Hardware Model Checking of Security Properties

Mridul Mahajan (IIM2017501)

Advisor: (Late) Prof. Pramod Subramanyan (IIT Kanpur)

Table of Contents

1. Introduction
2. Methodology
3. Experiments and Results
4. Conclusion and Future Work

Table of Contents

1. Introduction
2. Methodology
3. Experiments and Results
4. Conclusion and Future Work

Introduction

- Model checking as a verification technique.

Introduction

- Model checking as a verification technique.
- Logics for specification.

Introduction

- Model checking as a verification technique.
- Logics for specification.
- Logics specifically designed for security properties.

Introduction

- Model checking as a verification technique.
- Logics for specification.
- Logics specifically designed for security properties.
- Contributions:

Introduction

- Model checking as a verification technique.
- Logics for specification.
- Logics specifically designed for security properties.
- Contributions:
  - Augmented ABC (A System for Sequential Synthesis and Verification) with the ability to perform model checking of 2-safety properties.

Introduction

- Model checking as a verification technique.
- Logics for specification.
- Logics specifically designed for security properties.
- Contributions:
  - Augmented ABC (A System for Sequential Synthesis and Verification) with the ability to perform model checking of 2-safety properties.
  - Some suggestions to improve Lazy Self-Composition.

Table of Contents

1 Introduction

2 Methodology

3 Experiments and Results

4 Conclusion and Future Work

Model Checking

- Models the design to be verified as a transition system.

Model Checking

- Models the design to be verified as a transition system.
- Checks temporal properties on the modeled FSM.

Model Checking

- Models the design to be verified as a transition system.
- Checks temporal properties on the modeled FSM.
- Types of properties:

Model Checking

- Models the design to be verified as a transition system.
- Checks temporal properties on the modeled FSM.
- Types of properties:
  - Safety Properties: What should not happen.

Model Checking

- Models the design to be verified as a transition system.
- Checks temporal properties on the modeled FSM.
- Types of properties:
    - Safety Properties: What should not happen.
    - Liveness Properties: What should eventually happen.

Model Checking

- Models the design to be verified as a transition system.
- Checks temporal properties on the modeled FSM.
- Types of properties:
    - Safety Properties: What should not happen.
    - Liveness Properties: What should eventually happen.
- Caution: Cannot determine if the system is correct or not!

Model Checking

- Models the design to be verified as a transition system.
- Checks temporal properties on the modeled FSM.
- Types of properties:
  - Safety Properties: What should not happen.
  - Liveness Properties: What should eventually happen.
- Caution: Cannot determine if the system is correct or not!
- A simple model checking algorithm: Use a graph-search algorithm.

Model Checking

- Represent the transition system using a Kripke structure.

Model Checking

- Represent the transition system using a Kripke structure.
- $M = (S, I, T, L)$, $I \subseteq S$, $T \subseteq S \times S$, $L : S \rightarrow P(A)$.

Model Checking

- Represent the transition system using a Kripke structure.
- $M = (S, I, T, L)$, $I \subseteq S$, $T \subseteq S \times S$, $L : S \rightarrow P(A)$.
- $\pi = (s_0, s_1, \ldots)$, such that $T(s_i, s_{i+1})$ should hold for $0 \leq i < |\pi| - 1$.

Model Checking

- Represent the transition system using a Kripke structure.
- $M = (S, I, T, L)$, $I \subseteq S$, $T \subseteq S \times S$, $L : S \rightarrow P(A)$.
- $\pi = (s_0, s_1, ...)$, such that $T(s_i, s_{i+1})$ should hold for $0 \leq i < |\pi| - 1$.
- Initialized path: $I(s_0)$ holds True.

Linear-time Temporal Logic (LTL)

- Extends classical logic by introducing temporal operators.

Linear-time Temporal Logic (LTL)

- Extends classical logic by introducing temporal operators.
- Some of these operators are:

Linear-time Temporal Logic (LTL)

- Extends classical logic by introducing temporal operators.
- Some of these operators are:
  - The Next operator X.

Linear-time Temporal Logic (LTL)

- Extends classical logic by introducing temporal operators.
- Some of these operators are:
  - The Next operator X.
  - The Globally operator G.

Linear-time Temporal Logic (LTL)

- Extends classical logic by introducing temporal operators.
- Some of these operators are:
  - The Next operator X.
  - The Globally operator G.
  - The Finally operator F.

Linear-time Temporal Logic (LTL)

- Extends classical logic by introducing temporal operators.
- Some of these operators are:
  - The Next operator X.
  - The Globally operator G.
  - The Finally operator F.
- The duality property: $\neg \mathrm{F}p$ is equivalent to $\mathrm{G}\neg p$.

Hyperproperties and HyperLTL

- Trace property: A set of infinite traces.

Hyperproperties and HyperLTL

- Trace property: A set of infinite traces.
- $\mathrm{T} \models \mathrm{H} \triangleq \mathrm{T} \in \mathrm{H}$, where H is a hyperproperty and T is a set of traces.

Hyperproperties and HyperLTL

- Trace property: A set of infinite traces.
- $\mathrm{T} \models \mathrm{H} \triangleq \mathrm{T} \in \mathrm{H}$, where H is a hyperproperty and T is a set of traces.
- HyperLTL introduces path quantifiers on top of LTL.

Observational Determinism and Self-Composition

- $\forall_{\pi_1}\forall_{\pi_2} G(a_{\pi_1} == a_{\pi_2}) \Rightarrow G(b_{\pi_1} == b_{\pi_2})$

Observational Determinism and Self-Composition

- $\forall_{\pi_1} \forall_{\pi_2} G(a_{\pi_1} == a_{\pi_2}) \Rightarrow G(b_{\pi_1} == b_{\pi_2})$
- Can be used to show that an attacker's observations are a deterministic function of their inputs.

Observational Determinism and Self-Composition

- $\forall_{\pi_1}\forall_{\pi_2} G(a_{\pi_1} == a_{\pi_2}) \Rightarrow G(b_{\pi_1} == b_{\pi_2})$
- Can be used to show that an attacker's observations are a deterministic function of their inputs.
- Self-Composition reduces this to $G(a_1 == a_2) \Rightarrow G(b_1 == b_2)$.

Self-Composition on Circuits

- And-Inverted Graph (AIG):
  - A directed acyclic graph (DAG).
  - Each node is a two-input AND gate.
  - Each fan-in or fan-out edge has an optional attribute to indicate inverter on that edge.
  - Manipulating an AIG is not a trivial but a tricky task.

Procedure

- A high level description:
  1. Read the network twice.
  2. Strash the networks.
  3. Start from the principal inputs (PIs).
  4. Topologically sort the latches in the original AIGs and build a part of the self-composed AIG.
  5. Topologically sort the AND-gates in the original AIGs and build a part of the self-composed AIG.
  6. Connect the fan-ins and fan-outs of the latches.
  7. Encode the assertions and assumptions.

Encoding assertions and assumptions

- In ABC, we can only check properties of the form: $G(a_1 \wedge a_2 \wedge \ldots \wedge a_n) \Rightarrow G(b_1 \wedge b_2 \wedge \ldots \wedge b_m)$.

Encoding assertions and assumptions

- In ABC, we can only check properties of the form: $G(a_1 \wedge a_2 \wedge ... \wedge a_n) \Rightarrow G(b_1 \wedge b_2 \wedge ... \wedge b_m)$.
- Suppose we have a fomula of the following form: $G(a \Rightarrow X(b))$.

Encoding assertions and assumptions

- In ABC, we can only check properties of the form: $\mathrm{G}(a_1 \wedge a_2 \wedge \ldots \wedge a_n) \Rightarrow \mathrm{G}(b_1 \wedge b_2 \wedge \ldots \wedge b_m)$.
- Suppose we have a fomula of the following form: $\mathrm{G}(a \Rightarrow \mathrm{X}(b))$.
- Construct a temporal tester.

Encoding assertions and assumptions

- In ABC, we can only check properties of the form: $\mathrm{G}(a_1 \wedge a_2 \wedge \ldots \wedge a_n) \Rightarrow \mathrm{G}(b_1 \wedge b_2 \wedge \ldots \wedge b_m)$.
- Suppose we have a fomula of the following form: $\mathrm{G}(a \Rightarrow \mathrm{X}(b))$.
- Construct a temporal tester.
- $a_{prev} = DFF(a)$.

Encoding assertions and assumptions

- In ABC, we can only check properties of the form: $\mathrm{G}(a_1 \wedge a_2 \wedge \ldots \wedge a_n) \Rightarrow \mathrm{G}(b_1 \wedge b_2 \wedge \ldots \wedge b_m)$.
- Suppose we have a fomula of the following form: $\mathrm{G}(a \Rightarrow \mathrm{X}(b))$.
- Construct a temporal tester.
- $a_{prev} = DFF(a)$.
- Construct a new AIG node called, say, *prop*.

Encoding assertions and assumptions

- In ABC, we can only check properties of the form: $\mathrm{G}(a_1 \wedge a_2 \wedge \ldots \wedge a_n) \Rightarrow \mathrm{G}(b_1 \wedge b_2 \wedge \ldots \wedge b_m)$.
- Suppose we have a fomula of the following form: $\mathrm{G}(a \Rightarrow \mathrm{X}(b))$.
- Construct a temporal tester.
- $a_{prev} = DFF(a)$.
- Construct a new AIG node called, say, *prop*.
- Ask ABC to check $\mathrm{G}(prop)$.

Encoding assertions and assumptions

- Suppose we have a property like: $\mathrm{G}(a) \Rightarrow \mathrm{G}(b)$.

Encoding assertions and assumptions

- Suppose we have a property like: $\mathrm{G}(a) \Rightarrow \mathrm{G}(b)$.
- The negation of the property is $\mathrm{G}(a) \wedge \mathrm{F}(\neg b)$.

Encoding assertions and assumptions

- Suppose we have a property like: $\mathrm{G}(a) \Rightarrow \mathrm{G}(b)$.
- The negation of the property is $\mathrm{G}(a) \wedge \mathrm{F}(\neg b)$.
- Handle the assumption using *constr*.

Encoding assertions and assumptions

- Suppose we have a property like: $\mathrm{G}(a) \Rightarrow \mathrm{G}(b)$.
- The negation of the property is $\mathrm{G}(a) \wedge \mathrm{F}(\neg b)$.
- Handle the assumption using *constr*.
- Use a temporal tester for the invariant $\neg b$.

Table of Contents

1. Introduction
2. Methodology
3. Experiments and Results
4. Conclusion and Future Work

- ```
  module ex(clk, a, c, b, d);
    input clk, a, c;
    output reg [7:0] b, d;

    initial b = 8'd0;
    initial d = 8'd0;

    always @(posedge clk)
    begin
      if (a)
         b = b + 8'd1;
      if (c && a)
        d = d - 8'd1;
    end
  endmodule
  ```
  ```

- Use Yosys as the frontend parser.

- Use Yosys as the frontend parser.
- Convert the module into the Aiger format.

- Use Yosys as the frontend parser.
- Convert the module into the Aiger format.
- Caution: Aiger uses the notion of an implicit clock!

- Use Yosys as the frontend parser.
- Convert the module into the Aiger format.
- Caution: Aiger uses the notion of an implicit clock!
- Use ABC for model checking the hyperproperties on this module.

- The usage details for the self_compose command:

```
UC Berkeley, ABC 1.01 (compiled Sep  5 2020 23:06:26)
abc 01> self_compose -h
usage: self_compose [-h] <file_name> <variable_list_file_name>
         builds a self-composed AIG from the file
-h     : print the command usage
```

- $\forall_{\pi_1}\forall_{\pi_2}\mathsf{G}(a_{\pi_1} == a_{\pi_2}) \Rightarrow \mathsf{G}(b_{\pi_1} == b_{\pi_2})$.

- $\forall_{\pi_1}\forall_{\pi_2}\mathsf{G}(a_{\pi_1} == a_{\pi_2}) \Rightarrow \mathsf{G}(b_{\pi_1} == b_{\pi_2})$.
- Create a file named prop1 which contains the following data:

```
1 8
i0
o0 o1 o2 o3 o4 o5 o6 o7
```

- $\forall_{\pi_1}\forall_{\pi_2} G(a_{\pi_1} == a_{\pi_2}) \Rightarrow G(b_{\pi_1} == b_{\pi_2})$.
- Create a file named prop1 which contains the following data:

```
1 8
i0
o0 o1 o2 o3 o4 o5 o6 o7
```

- Now, we build the self-composed model in ABC:

```
abc 01> self_compose ex.aig prop1
abc 02> strash
```

- ```
  abc 03> constr -N 1
  Setting the last 1 POs as constraint outputs.
  abc 03> fold
  ```
  ```

- ```
  abc 03> constr -N 1
  Setting the last 1 POs as constraint outputs.
  abc 03> fold
  ```
- Caution: Do not forget to fold the assumption.
  ```

- ```
  abc 03> constr -N 1
  Setting the last 1 POs as constraint outputs.
  abc 03> fold
  ```
- Caution: Do not forget to fold the assumption.
- Use ABC to check if the invariant holds true:

```
abc 04> pdr
Invariant F[8] : 16 clauses with 17 flops (out of 17)
Verification of invariant with 16 clauses was successful.
Property proved.  Time =     0.06 sec
```
```

- $\forall_{\pi_1} \forall_{\pi_2} G(a_{\pi_1} == a_{\pi_2}) \Rightarrow G(c_{\pi_1} == c_{\pi_2})$

- $\forall_{\pi_1}\forall_{\pi_2} G(a_{\pi_1} == a_{\pi_2}) \Rightarrow G(c_{\pi_1} == c_{\pi_2})$
- The file to be fed to the self_compose command is as follows (saved as prop2):

```
1 1
i0
i1
```

- $\forall_{\pi_1} \forall_{\pi_2} G(a_{\pi_1} == a_{\pi_2}) \Rightarrow G(c_{\pi_1} == c_{\pi_2})$
- The file to be fed to the self_compose command is as follows (saved as prop2):

```
1 1
i0
i1
```

- Check if the invariant holds true:

```
UC Berkeley, ABC 1.01 (compiled Sep  5 2020 23:06:26)
abc 01> self_compose ex.aig prop2
abc 02> strash
abc 03> constr -N 1
Setting the last 1 POs as constraint outputs.
abc 03> fold
abc 04> pdr
Output 0 of miter "ex" was asserted in frame 0.  Time =     0.04 sec
```

- $\forall_{\pi_1}\forall_{\pi_2}\mathrm{G}(a_{\pi_1} == a_{\pi_2}\ \&\&\ c_{\pi_1} == c_{\pi_2}) \Rightarrow \mathrm{G}(d_{\pi_1} == d_{\pi_2})$

- $\forall_{\pi_1}\forall_{\pi_2}\mathrm{G}(a_{\pi_1} == a_{\pi_2} \;\&\&\; c_{\pi_1} == c_{\pi_2}) \Rightarrow \mathrm{G}(d_{\pi_1} == d_{\pi_2})$
- The file to be fed to the self_compose command is as follows (saved as prop3):

```
2 8
i0 i1
o8 o9 o10 o11 o12 o13 o14 o15
```

- $\forall_{\pi_1}\forall_{\pi_2} G(a_{\pi_1} == a_{\pi_2} \ \&\& \ c_{\pi_1} == c_{\pi_2}) \Rightarrow G(d_{\pi_1} == d_{\pi_2})$
- The file to be fed to the self_compose command is as follows (saved as prop3):

```
2 8
i0 i1
o8 o9 o10 o11 o12 o13 o14 o15
```

- ```
  UC Berkeley, ABC 1.01 (compiled Sep  5 2020 23:06:26)
  abc 01> self_compose ex.aig prop3
  abc 02> strash
  abc 03> constr -N 2
  Setting the last 2 POs as constraint outputs.
  abc 03> fold
  abc 04> pdr
  Invariant F[1] : 16 clauses with 17 flops (out of 17)
  Verification of invariant with 16 clauses was successful.
  Property proved.  Time =     0.08 sec
  ```
  ```

Abstraction-Refinement Mechanisms

- A few terminologies to keep in mind:

Abstraction-Refinement Mechanisms

- A few terminologies to keep in mind:
  - Abstraction.

Abstraction-Refinement Mechanisms

- A few terminologies to keep in mind:
  - Abstraction.
  - Concretization.

Abstraction-Refinement Mechanisms

- A few terminologies to keep in mind:
  - Abstraction.
  - Concretization.
  - Refinement.

Abstraction-Refinement Mechanisms

- A few terminologies to keep in mind:
    - Abstraction.
    - Concretization.
    - Refinement.
    - Refutation.

Abstraction-Refinement Mechanisms

- A few terminologies to keep in mind:
    - Abstraction.
    - Concretization.
    - Refinement.
    - Refutation.
- Counterexample Guided Abstraction-Refinement: Use the refutation proof to refine the abstract model.

Abstraction-Refinement Mechanisms

- A few terminologies to keep in mind:
  - Abstraction.
  - Concretization.
  - Refinement.
  - Refutation.
- Counterexample Guided Abstraction-Refinement: Use the refutation proof to refine the abstract model.
- Proof-based Abstraction-Refinement: Eliminate all the counterexamples of length k (the length of the abstract counterexample).

Lazy Self-Composition

- A secure system: The low-security outputs are not affected by the high-security inputs.

Lazy Self-Composition

- A secure system: The low-security outputs are not affected by the high-security inputs.
- Lazy Self-Composition: Combines taint analysis and self-composition to form an abstraction-refinement framework.

Lazy Self-Composition

- A secure system: The low-security outputs are not affected by the high-security inputs.
- Lazy Self-Composition: Combines taint analysis and self-composition to form an abstraction-refinement framework.
- Taint Analysis: To each variable, assign a tainted variable. Check if the taint can be propagated from the high-security variables to the low-security variables.

Lazy Self-Composition

- A secure system: The low-security outputs are not affected by the high-security inputs.
- Lazy Self-Composition: Combines taint analysis and self-composition to form an abstraction-refinement framework.
- Taint Analysis: To each variable, assign a tainted variable. Check if the taint can be propagated from the high-security variables to the low-security variables.
- Important: The tainted model M_t can be thought of as an over-approximation of the self-composed model M_d.

Lazy Self-Composition

- A secure system: The low-security outputs are not affected by the high-security inputs.
- Lazy Self-Composition: Combines taint analysis and self-composition to form an abstraction-refinement framework.
- Taint Analysis: To each variable, assign a tainted variable. Check if the taint can be propagated from the high-security variables to the low-security variables.
- Important: The tainted model M_t can be thought of as an over-approximation of the self-composed model M_d.
- Taint variables track if the associated variable needs to be duplicated in the self-composed model.

Lazy Self-Composition

- Start with M_t.

Lazy Self-Composition

- Start with M_t.
- Check if there is a counterexample for M_t.

Lazy Self-Composition

- Start with M_t.
- Check if there is a counterexample for M_t.
- If necessary, build the corresponding self-composed model M_d (albeit in a lazy manner).

Lazy Self-Composition

- Start with M_t.
- Check if there is a counterexample for M_t.
- If necessary, build the corresponding self-composed model M_d (albeit in a lazy manner).
- Check if there is a counterexample for M_d.

Lazy Self-Composition

- Start with M_t.
- Check if there is a counterexample for M_t.
- If necessary, build the corresponding self-composed model M_d (albeit in a lazy manner).
- Check if there is a counterexample for M_d.
- If necessary, refine M_t using the information obtained from M_d.

Problems with Lazy Self-Composition

- The refinement process is dependent on the counterexample.

Problems with Lazy Self-Composition

- The refinement process is dependent on the counterexample.
- Instead, if proof based abstraction-refinement is used, then the refinement process becomes a bottleneck.

Problems with Lazy Self-Composition

- The refinement process is dependent on the counterexample.
- Instead, if proof based abstraction-refinement is used, then the refinement process becomes a bottleneck.
- Can we do better?

Improvements

- Refute all the spurious counterexamples of length k in M_t using proof based abstraction-refinement

Improvements

- Refute all the spurious counterexamples of length k in M_t using proof based abstraction-refinement
- If the time exceeds a certain threshold, then we terminate the refinement process.

Improvements

- Refute all the spurious counterexamples of length k in M_t using proof based abstraction-refinement
- If the time exceeds a certain threshold, then we terminate the refinement process.
- Refine M_t using information from M_d.

Table of Contents

1. Introduction
2. Methodology
3. Experiments and Results
4. Conclusion and Future Work

Conclusion and Future Work

- In this work, we have augmented ABC (A System for Sequential Synthesis and Verification) with the ability to perform model checking of 2-safety properties.

Conclusion and Future Work

- In this work, we have augmented ABC (A System for Sequential Synthesis and Verification) with the ability to perform model checking of 2-safety properties.
- In addition, we suggested some improvements for lazy self-composition.

Conclusion and Future Work

- In this work, we have augmented ABC (A System for Sequential Synthesis and Verification) with the ability to perform model checking of 2-safety properties.
- In addition, we suggested some improvements for lazy self-composition.
- In future, we need to determine how to efficiently perform taint analysis on an AIG.

Conclusion and Future Work

- In this work, we have augmented ABC (A System for Sequential Synthesis and Verification) with the ability to perform model checking of 2-safety properties.
- In addition, we suggested some improvements for lazy self-composition.
- In future, we need to determine how to efficiently perform taint analysis on an AIG.
- Also, the suggested modification to the lazy self-composition technique needs to implemented inside the ABC system or using ABC as a static library.

Conclusion and Future Work

- In this work, we have augmented ABC (A System for Sequential Synthesis and Verification) with the ability to perform model checking of 2-safety properties.
- In addition, we suggested some improvements for lazy self-composition.
- In future, we need to determine how to efficiently perform taint analysis on an AIG.
- Also, the suggested modification to the lazy self-composition technique needs to implemented inside the ABC system or using ABC as a static library.

Thanks for your attention!