Outline

- Motivation
- Verification Tool
- Verification of Modular Reduction
- Results and Observation
A Brief Introduction to Cryptography

Cryptography is a central feature of modern network computing. There are two types of cryptographic algorithms:

- **Symmetric key encryption/decryption**: Same key/algorithm for encryption and decryption. Examples include AES, SHA.
- **Public key encryption/decryption**: Different keys for encryption and decryption. Examples include RSA, PNG.

Public key encryption is based on modular arithmetic such as:

- **Modular reduction**: $A \mod N$
- **Modular inverse**: $A^{-1} \mod N$
- **Modular exponentiation**: $A^B \mod N$

Montgomery multiplier accelerates $A \times B \mod N$ computation.
A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG
A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG
- Public key encryption is based on modular arithmetic such as
  - Modular reduction $A \mod N$
A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG
- Public key encryption is based on modular arithmetic such as
  - Modular reduction $A \mod N$
  - Modular inverse $A^{-1} \mod N$
A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithms for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG
- Public key encryption is based on modular arithmetic such as
  - Modular reduction $A \mod N$
  - Modular inverse $A^{-1} \mod N$
  - Modular exponentiation $A^B \mod N$
A Brief Introduction to Cryptography

- Cryptography is a central feature of modern network computing.
- There are two types of cryptographic algorithms
  - Symmetric key encryption/decryption
    - Same key/algorithm for encryption and decryption
    - e.g. AES, SHA
  - Public key encryption/decryption
    - Different keys for encryption and decryption
    - e.g. RSA, PNG
- Public key encryption is based on modular arithmetic such as
  - Modular reduction $A \mod N$
  - Modular inverse $A^{-1} \mod N$
  - Modular exponentiation $A^B \mod N$
  - Montgomery multiplier accelerates $A^B \mod N$ computation.
On-chip Hardware Accelerator for Modular Reduction

- **Hardware Accelerator**
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator
On-chip Hardware Accelerator for Modular Reduction

- Hardware Accelerator
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator
- We worked on an asymmetric math function accelerator
On-chip Hardware Accelerator for Modular Reduction

■ Hardware Accelerator
  ■ On-chip co-processor that frees up CPU cycles
  ■ Tuned for certain tasks, often computationally expensive ones.
  ■ e.g. Graphic accelerator. Encryption accelerator

■ We worked on an *asymmetric math function accelerator*
  ■ Performs modular math for public key encryption.
On-chip Hardware Accelerator for Modular Reduction

- **Hardware Accelerator**
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator

- **We worked on an asymmetric math function accelerator**
  - Performs modular math for public key encryption.
  - Used for encryption acceleration.
On-chip Hardware Accelerator for Modular Reduction

- **Hardware Accelerator**
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator

- **We worked on an asymmetric math function accelerator**
  - Performs modular math for public key encryption.
  - Used for encryption acceleration.
  - Takes up to 4096-bit operands
On-chip Hardware Accelerator for Modular Reduction

- **Hardware Accelerator**
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator

- **We worked on an asymmetric math function accelerator**
  - Performs modular math for public key encryption.
  - Used for encryption acceleration.
  - Takes up to 4096-bit operands
  - Long delays: Thousands of clock cycles for a single operation
On-chip Hardware Accelerator for Modular Reduction

- **Hardware Accelerator**
  - On-chip co-processor that frees up CPU cycles
  - Tuned for certain tasks, often computationally expensive ones.
  - e.g. Graphic accelerator. Encryption accelerator

- **We worked on an *asymmetric math function accelerator***
  - Performs modular math for public key encryption.
  - Used for encryption acceleration.
  - Takes up to 4096-bit operands
  - Long delays: Thousands of clock cycles for a single operation
  - Implemented as a finite-state machine.
Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.
Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.

Traditional verification techniques have problems
- Simulation is too slow to provide a decent coverage.
Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.

Traditional verification techniques have problems:
- Simulation is too slow to provide a decent coverage.
- Even post-silicon testing is slow because of slow reference model computation by software.
Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.

Traditional verification techniques have problems

- Simulation is too slow to provide a decent coverage.
- Even post-silicon testing is slow because of slow reference model computation by software.
- Bit-level model-checking does not scale to thousands of cycles.
Why Is the Accelerator Difficult To Verify?

Verification is a challenge because of the vast state-space due to wide operands and long latency.

Traditional verification techniques have problems:
- Simulation is too slow to provide a decent coverage.
- Even post-silicon testing is slow because of slow reference model computation by software.
- Bit-level model-checking does not scale to thousands of cycles.
- Very time-consuming to analyze implementation details with a theorem prover.
A hybrid verification tool is a combination of a model checker and a theorem prover.

- e.g. Intel Forte based on symbolic trajectory evaluation.
A hybrid verification tool is a combination of a model checker and a theorem prover.

- e.g. Intel Forte based on symbolic trajectory evaluation.

We believe the full potential of hybrid verification tools have not been utilized because:

- Model checker is not tuned for this kind of proofs.
- Theorem prover is hard-to-use and time-consuming for many engineers.
Our tool ACL2SIX is a combination of
- IBM SixthSense Formal Verification Tool (Model Checker)
- ACL2 Theorem Prover
ACL2SIX

- Our tool **ACL2SIX** is a combination of
  - IBM SixthSense Formal Verification Tool (Model Checker)
  - ACL2 Theorem Prover
- ACL2SIX directly works on hardware given in HDL.
  - A quick translation of properties, not of hardware HDL.
  - The theorem prover does not deal with low-level details of hardware. The model checker abstracts them away.
Hybrid Verification of a Hardware Modular Reduction Engine

ACL2SIX Platform Data Flow

- User Inputs
  - Property Compilation
- Translated Property
- Verification Driver
- Hardware VHDL
- SixthSense
  - Complete Proof
  - Verified Property
- Success
  - Counter-Example Waveform
  - Fail
ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

(defthm adder-output
  (implies (natp n)
    (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
      (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
           (vhdl-sigvec (adder) "B" (0 31) n)))))

:hints ("goal" :clause-processor
         (:function acl2six :hint '(:cycle-var n))))
ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

(defun adder-output
  (implies (natp n)
    (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
           (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
                (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints ("goal" :clause-processor
            (:function acl2six :hint '(:cycle-var n))))

- Bit vectors are accessed by vhdl-sigvec with the syntax:
  (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

(defthm adder-output
  (implies (natp n)
    (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
      (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
        (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints ("goal" :clause-processor
    (:function acl2six :hint '(:cycle-var n))))

- Bit vectors are accessed by vhdl-sigvec with the syntax:
  (vhdl-sigvec DUT vector name field clock cycle)
ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

(defthm adder-output
  (implies (natp n)
           (equal (vhdl-sigvec (adder) ”SUM” (0 31) (+ n 2))
                  (bv+ (vhdl-sigvec (adder) ”A” (0 31) n)
                       (vhdl-sigvec (adder) ”B” (0 31) n)))))

:hints ("goal" :clause-processor
           (:function acl2six :hint '(:cycle-var n))))

- Bit vectors are accessed by vhdl-sigvec with the syntax:
  (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

(defun adder-output
  (implies (natp n)
    (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
      (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
            (vhdl-sigvec (adder) "B" (0 31) n))))

:hints ("goal" :clause-processor
            (:function acl2six :hint '((:cycle-var n))))

- Bit vectors are accessed by vhdl-sigvec with the syntax:
  (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

(defthm adder-output
  (implies (natp n)
    (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
      (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
        (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints ("goal":clause-processor
     (:function acl2six :hint'((:cycle-var n)))))

- Bit vectors are accessed by vhdl-sigvec with the syntax:
  (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

(defthm adder-output
  (implies (natp n)
    (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
       (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
       (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints ("goal" :clause-processor
           (:function acl2six :hint '(:cycle-var n))))

- Bit vectors are accessed by vhdl-sigvec with the syntax:
  (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
- Clock cycle is given by (variable + constant delay)
ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

(defthm adder-output
  (implies (natp n)
    (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
      (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
           (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints (:goal :clause-processor
           (:function acl2six :hint '(:cycle-var n)))))

- Bit vectors are accessed by vhdl-sigvec with the syntax:
  (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
- Clock cycle is given by (variable + constant delay)
- Pre-defined and user-defined bit-vector functions can be used.
ACL2SIX Theorem Example

Theorem to test the output of a 2-stage 32-bit adder.

(defthm adder-output
  (implies (natp n)
    (equal (vhdl-sigvec (adder) "SUM" (0 31) (+ n 2))
        (bv+ (vhdl-sigvec (adder) "A" (0 31) n)
              (vhdl-sigvec (adder) "B" (0 31) n))))
  :hints ("goal" :clause-processor
            (:function acl2six :hint '(:cycle-var n)))))

- Bit vectors are accessed by vhdl-sigvec with the syntax:
  (vhdl-sigvec ⟨DUT⟩ ⟨vector name⟩ ⟨field⟩ ⟨clock cycle⟩)
- Clock cycle is given by (variable + constant delay)
- Pre-defined and user-defined bit-vector functions can be used.
- Directive to call SixthSense from ACL2
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$  
  \[
  A = 00011100_2 \\
  N = 00000101_2
  \]
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  $A = 00011100_2$
  
  $N = 00000101_2$
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  \[
  A = 00011100_2 \\
  N = 00000101_2
  \]
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  $A = 00011100_2$
  
  $N = 00001010_2$
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  $A = 00011100_2$
  $N = 00010100_2$
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  $A = 00011100_2$
  
  $N = 00010100_2$

```
\begin{center}
\begin{tikzpicture}[node distance = 2cm, auto]

  \node (S0) [state] {$S_0$};
  \node (S1) [state] at (1,2) {$S_1$};
  \node (S2) [state] at (1,4) {$S_2$};
  \node (S3) [state] at (1,6) {$S_3$};
  \node (S4) [state] at (1,8) {$S_4$};

  \path[->]
  (S0) edge node [above] {Input $A_0$ and $N_0$} (S1)
  (S1) edge node [above] {Shift amt calculation} (S2)
  (S2) edge node [below] {Align Data} (S3)
  (S3) edge node [left] {If $N > A$} (S4)
  (S4) edge node [right] {$A = A_0 \mod N_0$} (S0);
\end{tikzpicture}
\end{center}
```
Motivation
Verification Tool
Verification of Modular Reduction
Results and Observation

Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  - $A = 00001000_2$
  - $N = 00010100_2$

![ FSM Diagram ]

- $S_0$: Input $A_0$ and $N_0$
- $S_1$: Shift amt calculation
- $S_2$: Align Data
- $S_3$: Subtract or add while shifting
- $S_4$: $A = A_0 \mod N_0$
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  $$A = 00001000_2$$
  $$N = 00001010_2$$

Diagram:

- $S_0$: Input $A_0$ and $N_0$
- $S_1$: Align Data
- $S_2$: Shift amt calculation
- $S_3$: Subtract or add while shifting
- $S_4$: $A = A_0 \mod N_0$
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute \( A_0 \mod N_0 \).

**Example:** compute \( 28 \mod 5 \)

\[
\begin{align*}
A &= 11111110_2 \\
N &= 00001010_2
\end{align*}
\]
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  \[ A = 1111110_2 \]
  \[ N = 00000101_2 \]
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

Example: compute $28 \mod 5$

$A = 00000011_2$
$N = 00000101_2$

If $N > A$

Shift amt calculation

Align Data

Subtract or add while shifting

$A = A_0 \mod N_0$
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  \[
  A = 00000011_2 \\
  N = 00000101_2
  \]

Align Data

Subtract or add while shifting

If $N > A$

Shift amt calculation

Input $A_0$ and $N_0$
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  $A = 00000011_2$
  
  $N = 00000101_2$

- Actual Operands are very long.
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  $A = 00000011_2$
  
  $N = 00000101_2$

- Actual Operands are very long.
- Many arithmetic operations are repeated in each transition.
Simplified Modular Reduction Engine

Modular reduction engine FSM to compute $A_0 \mod N_0$.

- Example: compute $28 \mod 5$
  
  $A = 00000011_2$
  
  $N = 00000101_2$

- Actual Operands are very long.
- Many arithmetic operations are repeated in each transition.
- State transition takes fixed but long clock cycles.

Sawada, Sandon, Paruthi, Baumgartner, Case, Mony
IBM Corporation
Hybrid Verification of a Hardware Modular Reduction Engine
Overall Approach to Verifying a State Transition Machine

- Use a divide-and-conquer approach.
  - Model checker is used to verify properties over each state transition.
  - Theorem prover is used to combine verified properties to form a complete proof, and also reason about high-level math.
Overall Approach to Verifying a State Transition Machine

- Use a divide-and-conquer approach.
  - Model checker is used to verify properties over each state transition.
  - Theorem prover is used to combine verified properties to form a complete proof, and also reason about high-level math.
- Make the model checker to work on bigger, more abstract sub-problems.
  - Hide the hardware details from the theorem prover.
  - Theorem prover requires smaller steps to create a proof.
How Should We Write Properties over State Transition?

- Typical state transition with pre-condition $P_i$ and post-condition $P_{i+1}$:
  \[ P_i(n) \implies P_{i+1}(n + \Delta_i) \]

- $\Delta_i$ is typically constant over 10 but less than 100.
How Should We Write Properties over State Transition?

- Typical state transition with pre-condition $P_i$ and post-condition $P_{i+1}$:
  \[ P_i(n) \implies P_{i+1}(n + \Delta_i) \]
- $\Delta_i$ is typically constant over 10 but less than 100.
- Actual conditions are written at high-level.
  - e.g. Multi-word subtraction is simply written as $A - N$ in $P_i$.
  - The hardware may repeat multiple subtractions over discontinuous data.
How Should We Write Properties over State Transition?

- Typical state transition with pre-condition $P_i$ and post-condition $P_{i+1}$:
  \[ P_i(n) \implies P_{i+1}(n + \Delta_i) \]
  - $\Delta_i$ is typically constant over 10 but less than 100.

- Actual conditions are written at high-level.
  - e.g. Multi-word subtraction is simply written as $A - N$ in $P_i$. The hardware may repeat multiple subtractions over discontinuous data.

- Frequently, we need to add global and state invariants to prove
  \[ (\text{inv}(n) \land \text{cond}_i(n) \land P_i(n)) \implies P_{i+1}(n + \Delta_i) \]
  - Invariant definitions are in VHDL and hidden from theorem prover.
Algorithm to verify $P_i(n) \implies P_{i+1}(n + \Delta_i)$

**Algorithm**

1. Convert $P_i(n) \implies P_{i+1}(n + \Delta_i)$ to a circuit and combine it with DUT and the driver. Result is $Q_i(n)$. 
Algorithm to verify \( P_i(n) \implies P_{i+1}(n + \Delta_i) \)

**Algorithm**

1. Convert \( P_i(n) \implies P_{i+1}(n + \Delta_i) \) to a circuit and combine it with DUT and the driver. Result is \( Q_i(n) \).

2. Simplify \( Q_i(n) \) by a number of combinational and sequential logic reduction algorithms. Result is \( Q'_i(n) \). If \( Q'_i(n) = T \), return.


Algorithm to verify $P_i(n) \implies P_{i+1}(n + \Delta_i)$

Algorithm

1. Convert $P_i(n) \implies P_{i+1}(n + \Delta_i)$ to a circuit and combine it with DUT and the driver. Result is $Q_i(n)$.

2. Simplify $Q_i(n)$ by a number of combinational and sequential logic reduction algorithms. Result is $Q_i'(n)$. If $Q_i'(n) = T$, return.

3. Prove $Q_i'(n)$ by $k$-induction. Base cases are proved by BMC. Inductive step is proved:
   $Q_i(n) \land Q_i(n + 1) \land \cdots \land Q_i(n + k - 1) \implies Q_i(n + k)$. 

Sawada, Sandon, Paruthi, Baumgartner, Case, Mony

IBM Corporation

Hybrid Verification of a Hardware Modular Reduction Engine
Algorithm to verify $P_i(n) \implies P_{i+1}(n + \Delta_i)$

**Algorithm**

1. Convert $P_i(n) \implies P_{i+1}(n + \Delta_i)$ to a circuit and combine it with DUT and the driver. Result is $Q_i(n)$.

2. Simplify $Q_i(n)$ by a number of combinational and sequential logic reduction algorithms. Result is $Q'_i(n)$. If $Q'_i(n) = T$, return.

3. Prove $Q'_i(n)$ by k-induction. Base cases are proved by BMC. Inductive step is proved:

   $$Q_i(n) \land Q_i(n + 1) \land \cdots \land Q_i(n + k - 1) \implies Q_i(n + k).$$

4. Increase $k$ and repeat Step 3.
Algorithm to verify $P_i(n) \implies P_{i+1}(n + \Delta_i)$

**Algorithm**

1. Convert $P_i(n) \implies P_{i+1}(n + \Delta_i)$ to a circuit and combine it with DUT and the driver. Result is $Q_i(n)$.

2. Simplify $Q_i(n)$ by a number of combinational and sequential logic reduction algorithms. Result is $Q'_i(n)$. If $Q'_i(n) = T$, return.

3. Prove $Q'_i(n)$ by $k$-induction. Base cases are proved by BMC. Inductive step is proved:
   
   $Q_i(n) \land Q_i(n + 1) \land \cdots \land Q_i(n + k - 1) \implies Q_i(n + k)$.

4. Increase $k$ and repeat Step 3.

Step 1 is performed by the theorem prover. Step 2-4 by the model checker.
Often an induction proof fails and a counter-example helps debugging.
Often an induction proof fails and a counter-example helps debugging.

Counter-example generation is difficult for transformation-based verification tool like SixthSense.

- An inductive counter-example does not start with an initial state.
- Some information is lost during transformation.
Generation of Counter-Examples for Induction Proof

- Often an induction proof fails and a counter-example helps debugging.
- Counter-example generation is difficult for transformation-based verification tool like SixthSense.
  - An inductive counter-example does not start with an initial state.
  - Some information is lost during transformation.
- Implemented a trace lifting to reflect true root cause of induction failure.
## Verification Results of Modular Reduction

<table>
<thead>
<tr>
<th>Data Width</th>
<th>56-bit</th>
<th>256-bit</th>
<th>384-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>10442s</td>
<td>20646s</td>
<td>37607s</td>
<td>98199s</td>
</tr>
<tr>
<td>Theorem Prover Time</td>
<td>257s</td>
<td>289s</td>
<td>474s</td>
<td>1690s</td>
</tr>
<tr>
<td>Property Check Time</td>
<td>10188s</td>
<td>20261s</td>
<td>37139s</td>
<td>97012s</td>
</tr>
<tr>
<td>Avg. Time per Prop.</td>
<td>118s</td>
<td>151s</td>
<td>223s</td>
<td>489s</td>
</tr>
<tr>
<td>Max Time per Prop.</td>
<td>138s</td>
<td>368s</td>
<td>1232s</td>
<td>3456s</td>
</tr>
</tbody>
</table>

- We finished modular reduction proof up to 512-bit.

1024-bit operation has properties that time-out in 24 hours. Individual property time increases rapidly as both state transition delay and input data increase. Most time spent in the model checker.
Verification Results of Modular Reduction

<table>
<thead>
<tr>
<th>Data Width</th>
<th>56-bit</th>
<th>256-bit</th>
<th>384-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>10442s</td>
<td>20646s</td>
<td>37607s</td>
<td>98199s</td>
</tr>
<tr>
<td>Theorem Prover Time</td>
<td>257s</td>
<td>289s</td>
<td>474s</td>
<td>1690s</td>
</tr>
<tr>
<td>Property Check Time</td>
<td>10188s</td>
<td>20261s</td>
<td>37139s</td>
<td>97012s</td>
</tr>
<tr>
<td>Avg. Time per Prop.</td>
<td>118s</td>
<td>151s</td>
<td>223s</td>
<td>489s</td>
</tr>
<tr>
<td>Max Time per Prop.</td>
<td>138s</td>
<td>368s</td>
<td>1232s</td>
<td>3456s</td>
</tr>
</tbody>
</table>

- We finished modular reduction proof up to 512-bit.
- 1024-bit operation has properties that time-out in 24 hours.
### Verification Results of Modular Reduction

<table>
<thead>
<tr>
<th>Data Width</th>
<th>56-bit</th>
<th>256-bit</th>
<th>384-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>10442s</td>
<td>20646s</td>
<td>37607s</td>
<td>98199s</td>
</tr>
<tr>
<td>Theorem Prover Time</td>
<td>257s</td>
<td>289s</td>
<td>474s</td>
<td>1690s</td>
</tr>
<tr>
<td>Property Check Time</td>
<td>10188s</td>
<td>20261s</td>
<td>37139s</td>
<td>97012s</td>
</tr>
<tr>
<td>Avg. Time per Prop.</td>
<td>118s</td>
<td>151s</td>
<td>223s</td>
<td>489s</td>
</tr>
<tr>
<td>Max Time per Prop.</td>
<td>138s</td>
<td>368s</td>
<td>1232s</td>
<td>3456s</td>
</tr>
</tbody>
</table>

- We finished modular reduction proof up to 512-bit.
- 1024-bit operation has properties that time-out in 24 hours.
- Individual property time increases rapidly as both state transition delay and input data increase.
Verification Results of Modular Reduction

<table>
<thead>
<tr>
<th>Data Width</th>
<th>56-bit</th>
<th>256-bit</th>
<th>384-bit</th>
<th>512-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>10442s</td>
<td>20646s</td>
<td>37607s</td>
<td>98199s</td>
</tr>
<tr>
<td>Theorem Prover Time</td>
<td>257s</td>
<td>289s</td>
<td>474s</td>
<td>1690s</td>
</tr>
<tr>
<td>Property Check Time</td>
<td>10188s</td>
<td>20261s</td>
<td>37139s</td>
<td>97012s</td>
</tr>
<tr>
<td>Avg. Time per Prop.</td>
<td>118s</td>
<td>151s</td>
<td>223s</td>
<td>489s</td>
</tr>
<tr>
<td>Max Time per Prop.</td>
<td>138s</td>
<td>368s</td>
<td>1232s</td>
<td>3456s</td>
</tr>
</tbody>
</table>

- We finished modular reduction proof up to 512-bit.
- 1024-bit operation has properties that time-out in 24 hours.
- Individual property time increases rapidly as both state transition delay and input data increase.
- Most time spent in the model checker.
Conclusion

- We verified a number of modular operations.
  - Modular reduction, modular addition and subtraction.
  - Montgomery multiplier
We verified a number of modular operations.

- Modular reduction, modular addition and subtraction.
- Montgomery multiplier

Analysis of modular inverse uncovered an overflow problem.
We verified a number of modular operations.
- Modular reduction, modular addition and subtraction.
- Montgomery multiplier

Analysis of modular inverse uncovered an overflow problem.
The key is to use a powerful model checker to verify a larger sub-problems. Reduced theorem proving effort.
Conclusion

- We verified a number of modular operations.
  - Modular reduction, modular addition and subtraction.
  - Montgomery multiplier
- Analysis of modular inverse uncovered an overflow problem.
- The key is to use a powerful model checker to verify a larger sub-problems. Reduced theorem proving effort.
- Still full 4096-bits operation is hard to verify. Need to improve model checker for this type of proof.
Conclusion

- We verified a number of modular operations.
  - Modular reduction, modular addition and subtraction.
  - Montgomery multiplier
- Analysis of modular inverse uncovered an overflow problem.
- The key is to use a powerful model checker to verify a larger sub-problems. Reduced theorem proving effort.
- Still full 4096-bits operation is hard to verify. Need to improve model checker for this type of proof.
- Theorem proving is still a bottleneck to apply in an industrial setting. Need more automation or more productivity.