SMT for Cryptographic & Concurrent Software Verification (Part II)

Chao Wang
ECE Dept., Virginia Tech

SAT/SMT Summer School, Semmering, Austria, July 10-12, 2014
Today’s Talk

- Performance
- Reliability
- **Security**

**Embedded Software**
- C/C++ program
- Numerical analysis
- Abstract interpretation

**Concurrent Software**
- Multi-core, multi-threaded
- Concurrency
- Symbolic predictive analysis

**Cryptographic Software**
- Security and Privacy
- Power side channel
- SMT based analysis

**VLSI Circuits**
- Finite state machine
- Boolean analysis
- Model checking

Univ. Colorado, Boulder 2000-2004

NEC Labs, Princeton 2004-2011

Virginia Tech, Blacksburg 2011-present
Cryptographic Algorithm: an example

- Plaintext is encrypted using the Secret Key stored on chip.
- System will become useless if the adversary knows the Secret Key.
Power Side-Channel Attack

Adversary analyzes the power consumption to identify the secret key

Plaintext  →  Encryption Algorithm  →  Ciphertext

0110 1001 1011 0010 1110

1001 1011 0110 0010 1110

Secret Key 010011100

Power Consumption Analysis

Identify the Secret Key
Differential Power Analysis
Useful information can be identified from the power traces through simple and differential power analysis (SPA and DPA [Kocher 1999])
Power Trace

- Further zoom in, will make you monitor power consumption of individual instructions.
Masking Countermeasure

- Masking introduces random variables to cryptographic software code to break the statistical dependence between the secret key and the power side channel.
Masking Countermeasure (linear)

- Internal nodes may leak sensitive information.

- Internal nodes no longer can leak sensitive information.

- Since \( f(x) = f(x \oplus r) \oplus f(r) \), they will have the same result but the masked software code can defend against power side channel attacks.
Masking Countermeasure (non-linear)

• No easy way to create the countermeasure
  – Often has to be redesigned from scratch
  – Labor intensive and error prone
  – No analysis tools to assess how secure it is

• For example,
  – Power consumption is logically dependent on random variables doesn’t mean it’s statistically independent of the secret data
Statistical Dependence

Question: Is $n1$ statistically independent of secret bit $i1$ now?
Motivating Example

- Outputs are logically dependent on random bits (\(r_1, r_2\))

\[
\begin{align*}
  o_1 &= k \land (r_1 \land r_2) \\
  o_2 &= k \lor (r_1 \land r_2) \\
  o_3 &= k \oplus (r_1 \land r_2) \\
  o_4 &= k \oplus (r_1 \oplus r_2)
\end{align*}
\]

- Only \(o_4\) is perfectly masked.
- **Perfect masking**: statistically independent of sensitive data (\(k\)).

<table>
<thead>
<tr>
<th>k</th>
<th>r1</th>
<th>r2</th>
<th>o1</th>
<th>o2</th>
<th>o3</th>
<th>o4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Truth table for \(o_1, o_2, o_3, o_4\)
SC Sniffer

- First automated verification method for checking if cryptographic software code is perfectly masked
Verifying Perfect Masking

- Checking if there is statistical dependence between any intermediate computation result and the sensitive data
Notation

- \((x,k)\) – the pair of plaintext and secret key
- \(F(x,k)\) – the function
- \(r\) – random variable
- \(I_1(x,k,r), \ldots, I_d(x,k,r)\) – intermediate computation results
- \(D_{x,k}\) – the probability for \(I(x,k,r)\) to be logical 1
- \(D_{x,k'}\) – the probability for \(I(x,k',r)\) to be logical 1

for all \(I(,,)\)

for all \((x,k,k')\).

\[
D_{x,k} = D_{x,k'}
\]

SMT Formula?
Notation

- \((x,k)\) – the pair of plaintext and secret key
- \(F(x,k)\) – the function
- \(r\) – random variable
- \(I_1(x,k,r), \ldots, I_d(x,k,r)\) – intermediate computation results
- \(\text{SAT}#(I(x,k,r))\) – number of solutions for \(I(x,k,r)\)
- \(\text{SAT}#(I(x,k',r))\) – number of solutions for \(I(x,k',r)\)

\[
\text{SAT}#(I(x,k,r)) = \text{SAT}#(I(x,k',r))
\]

for all \(I(,,)\)

for all \((x,k,k').\)
Notation

- \((x,k)\) – the pair of plaintext and secret key
- \(F(x,k)\) – the function
- \(r\) – random variable
- \(I_1(x,k,r),\ldots,I_d(x,k,r)\) – intermediate computation results
- \(\text{SAT}#(I(x,k,r))\) – number of solutions for \(I(x,k,r)\)
- \(\text{SAT}#(I(x,k’,r))\) – number of solutions for \(I(x,k’,r)\)

\[
\text{exists } I(,,) \text{ exists } (x,k,k'). \quad \text{SAT}#( I(x,k,r) ) \neq \text{SAT}#( I(x,k’,r) )
\]
SMT Encoding

\[ \Phi := \left( \bigwedge_{r=0}^{2^s-1} \psi^r_k \right) \land \left( \bigwedge_{r=0}^{2^s-1} \psi^r_{k'} \right) \land \psi_{b2i} \land \psi_{\text{sum}} \land \psi_{\text{diff}} \]

- \( \psi^r_k \): Function of \( I(x,k,r) \) with concrete \( r \) value (from 0 to \( 2^s-1 \)).
- \( \psi^r_{k'} \): Function of \( I(x,k',r) \) with concrete \( r \) value (from 0 to \( 2^s-1 \)).
- \( \psi_{2bi} \): Boolean to integer conversion of instruction output value.
- \( \psi_{\text{sum}} \): Encodes summation of logical 1s for each key (\( k \) and \( k' \)).
- \( \psi_{\text{diff}} \): Asserts that the two summations are unequal.
SMT Encoding Example: $I(k_1, k_2, r_1, r_2)$
Scalability Problem

• Need a SAT# solver, but we don’t have it…

• Our SMT encoding size is exponential w.r.t. the number of random variable bits.
  – $\text{Encoding } \propto 2^R$
  – Cannot scale to practical program sizes
Solution: Partitioned Verification

- Combine static code analysis and SMT-based verification
- Apply SMT solver only to small code regions, one at a time
  - Categorize the supporting variables for each instruction.
  - Eliminate unnecessary random variables.
  - Identify and extract small code regions from control-flow graph.
  - Statically analyze region and decide if it is guaranteed to be perfectly masked, and if yes, skip formal verification.
  - Formally verify region for perfect masking.
Partitioned Verification
Identifying the Smallest Code Region

- Traverse fan-in cone of the node to select code region to be verified.
- Stop enlarging the region when a fresh random variable is found.

\[
I_2 := I_1 \oplus \text{de-mask}(x, k, r) \\
:= r_{new} \oplus \text{mask}(x, k, r) \oplus \text{de-mask}(x, k, r) \\
:= r_{new} \oplus (\ldots) \\
:= r_{dummy}
\]
Experiments

• Built upon Clang/LLVM + Yices SMT solver
  – *Runs in two modes*
    • *Monolithic*
    • *Partitioned*

• Evaluated on a set of public benchmarks
  – *C code of crypto algorithms such as AES and MAC-Keccak*
## Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>LoC</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Masked Key Whitening</td>
<td>79</td>
<td>47</td>
</tr>
<tr>
<td>P2</td>
<td>De-mask and then Mask</td>
<td>67</td>
<td>31</td>
</tr>
<tr>
<td>P3</td>
<td>AES Shift Rows [2\textsuperscript{nd} order]</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>P4</td>
<td>Messerges Boolean to Arithmetic [2\textsuperscript{nd} order]</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>P5</td>
<td>Goubin Boolean to Arithmetic [2\textsuperscript{nd} order]</td>
<td>27</td>
<td>60</td>
</tr>
<tr>
<td>P6</td>
<td>Gate logic for AES S-Box (1\textsuperscript{st} impl.)</td>
<td>32</td>
<td>9</td>
</tr>
<tr>
<td>P7</td>
<td>Gate logic for AES S-Box (2\textsuperscript{nd} impl.)</td>
<td>40</td>
<td>6</td>
</tr>
<tr>
<td>P8</td>
<td>Masked Chi function MAC-Keccak (1\textsuperscript{st})</td>
<td>59</td>
<td>19</td>
</tr>
<tr>
<td>P9</td>
<td>Masked Chi function MAC-Keccak (2\textsuperscript{nd})</td>
<td>60</td>
<td>19</td>
</tr>
<tr>
<td>P10</td>
<td>Syn. Masked Chi func MAC-Keccak (1\textsuperscript{st})</td>
<td>66</td>
<td>22</td>
</tr>
<tr>
<td>P11</td>
<td>Syn. Masked Chi func MAC-Keccak (2\textsuperscript{nd})</td>
<td>66</td>
<td>22</td>
</tr>
<tr>
<td>P12</td>
<td>MAC-Keccak 512b Perfect masked</td>
<td>285k</td>
<td>128k</td>
</tr>
<tr>
<td>P13</td>
<td>MAC-Keccak 512b De-mask and mask- compiler err.</td>
<td>285k</td>
<td>128k</td>
</tr>
<tr>
<td>P14</td>
<td>MAC-Keccak 512b Not-perfect masked Chi fun. (v1)</td>
<td>285k</td>
<td>128k</td>
</tr>
<tr>
<td>P15</td>
<td>MAC-Keccak 512b Not-perfect masked Chi fun. (v2)</td>
<td>285k</td>
<td>152k</td>
</tr>
<tr>
<td>P16</td>
<td>MAC-Keccak 512b Not-perfect masked Chi fun. (v3)</td>
<td>285k</td>
<td>128k</td>
</tr>
<tr>
<td>P17</td>
<td>MAC-Keccak 512b Unmasked Pi function</td>
<td>285k</td>
<td>131k</td>
</tr>
</tbody>
</table>
### Results

**[Eldib, Wang, Schaumont, TACAS14]**

<table>
<thead>
<tr>
<th>Name</th>
<th>Sleuth</th>
<th>SC Sniffer (monolithic)</th>
<th>SC Sniffer (incremental)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>masked</td>
<td>nodes failed</td>
<td>nodes checked</td>
</tr>
<tr>
<td>P1</td>
<td>No</td>
<td>16</td>
<td>47</td>
</tr>
<tr>
<td>P2</td>
<td>No</td>
<td>8</td>
<td>31</td>
</tr>
<tr>
<td>P3</td>
<td>No</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td>P4</td>
<td>No</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>P5</td>
<td>No</td>
<td>2</td>
<td>60</td>
</tr>
<tr>
<td>P6</td>
<td>Yes</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>P7</td>
<td>Yes</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>P8</td>
<td>No</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>P9</td>
<td>Yes</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>P10</td>
<td>Yes</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>P11</td>
<td>Yes</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>P12</td>
<td>Yes</td>
<td>0</td>
<td>128k</td>
</tr>
<tr>
<td>P13</td>
<td>No</td>
<td>2560</td>
<td>128k</td>
</tr>
<tr>
<td>P14</td>
<td>Yes</td>
<td>0</td>
<td>128k</td>
</tr>
<tr>
<td>P15</td>
<td>Yes</td>
<td>0</td>
<td>152k</td>
</tr>
<tr>
<td>P16</td>
<td>No</td>
<td>512</td>
<td>128k</td>
</tr>
<tr>
<td>P17</td>
<td>No</td>
<td>4096</td>
<td>131k</td>
</tr>
</tbody>
</table>

**Sleuth:** in CHES’2013

by Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne

EPFL, TU Delft, and Univ. Lugano
Results -- scalability w.r.t. code size

- On the 1-bit version of Benchmark P1 for 1 to 10 crypto rounds.
Quantify & Evaluate Masking Strength

\[ o_1 = k \land (r_1 \land r_2) \]
\[ o_2 = k \lor (r_1 \land r_2) \]
\[ o_3 = k \oplus (r_1 \land r_2) \]
\[ o_4 = k \oplus (r_1 \oplus r_2) \]

\} \text{ many possible masking equations exist}

Which is the best?
New Notion of QMS

- Quantitative Masking Strength (QMS) reflects the amount of information leakage via side channels
  - The lower the amount of information leakage, the better
Quantifying & Evaluating the Leakage

$$\Delta qms = \max(|P_1(k_i) - P_1(k_j)|)$$

<table>
<thead>
<tr>
<th>k</th>
<th>r1</th>
<th>r2</th>
<th>o1</th>
<th>o2</th>
<th>o3</th>
<th>o4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$$\Delta qms(o1) = |0/4 - 1/4| = 0.25$$
$$\Delta qms(o2) = |1/4 - 4/4| = 0.75$$
$$\Delta qms(o3) = |1/4 - 3/4| = 0.50$$
$$\Delta qms(o4) = |2/4 - 2/4| = 0.00$$

QMS = 1 - $\Delta qms$

The higher QMS (statistical independence with secret), the better
Iteratively compute $\Delta_{qms}$

1: `COMPUTEQMS (Prog) {`
2:  $\Delta_{low} \leftarrow 0.00$
3:  $\Delta_{high} \leftarrow 1.00$
4:  `while ( $\Delta_{low} \leq \Delta_{high}$ ) {`
5:  $\Delta_{mid} \leftarrow (\Delta_{low} + \Delta_{high}) / 2.0$
6:  `if ( CHECKQMS( Prog, $\Delta_{mid}$ ) = SAT )`
7:  $\Delta_{low} \leftarrow \Delta_{mid} + 0.01$;
8:  `else`
9:  $\Delta_{high} \leftarrow \Delta_{high} - 0.01$;
10: }`
11: `return $\Delta_{low}$`
12: }`
Iteratively compute $\Delta qms$
QMS $\propto$ number of measured traces needed to guess the key ($N_{trace}$)

[ Eldib, Wang, Taha & Schaumont, DAC 2014 ]
The Big Picture

manually verified/secured systems ➔ automatically verified/secured systems
Security By Compilation

Program Synthesis

Unprotected Software code → Protected Software code

Synthesis of Masking Countermeasures against Side Channel Attacks

Hassan Eldib and Chao Wang
Department of ECE, Virginia Tech, Blacksburg, Virginia, USA
{heldib, chaowang}@vt.edu

[ Eldib & Wang, CAV14 ]
Inductive Program Synthesis

(c.f. [Eldib & Wang, FMCAD 2013])
(c.f. [Alur et. al., FMCAD 2013])
Outperformed Crypto Experts

Countermeasures for the Chi Function in MAC-Keccak (new SHA-3 crypto hashing standard by NIST)

- Handcrafted by crypto experts – 14 operations
- Generated by synthesis tool – 12 operations (more compact)

Conclusions

• SMT solver is useful in verifying and synthesizing countermeasures for cryptographic software code
  – Manual approach can be labor intensive, error prone, and may produce less efficient code

• Need solvers that support SAT#
  – Don’t know any solver that can support it yet
  – Can be a good research topic

• Need to understand the characteristics of the application to make the SMT based approach scalable
  – Partitioned Verification (XOR is commutative)
Today’s Talk

- Performance
- **Reliability**
- **Security**

**Embedded Software**
- C/C++ program
- Numerical analysis
- Abstract interpretation

**Cryptographic Software**
- Security and Privacy
- Power side channel
- SMT based analysis

**Concurrent Software**
- Multi-core, multi-threaded
- Concurrency
- Symbolic predictive analysis

**VLSI Circuits**
- Finite state machine
- Boolean analysis
- Model checking

Univ. Colorado, Boulder 2000-2004
NEC Labs, Princeton 2004-2011
Virginia Tech, Blacksburg 2011-present