UPEC-PN: Exhaustive constant time verification of low-level software using property checking

Philipp Schmitz, Johannes Müller, Christian Bartsch, Dominik Stoffel, Wolfgang Kunz
Dept. of Electrical & Computer Engineering
RPTU Kaiserslautern-Landau

MBMV, Mar. 24, 2023
Motivation

Security of low-level software

- Prevalence of timing-based side-channel attacks

- Constant time programming as countermeasure
  - How do we know whether the code is constant time?
    - Wanted: Formal method to provide guarantees
  - Is looking at the software enough?
    - Take necessary hardware detail into account
Goals:

- A scalable formal verification method to provide security guarantees for low-level constant time software.

- A modular computational model that
  - provides the necessary detail and
  - is abstract enough to scale well.
Our notion:

- Secret-independence of:
  - Control flow
  - Memory access targets
  - Execution time of individual instructions

- **Conservative** view → not all violations lead to exploits
- **Exhaustive** view → detects all possible vulnerabilities
Unique Program Execution Checking (UPEC) – DATE’19

- Originally: Formal approach for detecting Transient Execution Attacks
- Uses property checking on a bounded model with a symbolic initial state
  - Exhaustive and scalable
- 2-safety miter model
- Checks whether some protected secret data can influence the architectural state of the system
Computational model

Constraint:
other_mem_1 = other_mem_2

Memory_1

secret_data_1
other_mem_1

CPU_1

SoC_1

Memory_2

secret_data_2
other_mem_2

CPU_2

SoC_2
Background

Program Netlist (PN) – ASPDAC’13

- Formal representation of the ISA behavior for specific software
- Abstract sequential processor
- Compact computational model
  - Merge execution paths
  - Prune unreachable paths
- Result: Combinational circuit representing all possible executions
UPEC-PN

- Verification method for constant time programming
- Apply UPEC approach to PNs
  - Divide PN inputs:
    - $\Psi_i$: initial program state
    - $\Pi_p$: public program inputs
    - $\Pi_c$: confidential program inputs
  - Abstract security function $\omega(\Psi_i, \Pi_p, \Pi_c)$ models security targets

\[ \forall \Pi_c^1, \Pi_c^2: \omega(\Psi_i, \Pi_p, \Pi_c^1) = \omega(\Psi_i, \Pi_p, \Pi_c^2) \]
\[ \Pi_c^1 \rightarrow \omega(\Psi_i, \Pi_p, \Pi_c^1) \]

\[ \Psi_i, \Pi_p \rightarrow \omega(\Psi_i, \Pi_p, \Pi_c^2) \]

\[ \Pi_c^2 \rightarrow \omega(\Psi_i, \Pi_p, \Pi_c^2) \]
Constant time security targets

- **Refine** abstract security function $\omega$ to formalize security target
  - Control flow
  - Memory access
  - Individual instruction execution time

- **Remember:**
  - **Conservative** view $\rightarrow$ not all violations lead to exploits
  - **Exhaustive** view $\rightarrow$ detects all possible vulnerabilities
Microarchitectural detail

- **Observation:** Conservative view may lead to a lot of *false alerts*
  - ISA-level model does not contain enough detail to judge if it is a real vulnerability

- **Solution:** add microarchitectural detail to the PN
  - Cache model
  - Architecture-specific instruction times
Conservative proofs

Globally valid
Fast

Enhanced proofs

Architecture-specific
More complex

→ Find the sweet spot for least complexity and conservatism
RSA

- Loop-based implementation using fast exponentiation
- UPEC-PN detects secret-dependent control flow

```c
int powMod(int date, unsigned exp, int mod) {
    int result = 1;
    if (mod == 0) return 0;
    if (mod == -1) return 0;
    if (mod < -30000) return 0;
    if (mod > 30000) return 0;

    while (exp > 0) {
        if ((exp & 1) == 1) {
            result = (result * date) % mod;
        }
        date = (date * date) % mod;
        exp = exp >> 1;
    }
    return result % mod;
}
```
Case Study

RSA

Software fix for control flow dependencies

```c
int i = 32;
while (i-- > 0) {
    c_true = (exp & 1);
    __asm__("slti %[rd], %[rs1], 1" : [rd] "=r" (c_false) : [rs1] "r" (c_true));
    interm = (result * date) % mod;
    date = (date * date) % mod;
    exp = exp >> 1;
    result = c_true * interm + c_false*result;
}
```
AES

- Substitution-box-based implementation

- Key-dependent look-ups

\[
\begin{array}{cccc}
  a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\
  a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\
  a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\
  a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\
\end{array}
\]

\[
\begin{array}{cccc}
  b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\
  b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\
  b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\
  b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\
\end{array}
\]

\[S(a_{3,2})\]
AES

- UPEC-PN detects secret-dependent memory targets
  - Counterexamples pinpoint the address range
- Exploitability depends on the system
- Possible countermeasure:
  - Load the substitution box into the cache to ensure cache hits
  - Add abstract cache model to the computational model
Summary:

<table>
<thead>
<tr>
<th>Software</th>
<th>Control Flow</th>
<th>Memory Access</th>
<th>#ICs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (s)</td>
<td>Mem (MB)</td>
<td>SI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time (s)</td>
</tr>
<tr>
<td>RSA</td>
<td>43</td>
<td>8585</td>
<td>✗</td>
</tr>
<tr>
<td>Fixed RSA</td>
<td>39</td>
<td>8605</td>
<td>✓</td>
</tr>
<tr>
<td>AES</td>
<td>&lt;1</td>
<td>700</td>
<td>✓</td>
</tr>
</tbody>
</table>

Proof of concept – UPEC-PN detects the expected vulnerabilities
Conclusion

**UPEC-PN**

- Provides *architecture-independent* security guarantees
- Detects ISA-level-visible constant time violations
- Enables the consideration of necessary *microarchitectural detail*
- Is *independent* of a specific toolchain

**Future Work**

- Conduct experiments on more low-level programs
- Support for other ISAs
Thank you for your attention!

Many thanks to many collaborators!

Jörg Bormann, Lucas Deutschmann, Anna Lena Duque Antón, Mohammad Rahmani Fadiheh, Wolfgang Ecker, Jason Fung, Tobias Jauch, Dino Mehmedagić, Subhasish Mitra, Sayak Ray, Stian Gerlach Sørensen, Alex Wezel