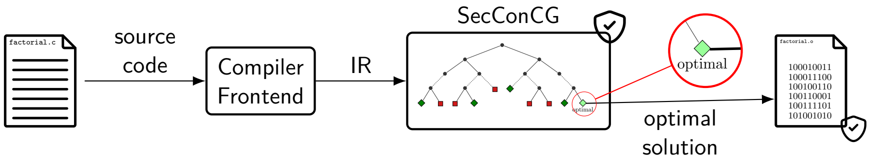
```
9d001be8 <xor_mem>:
  lw  $a2, 0($a2)
  lw  $a1, 0($a1)
  xor $a1, $a1, $a2
  lw  $a0, 0($a0)
  xor $v0, $a0, $a1
  jr  $ra
  ...
```

Contents

Secure-by-Construction Code Optimization (SecCG)

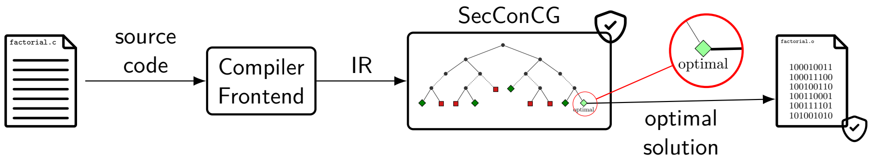


Secure-by-Construction Code Optimization (SecCG)



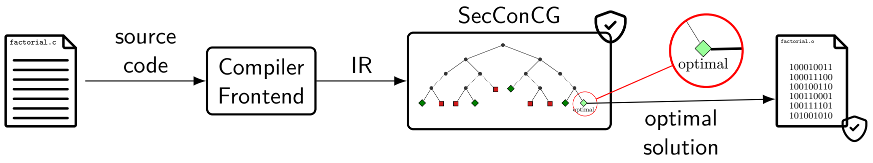
- Perform **security analysis** to extract information about the program variables

Secure-by-Construction Code Optimization (SecCG)



- Perform **security analysis** to extract information about the program variables
- Extend constraint-based compiler backend with **security constraints**

Secure-by-Construction Code Optimization (SecCG)



- Perform **security analysis** to extract information about the program variables
- Extend constraint-based compiler backend with **security constraints**
- Generate the **optimal** and **secure** solution

Register-Reuse Transitional Effects

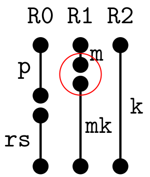
```
u32 Xor(u32 p, u32 m,
        u32 k) {
 u32 mk = m ⊕ k;
 u32 rs = mk ⊕ p;
 return rs;
}
```

Exclusive OR in C

Register-Reuse Transitional Effects

```
u32 Xor(u32 p, u32 m,
        u32 k) {
 u32 mk = m ⊕ k;
 u32 rs = mk ⊕ p;
 return rs;
}
```

Exclusive OR in C



```
R0: p, R1: m,
R2: k
 R1 = R1 ⊕ R2
 R0 = R0 ⊕ R1
```

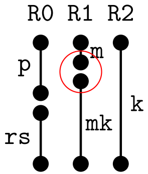
Vulnerable Register Allocation

Register `R1` changes value from `m` to `m` ⊕ `k`, which reveals information about `k`.

Register-Reuse Transitional Effects

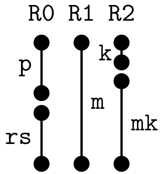
```
u32 Xor(u32 p, u32 m,
        u32 k) {
  u32 mk = m ⊕ k;
  u32 rs = mk ⊕ p;
  return rs;
}
```

Exclusive OR in C



```
R0: p, R1: m,
R2: k
  R1 = R1 ⊕ R2
  R0 = R0 ⊕ R1
```

Vulnerable Register Allocation



```
R0: p, R1: m,
R2: k
  R2 = R2 ⊕ R1
  R0 = R0 ⊕ R2
```

Secure Register Allocation

Register R2 changes value from k to $\mathtt{m} \oplus \mathtt{k}$, which does not leak secret information.

Modeling Leak-Free Code

Mitigations

- **Register** overwrite leaks

Modeling Leak-Free Code

Mitigations

- **Register** overwrite leaks

Generate Constraint Model

Modeling Leak-Free Code

Mitigations

- **Register** overwrite leaks

Generate Constraint Model

- Generate **set of pairs of variables** that should not follow each other on the same register

Modeling Leak-Free Code

Mitigations

- **Register** overwrite leaks

Generate Constraint Model

- Generate **set of pairs of variables** that should not follow each other on the same register

Proof

Modeling Leak-Free Code

Mitigations

- **Register** overwrite leaks

Generate Constraint Model

- Generate **set of pairs of variables** that should not follow each other on the same register

Proof

- The generated code does not leak secrets via **register-reuse transitions**

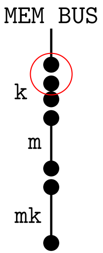
Memory-Bus Transitional Effects

```
u32 Xor(u32 *p, u32 *m,
        u32 *k, u32 *r) {
 u32 ki = *k;
 u32 mi = *m;
 u32 mk = mi ⊕ ki;
 *r = mk;
 ...
}
```

Memory Operations in C

Memory-Bus Transitional Effects

```
u32 Xor(u32 *p, u32 *m,
        u32 *k, u32 *r) {
  u32 ki = *k;
  u32 mi = *m;
  u32 mk = mi ⊕ ki;
  *r = mk;
  ...
}
```



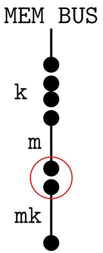
```
R0: *p, R1: *m,
R2: *k, R3: *r
  R2 = load R2
  R1 = load R1
  R2 = R2 ⊕ R1
  store R2
  ...
```

Memory Operations in C *Vulnerable Instruction Scheduling*

The first `load` transfers a secret value via the `MEM BUS`, which leaks if the initial value of `MEM BUS` is constant.

Memory-Bus Transitional Effects

```
u32 Xor(u32 *p, u32 *m,
        u32 *k, u32 *r) {
  u32 ki = *k;
  u32 mi = *m;
  u32 mk = mi ⊕ ki;
  *r = mk;
  ...
}
```



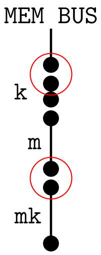
```
R0: *p, R1: *m,
R2: *k, R3: *r
  R2 = load R2
  R1 = load R1
  R2 = R2 ⊕ R1
  store R2
  ...
```

Memory Operations in C *Vulnerable Instruction Scheduling*

Memory-Bus Transitional Effects

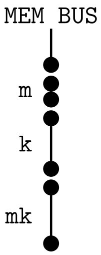
```
u32 Xor(u32 *p, u32 *m,
        u32 *k, u32 *r) {
  u32 ki = *k;
  u32 mi = *m;
  u32 mk = mi ⊕ ki;
  *r = mk;
  ...
}
```

Memory Operations in C



```
R0: *p, R1: *m,
R2: *k, R3: *r
  R2 = load R2
  R1 = load R1
  R2 = R2 ⊕ R1
  store R2
  ...
```

Vulnerable Instruction Scheduling



```
R0: *p, R1: *m,
R2: *k, R3: *r
  R1 = load R1
  R2 = load R2
  R2 = R2 ⊕ R1
  store R2
  ...
```

Secure Instruction Scheduling

Changing the first `load` instruction after loading a random value removes the leaks.

Modeling Leak-Free Code

Mitigations

- **Memory-bus** overwrite leaks

Modeling Leak-Free Code

Mitigations

- **Memory-bus** overwrite leaks

Generate Constraint Model

Modeling Leak-Free Code

Mitigations

- **Memory-bus** overwrite leaks

Generate Constraint Model

- Generate **set of pairs of memory operations** that should not follow each other

Modeling Leak-Free Code

Mitigations

- **Memory-bus** overwrite leaks

Generate Constraint Model

- Generate **set of pairs of memory operations** that should not follow each other

Proof

Modeling Leak-Free Code

Mitigations

- **Memory-bus** overwrite leaks

Generate Constraint Model

- Generate **set of pairs of memory operations** that should not follow each other

Proof

- The generated code does not leak via **memory-bus overwrite transitions**

Evaluation

Experiments

- Architecture: MIPS32 and ARM Cortex M0

Evaluation

Experiments

- Architecture: MIPS32 and ARM Cortex M0
- Benchmarks: 12 masked programs in C and C++

Evaluation

Experiments

- Architecture: MIPS32 and ARM Cortex M0
- Benchmarks: 12 masked programs in C and C++
- Portfolio: Gecode v6.2, Chuffed (Geas and OR-Tools have worse performance)

Evaluation

Experiments

- Architecture: MIPS32 and ARM Cortex M0
- Benchmarks: 12 masked programs in C and C++
- Portfolio: Gecode v6.2, Chuffed (Geas and OR-Tools have worse performance)

Results

Evaluation

Experiments

- Architecture: MIPS32 and ARM Cortex M0
- Benchmarks: 12 masked programs in C and C++
- Portfolio: Gecode v6.2, Chuffed (Geas and OR-Tools have worse performance)

Results

- **Performance Overhead**[1]: 13% overhead - 5% improvement

1 compared to non-secure optimal

Evaluation

Experiments

- Architecture: MIPS32 and ARM Cortex M0
- Benchmarks: 12 masked programs in C and C++
- Portfolio: Gecode v6.2, Chuffed (Geas and OR-Tools have worse performance)

Results

- **Performance Overhead**[1]: 13% overhead - 5% improvement
- **Performance Improvement**[2]: geometric-mean speedup 3.5 for ARM and 2.9 for MIPS32

1 compared to non-secure optimal
2 compared to secure non-optimal

Evaluation

Experiments

- Architecture: MIPS32 and ARM Cortex M0
- Benchmarks: 12 masked programs in C and C++
- Portfolio: Gecode v6.2, Chuffed (Geas and OR-Tools have worse performance)

Results

- **Performance Overhead**[1]: 13% overhead - 5% improvement
- **Performance Improvement**[2]: geometric-mean speedup 3.5 for ARM and 2.9 for MIPS32
- **Compilation Overhead**[1]: up to 50 times slowdown

1 compared to non-secure optimal
2 compared to secure non-optimal

Contents

Conclusion and Future Work

Conclusion

- Design and evaluate a combinatorial compiler approach to generate **optimized** code to mitigate

Conclusion and Future Work

Conclusion

- Design and evaluate a combinatorial compiler approach to generate **optimized** code to mitigate
  - **Register-reuse** transitional leaks

Conclusion and Future Work

Conclusion

- Design and evaluate a combinatorial compiler approach to generate **optimized** code to mitigate
  - **Register-reuse** transitional leaks
  - **Memory-bus** transitional leaks

Conclusion and Future Work

Conclusion

- Design and evaluate a combinatorial compiler approach to generate **optimized** code to mitigate
  - **Register-reuse** transitional leaks
  - **Memory-bus** transitional leaks
- The code is available: `https://github.com/romits800/seccon_experiments.git`

Conclusion and Future Work

Conclusion

- Design and evaluate a combinatorial compiler approach to generate **optimized** code to mitigate
  - **Register-reuse** transitional leaks
  - **Memory-bus** transitional leaks
- The code is available:
  `https://github.com/romits800/seccon_experiments.git`

Future Work

Conclusion and Future Work

Conclusion

- Design and evaluate a combinatorial compiler approach to generate **optimized** code to mitigate
  - **Register-reuse** transitional leaks
  - **Memory-bus** transitional leaks
- The code is available: `https://github.com/romits800/seccon_experiments.git`

Future Work

- Consider additional transitional leaks (e.g. memory overwrite)

Conclusion and Future Work

Conclusion

- Design and evaluate a combinatorial compiler approach to generate **optimized** code to mitigate
  - **Register-reuse** transitional leaks
  - **Memory-bus** transitional leaks
- The code is available:
  `https://github.com/romits800/seccon_experiments.git`

Future Work

- Consider additional transitional leaks (e.g. memory overwrite)
- Improve scalability of the approach by decomposition

Conclusion and Future Work

Conclusion

- Design and evaluate a combinatorial compiler approach to generate **optimized** code to mitigate
  - **Register-reuse** transitional leaks
  - **Memory-bus** transitional leaks
- The code is available:
  `https://github.com/romits800/seccon_experiments.git`

Future Work

- Consider additional transitional leaks (e.g. memory overwrite)
- Improve scalability of the approach by decomposition
- Evaluate the generate code on hardware

Thank you!