Contents

Introduction

Background

Approach

Conclusion and Future Work
Contents

Introduction

Background

Approach

Conclusion and Future Work
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

```
1 u32 Xor(u32 pub, u32 key) {
2     u32 t = pub ⊕ key;
3     return t;
4 }
```
PSC Attacks

insecure Xor

1 u32 Xor(u32 pub, u32 key) {
2   u32 t = pub ⊕ key;
3   return t;
4 }

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

insecure Xor

1 u32 Xor(u32 pub, u32 key) {
2     u32 t = pub \oplus key;
3     return t;
4 }  

software secure Xor

1 u32 SecXor(u32 pub, 2 u32 mask, 3 u32 key) {
4     u32 mk = mask \& key;
5     u32 t = mk \& pub;
6     return (mask, t);
7 }  

binary of secure Xor

1 u32 SecXor(u32 pub, 2 u32 mask, 3 u32 key) {
4     reg1 mask;
5     reg1 reg1 key;
6     reg1 reg1 pub;
7     return (mask, reg1);
8 }  

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

insecure Xor

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

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

```
0x00000000 ⊕ 0xBAADC0DE
0xBAADC0DE
```
### PSC Attacks

#### insecure Xor

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

#### software secure Xor

```c
u32 SecXor(u32 pub, u32 mask, u32 key) {
    u32 mk = mask ^ key;
    u32 t = mk ^ pub;
    return (mask, t);
}
```
PSC Attacks

insecure Xor

```c
1 u32 Xor(u32 pub, u32 key) {
2     u32 t = pub ⊕ key;
3     return t;
4 }
```

software secure Xor

```c
1 u32 SecXor(u32 pub,
2     u32 mask,
3     u32 key) {
4     u32 mk = mask ⊕ key;
5     u32 t = mk ⊕ pub;
6     return (mask, t);
7 }
```

binary of secure Xor

```c
1 u32 SecXor(u32 pub,
2     u32 mask,
3     u32 key) {
4     reg1 ← mask;
5     reg1 ← reg1 ⊕ key;
6     reg1 ← reg1 ⊕ pub;
7     return (mask, reg1);
8 }
```
PSC Attacks

insecure Xor

```c
u32 Xor(u32 pub, u32 key) {
    u32 t = pub \oplus key;
    return t;
}
```

software secure Xor

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

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

binary of secure Xor

```c
reg1 \leftarrow mask;
reg1 \leftarrow reg1 \oplus key;
reg1 \leftarrow reg1 \oplus pub;
return (mask, reg1);
```
PSC Attacks

insecure Xor

```c
1 u32 Xor(u32 pub, u32 key) {
2     u32 t = pub ^ key;
3     return t;
4 }
```

software secure Xor

```c
1 u32 SecXor(u32 pub,
2     u32 mask,
3     u32 key) {
4     u32 mk = mask ^ key;
5     u32 t = mk ^ pub;
6     return (mask, t);
7 }
```

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

binary of secure Xor

```c
1 u32 SecXor(u32 pub,
2     u32 mask,
3     u32 key) {
4     reg1 ← mask;
5     reg1 ← reg1 ^ key;
6     reg1 ← reg1 ^ pub;
7     return (mask, reg1);
8 }
```
PSC Attacks

insecure Xor

```c
1 u32 Xor(u32 pub, u32 key) {
2     u32 t = pub \oplus key;
3     return t;
4 }
```

software secure Xor

```c
1 u32 SecXor(u32 pub,
2     u32 mask,
3     u32 key) {
4     u32 mk = mask \oplus key;
5     u32 t = mk \oplus pub;
6     return (mask, t);
7 }
```

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

Binary of secure Xor

```c
1 u32 SecXor(u32 pub,
2     u32 mask,
3     u32 key) {
4     reg1 \leftarrow mask;
5     reg1 \leftarrow reg1 \oplus key;
6     reg1 \leftarrow reg1 \oplus pub;
7     return (mask, reg1);
8 }
```
PSC Attacks

insecure Xor

```c
1 u32 Xor(u32 pub, u32 key) {
2     u32 t = pub ^ key;
3     return t;
4 }
```

software secure Xor

```c
1 u32 SecXor(u32 pub,
2     u32 mask,
3     u32 key) {
4     u32 mk = mask ^ key;
5     u32 t = mk ^ pub;
6     return (mask, t);
7 }
```

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

binary of secure Xor

```c
1 u32 SecXor(u32 pub,
2     u32 mask,
3     u32 key) {
4     reg1 ← mask;
5     reg1 ← reg1 ^ key;
6     reg1 ← reg1 ^ pub;
7     return (mask, reg1);
8 }
```
Securing Binary Code

General-Purpose Compilation

- Focus on performance

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

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
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

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
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)

Source Code

factorial.c

Target Code

factorial.o

101001010
100111101
100110001
100100110
100011100
100010011

7 / 21
Security-Aware Code Generation (SecCG)

Source Code

▶ **secure** against PSC attacks

Target Code
Security-Aware Code Generation (SecCG)

- secure against PSC attacks
- highly optimized
Security-Aware Code Generation (SecCG)

- secure against PSC attacks
- highly optimized
- portability

Source Code

```
factorial.c
```

SecCG

Target Code

```
factorial.o
```

101001010
100111101
100110001
100100110
100011100
100010011
Contents

Introduction

Background

Approach

Conclusion and Future Work
Constraint Programming (CP)

Modeling
- Variables
- Constraints
- Objective Function

Solving
- Propagation
- Search

CP strengths:
- Global constraints
- Control over search (e.g. Gecode)
Constraint Programming (CP)

Modeling
- Variables
- Constraints
- Objective Function

Solving
- Propagation
- Search

CP strengths:
Constraint Programming (CP)

Modeling
- Variables
- Constraints
- Objective Function

Solving
- Propagation
- Search

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

Unison (Castañeda Lozano et al. CP’12)

source code → Compiler Frontend → IR

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)
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

Optimization goal:

- execution time (speed)
- code size (size)
Constraint-Based Compiler Backend

**Decision variables:**

- $c$: the **issue cycle** for each instruction
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

Unison (Castañeda Lozano et al. CP’12)
Decision variables:
- $c$: the **issue cycle** for each instruction
- $m$: the **processor instruction** for each instruction
- $r$: the processor **register** for each operand

Constraints:
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

Unison (Castañeda Lozano et al. CP’12)
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

Unison (Castañeda Lozano et al. CP’12)
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

Unison (Castañeda Lozano et al. CP’12)
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)
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)
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

```c
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;
}
```

```assembly
1  9d001be8 <xor_mem>:
2     lw  $a1, 0($a1)
3     lw  $a2, 0($a2)
4     xor $a1, $a1, $a2
5     lw  $a0, 0($a0)
6     xor $v0, $a0, $a1
7     jr  $ra
8     ...
```
Example: Exclusive OR

```
    9d001be8 <xor_mem>:
2    lw $a1, 0($a1)
3    lw $a2, 0($a2)
4    xor $a1, $a1, $a2
5    lw $a0, 0($a0)
6    xor $v0, $a0, $a1
7    jr $ra
8    ...
```
Example: Exclusive OR

1  9d001be8 <xor_mem>:
2    lw  $a1, 0($a1)
3    lw  $a2, 0($a2)
4    xor $a1, $a1, $a2
5    lw  $a0, 0($a0)
6    xor $v0, $a0, $a1
7    jr  $ra
8    ...

Register allocation: $a1 to $t1
Example: Exclusive OR

1 9d001be8 <xor_mem>:
2    lw  $a1, 0($a1)
3    lw  $a2, 0($a2)
4    xor $a1, $a1, $a2
5    lw  $a0, 0($a0)
6    xor $v0, $a0, $a1
7    jr  $ra
8    ...

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

Allows generating **secure** solutions!

```
1  9d001be8 <xor_mem>:
2    lw   $a1, 0($a1)
3    lw   $a2, 0($a2)
4    xor  $a1, $a1, $a2
5    lw   $a0, 0($a0)
6    xor  $v0, $a0, $a1
7    jr   $ra
8    ...
```
Contents

Introduction

Background

Approach

Conclusion and Future Work
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

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

Exclusive OR in C
```
Register-Reuse Transitional Effects

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

Exclusive OR in C

Vulnerable Register Allocation

Register R1 changes value from \( m \) to \( m \oplus k \), which reveals information about \( k \).
u32 Xor(u32 p, u32 m, u32 k) {
    u32 mk = m ⊕ k;
    u32 rs = mk ⊕ p;
    return rs;
}

Exclusive OR in C

Vulnerable Register Allocation

Secure Register Allocation

Register R2 changes value from $k$ to $m \oplus k$, 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

```c
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;
    ...
}

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

```c
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

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.
- Changing the first load instruction after loading a random value removes the leaks.
Memory-Bus Transitional Effects

```c
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**

- **Vulnerable Instruction Scheduling**
  - R0: *p, R1: *m, R2: *k, R3: *r
  - R2 = load R2
  - R1 = load R1
  - R2 = R2 ⊕ R1
  - store R2
  - ...

- **Secure Instruction Scheduling**
  - R0: *p, R1: *m, R2: *k, R3: *r
  - R1 = load R1
  - R2 = load R2
  - R2 = R2 ⊕ R1
  - store R2
  - ...

Changing the first load instruction after loading a random value removes the leaks.
Modeling Leak-Free Code

Mitigations

- **Memory-bus** overwrite leaks
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

Results

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

Compilation Overhead: up to 50 times slowdown compared to non-secure optimal
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)

---

Results

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

Introduction

Background

Approach

Conclusion and Future Work
Conclusion and Future Work

Conclusion

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

Future Work

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

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

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

Future Work

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

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!