

A brief tutorial on formal verification with applications to security protocols

Nehul Jain

Ansuman Banerjee

Indian Statistical Institute

Outline

- Formal Verification: The basics
 - Explicit Model checking
 - Symbolic Analysis
 - CEGAR
 - Equivalence checking
- Formal verification: In the security context
 - Case studies on AES

June 2002

“Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated \$59.5 billion annually, or about 0.6 percent of the gross domestic product

...

At the national level, over half of the costs are borne by software users and the remainder by software developers/vendors.”

“The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated **\$22.2 billion, could be eliminated** by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects.”

Model Checking



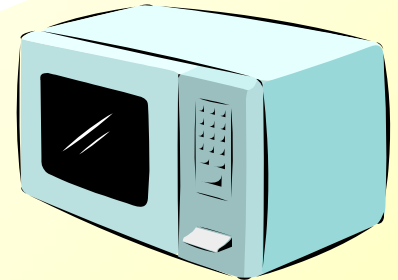
- Developed independently by **Clarke and Emerson** and by **Queille and Sifakis** in early 1980's.
- **Properties** are written in **propositional temporal logic**.
- Systems are modeled by **finite state machines**.
- Verification procedure is an **exhaustive search of the state space** of the design.
- Model checking **complements** testing/simulation.

Advantages of Model Checking



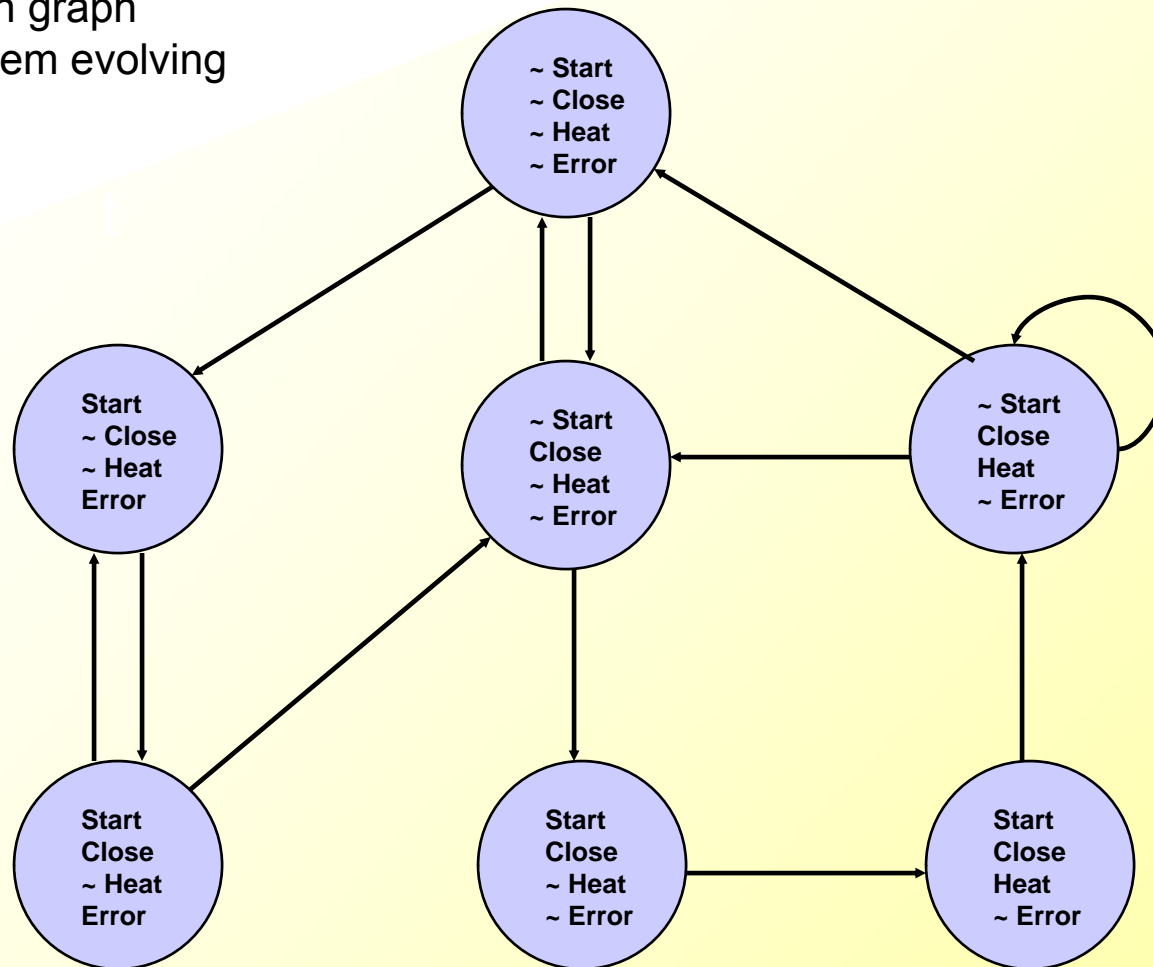
- **No proofs!!!**
- **Fast (compared to other rigorous methods)**
- **Diagnostic counterexamples**
- **No problem with partial specifications / properties**
- **Logics can easily express many concurrency properties**

Model of computation



Microwave Oven Example

State-transition graph describes system evolving over time.



Temporal Logic



- The oven doesn't **heat up** until the **door is closed**.
- **Not** **heat_up** holds **until** **door_closed**
- $(\sim \text{heat_up}) \text{ U } \text{door_closed}$

Basic Temporal Operators



The symbol “**p**” is an atomic proposition, e.g. “**heat_up**” or “**door_closed**”.

- **F**p - p holds sometime in the *future*.
- **G**p - p holds *globally* in the future.
- **X**p - p holds *next* time.
- p**U**q - p holds *until* q holds.

Model Checking Problem



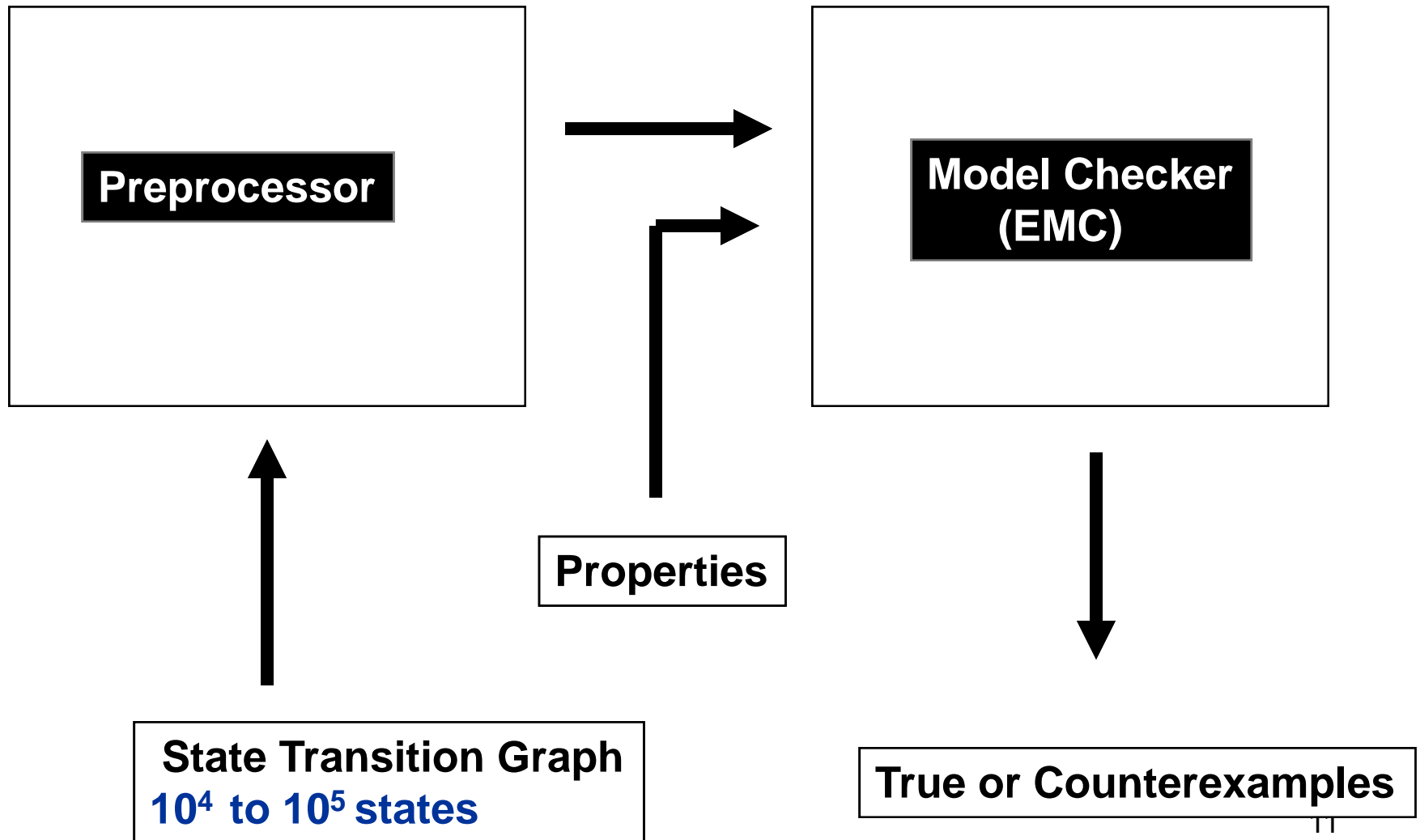
Let M be a model, i.e., a **state-transition graph**.

Let f be the **property** in temporal logic.

Find all states s such that M has property f at state s .

Efficient Algorithms: CE81, CES83

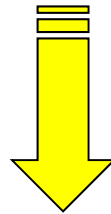
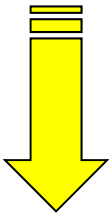
The EMC System 1982/83



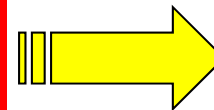
Model Checker Architecture

System Description

Formal Specification



State Explosion Problem!!

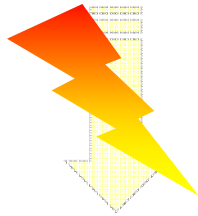


**Validation
or
Counterexample**

Model Checker

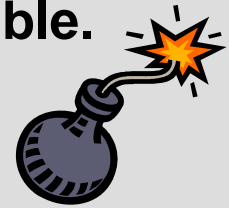
The State Explosion Problem

System Description



State Transition Graph

Combinatorial explosion of system states renders explicit model construction infeasible.



Exponential Growth of ...

- ... global state space in number of concurrent components.
- ... memory states in memory size.

Feasibility of model checking inherently tied to handling state explosion.

Combating State Explosion



- **Binary Decision Diagrams** can be used to represent state transition systems more efficiently.
→ **Symbolic Model Checking 1992**
- **Semantic techniques** for alleviating state explosion:
 - Partial Order Reduction.
 - Abstraction.
 - Compositional reasoning.
 - Symmetry.
 - Cone of influence reduction.
 - Semantic minimization.

Model Checking since 1981



1981	Clarke / Emerson: CTL Model Checking Sifakis / Quielle	10^5
1982	EMC: Explicit Model Checker Clarke, Emerson, Sistla	
1990	Symbolic Model Checking Burch, Clarke, Dill, McMillan	10^{100}
1992	SMV: Symbolic Model Verifier McMillan	
	1990s: Formal Hardware Verification in Industry: Intel, IBM, Motorola, etc.	
1998	Bounded Model Checking using SAT Biere, Clarke, Zhu	10^{1000}
2000	Counterexample-guided Abstraction Refinement Clarke, Grumberg, Jha, Lu, Veith	

Model Checking since 1981



1981 Clarke / Emerson: CTL Model Checking
Sifakis / Quielle

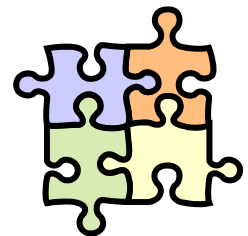
1982 EMC: Explicit Model Checker
Clarke, Emerson, Sistla

1990 Symbolic Model Checking
Burch, Clarke, Dill, McMillan

1992 SMV: Symbolic Model Verifier
McMillan

1998 **Bounded Model Checking** using SAT
Biere, Clarke, Zhu

2000 **Counterexample-guided Abstraction Refinement**
Clarke, Grumberg, Jha, Lu, Veith



 **CBMC**

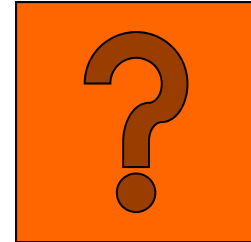
 **MAGIC**

Grand Challenge: **Model Check Software !**

What makes Software Model Checking
different ?

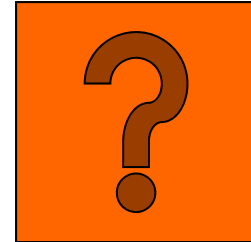


What Makes Software Model Checking Different ?



- Large/unbounded base types: `int`, `float`, `string`
- User-defined types/classes
- Pointers/aliasing + unbounded #'s of heap-allocated cells
- Procedure calls/recursion/calls through pointers/dynamic method lookup/overloading
- Concurrency + unbounded #'s of threads

What Makes Software Model Checking Different ?



- Templates/generics/include files
- Interrupts/exceptions/callbacks
- Use of secondary storage: files, databases
- Absent source code for: libraries, system calls, mobile code
- Esoteric features: continuations, self-modifying code
- Size (e.g., MS Word = 1.4 MLOC)

Grand Challenge: Model Check Software !

Early attempts in the 1980s failed to scale.

2000s: renewed interest / demand:

Java Pathfinder: NASA Ames

SLAM: Microsoft

Bandera: Kansas State

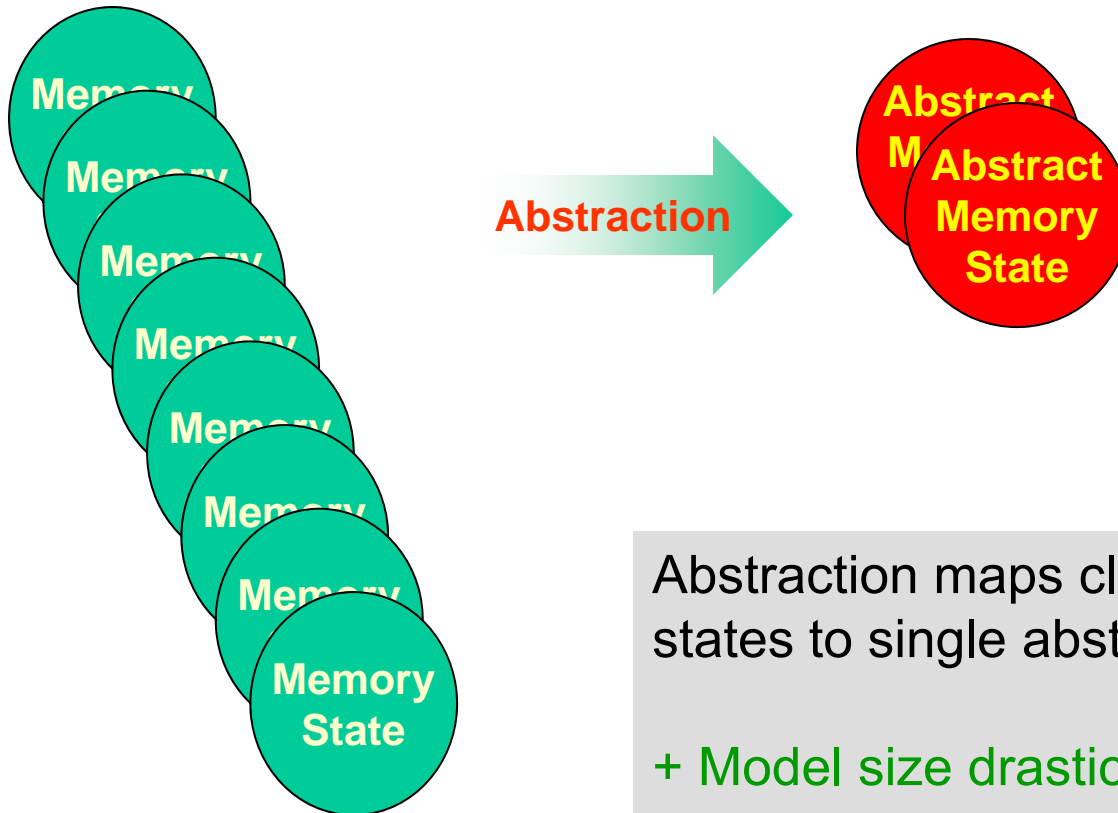
BLAST: Berkeley

...

SLAM shipped to Windows device driver developers.

In general, these tools are unable to handle **complex data structures** and **concurrency**.

Counterexample-Guided Abstraction Refinement



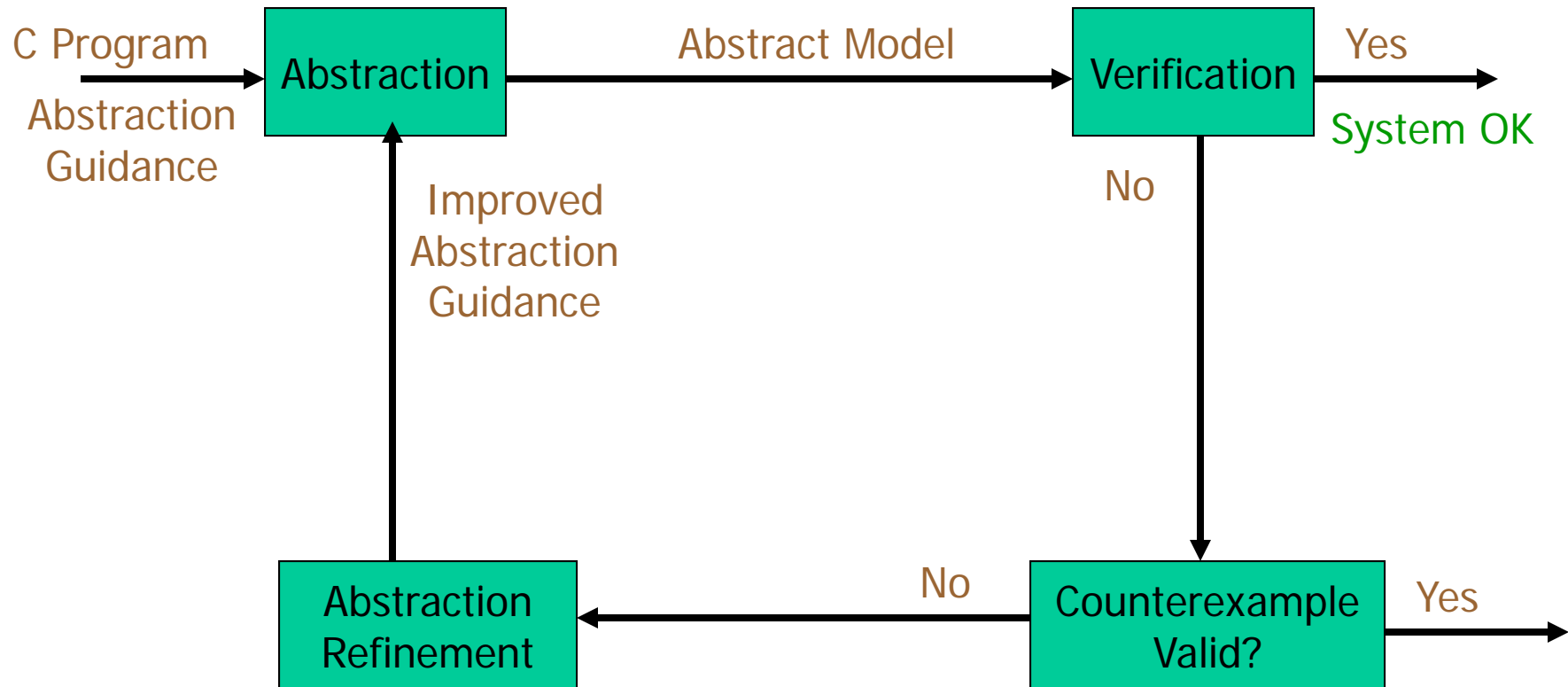
Abstraction maps classes of similar memory states to single abstract memory states.

+ Model size drastically reduced.

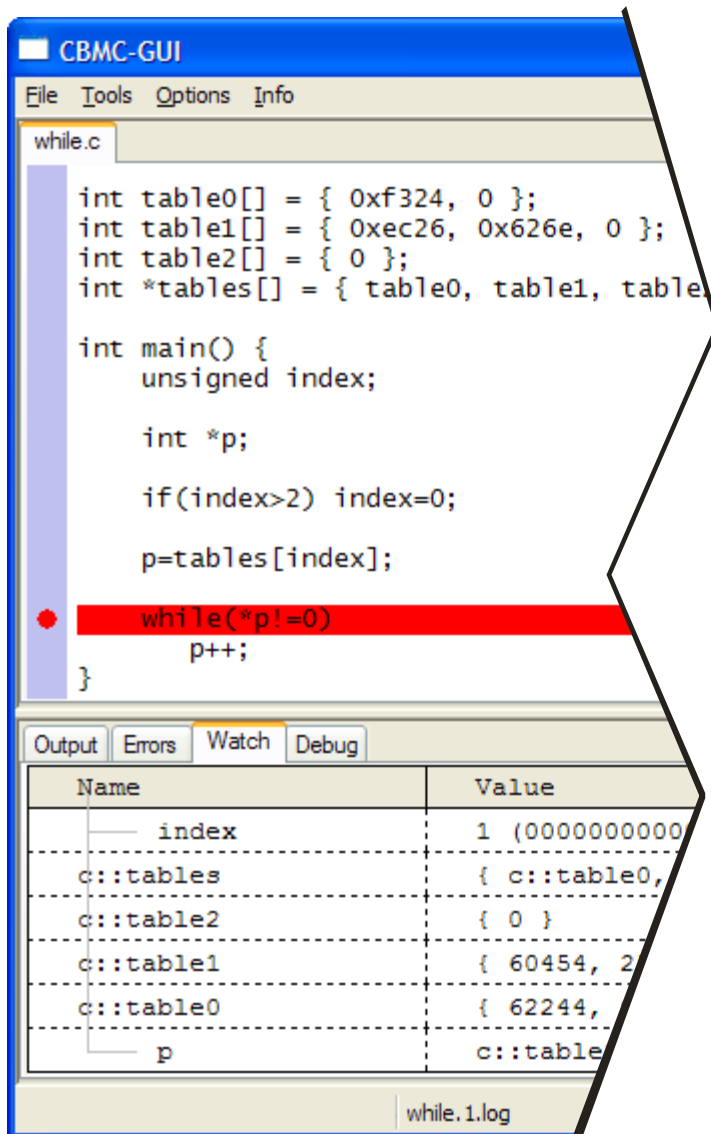
- Invalid counterexamples possible.

The MAGIC Tool:

Counterexample-Guided Abstraction Refinement



CBMC: Embedded Systems Verification



- Method:
Bounded Model Checking
- Implemented **GUI** to facilitate tech transfer
- Applications:
 - Part of train controller from GE
 - Cryptographic algorithms (DES, AES, SHS)
 - C Models of ASICs provided by nVidia

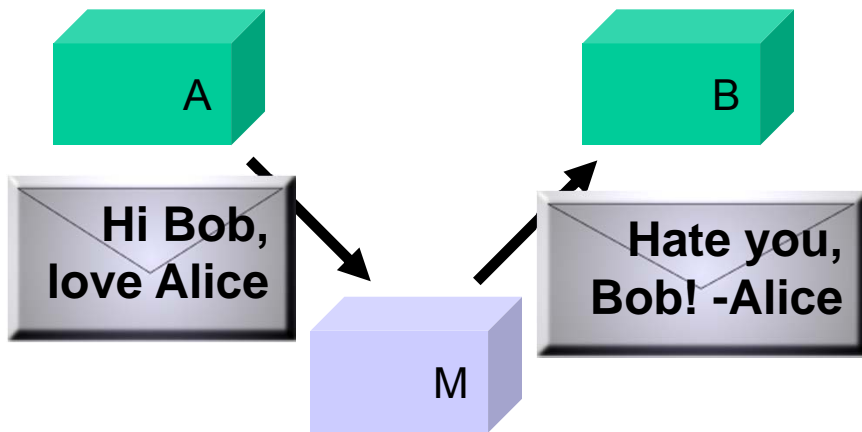
Session 2

Formal Analysis: In the security context

Formal Methods

- Dolev&Yao first formalize N&S problem in early 80s
 - Public key decryption: $\{ \{ M \}_{K_A} \}_{K_A^{-1}} = M$
 - Their work now widely recognised, but at the time, few proof techniques, and little applied
- In 1987, Burrows, Abadi and Needham (BAN) propose a systematic rule-based logic for reasoning about protocols
 - If P believes that he shares a key K with Q, and sees the message M encrypted under K, then he will believe that Q once said M
 - If P believes that the message M is fresh, and also believes that Q once said M, then he will believe that Q believes M
 - Incomplete, but useful; hugely influential

A Potted History



We assume that an intruder can interpose a computer on all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. While this may seem an extreme view, it is the only safe one when designing authentication protocols.

Needham and Schroeder CACM (1978)

- 1978: N&S propose authentication protocols for “large networks of computers”
- 1981: Denning and Sacco find attack found on N&S symmetric key protocol
- 1983: Dolev and Yao first formalize secrecy properties wrt N&S threat model, using formal algebra
- 1987: Burrows, Abadi, Needham invent authentication logic; incomplete, but useful
- 1994: Hickman invents first version of SSL; holes in v1, v2, but v3 fixes these, very widely deployed
- 1994: Ylonen invents SSH; holes in v1, but v2 good, very widely deployed
- 1995: Abadi, Anderson, Needham, et al propose various informal “robustness principles”
- 1995: Lowe finds insider attack on N&S asymmetric protocol; rejuvenates interest in FMs
- circa 2000: Several FMs for “D&Y problem”: tradeoff between accuracy and approximation
- circa 2005: Many FMs now developed; several deliver both accuracy and automation
- 2005: Cervesato et al find same insider attack as Lowe on proposed public-key Kerberos

Job Done?

- After intense effort on symbolic reasoning, there are now several techniques for automatically proving properties of protocols represented within a symbolic, algebraic model
 - eg Athena, TAPS, ProVerif, FDR, AVISPA, etc
- Moreover, many of the unwarranted Dolev Yao abstractions (eg that message length is unobservable) are being addressed by relating symbolic techniques to the probabilistic computational models used by cryptographers
 - See the proceedings of the *Formal and Computational Cryptography* workshops, for example

The trouble is

- While practitioners are typically happy for researchers to write formal models of their natural language specifications, and to apply design principles and formal tools, they are reluctant to do so themselves
- Specs are always refined by implementation experience, so absolute correctness (at least of V1) is not a goal
 - Timely agreement is more important
- So specs tend to be partial and ambiguous.
- Implementation code is the closest we get to a formal description of most protocols
- Hence, we need to learn from other areas of verification, and build tools to analyse code

From Model to Code

- Many formalisms for crypto protocols (including those based on process algebra and process calculi) amount to small programming languages
- Several tools have successfully demonstrated the idea:
 - Strand spaces: Perrig, Song, Phan (2001), Lukell et al (2003)
 - CAPSL: Muller and Millen (2001)
 - Spi calculus: Lashari (2002), Pozza, Sista, Durante (2004)
 - Apparently, the resulting code does not interoperate with other implementations
- But this amounts to growing a formal model into a full programming language, building a compiler, educating developers and so on.

From Code to Model

- Many code analysis tools can detect security issues, such as buffer overruns, but tools to extract D&Y models from code are comparatively new
- Bhargavan, Fournet, and Gordon (CCS'04) extracted verifiable pi-calculus models from XML policies configuring some WS-Security protocols
 - First extraction of D&Y models from implementation files
- Goubault-Larrecq and Parrennes (VMCAI'05) did first tool to extract D&Y models from the source code (in C) of a crypto protocol
 - Based on a pointer analysis they extract a Horn clause model suitable for analysis by other tools eg SPASS
 - They analyse one of two roles in the NSL protocol

Correctness vs Security

- Program or system **correctness**:
program satisfies specification
 - For reasonable input, get reasonable output
- Program or system **security**:
program properties preserved in face of attack
 - For unreasonable input, output not completely disastrous
- Main differences
 - Active interference from adversary
 - Refinement techniques may fail
 - Abstraction is very difficult to achieve in security:
what if the adversary operates below your level of abstraction?

Security Analysis

- ❶ Model system
- ❷ Model adversary
- ❸ Identify security properties
- ❹ See if properties preserved under attack

Theme #1: there are many notions of what it means for a protocol to be “secure”

- Result

- Under given assumptions about system, no attack of a certain form will destroy specified properties
- There is no “absolute” security

Theme #2: there are many ways of looking for security flaws

Theme #1: Protocols and Properties

- Authentication
 - Needham-Schroeder, Kerberos
- Key establishment
 - SSL/TLS, IPSec protocols (IKE, JFK, IKEv2)
- Secure group protocols
 - Group Diffie-Hellman, CLIQUES, key trees and graphs
- Anonymity
 - MIX, Onion routing, Mixmaster and Mixminion
- Electronic payments, wireless security, fair exchange, privacy...

Some of these are excellent topics for a project or the paper-reading assignment

Theme #2: Formal Analysis Methods

- Focus on special-purpose security applications
 - Some techniques are very different from those used in hardware verification
 - In all cases, the main difficulty is modeling the attacker
- Simple, mechanical models of the attacker

Variety of Tools and Techniques

Secrecy
Authentication
Authorization

- Explicit finite-state checking
 - Mur ϕ model checker
- Infinite-state symbolic model checking
 - SRI constraint solver
- Process algebras
 - Applied pi-calculus

Anonymity

◆ Probabilistic model checking
PRISM probabilistic model checker

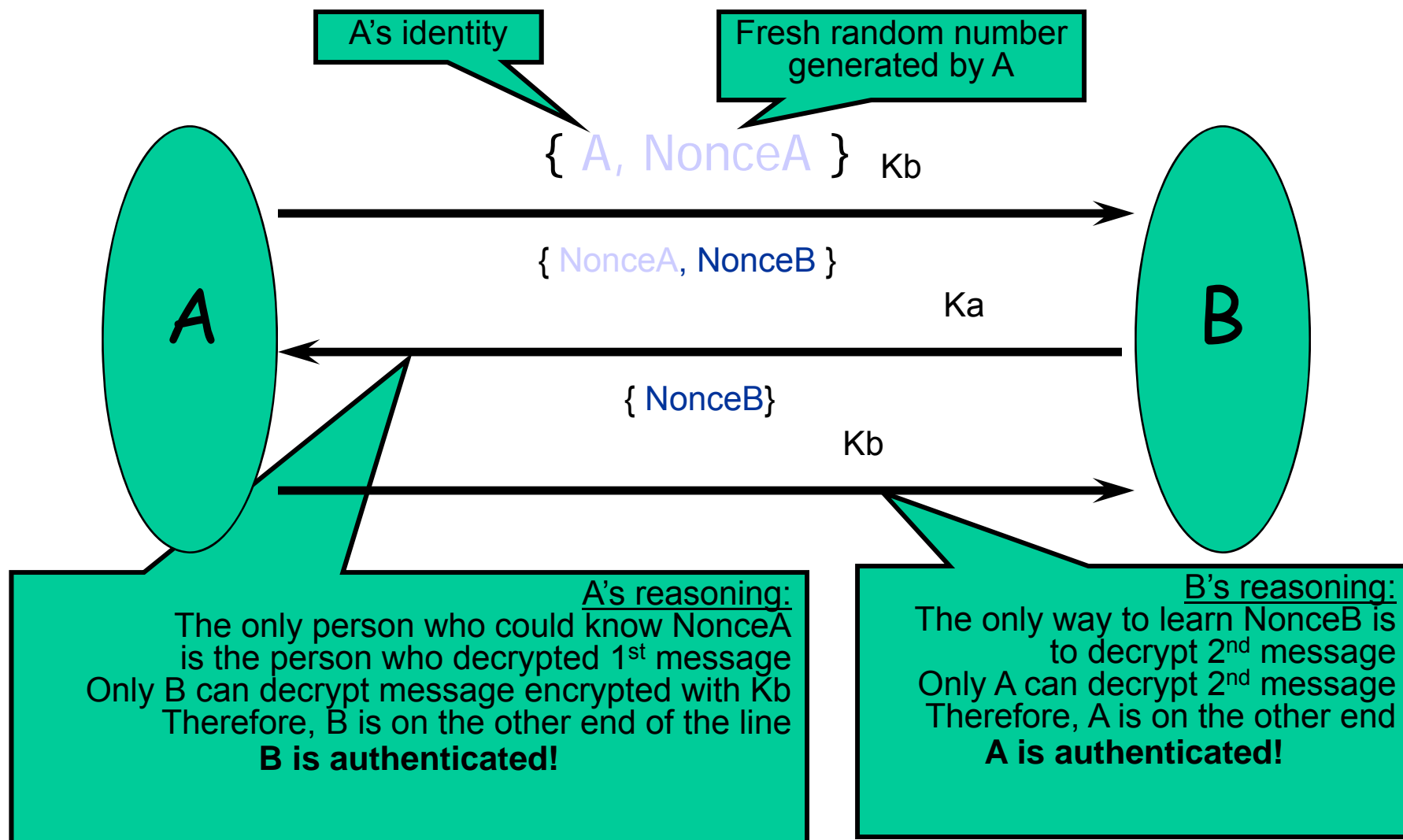
Fairness

◆ Game-based verification
MOCHA model checker

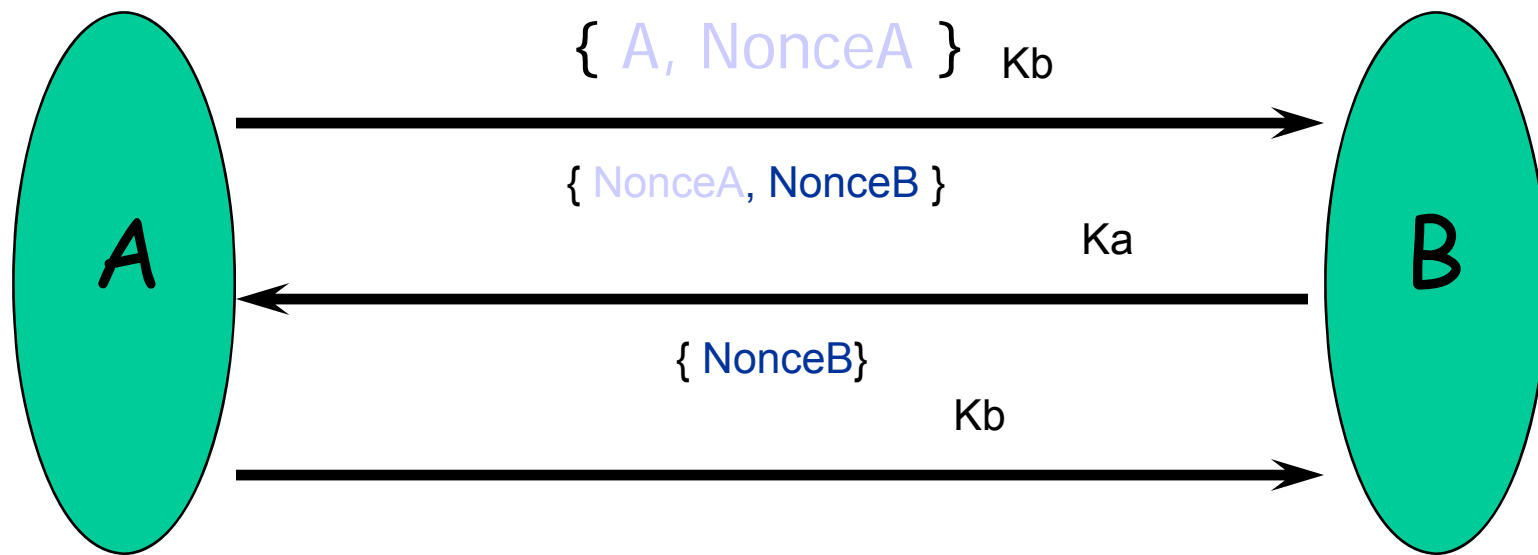
Example: Needham-Schroeder

- Very (in)famous example
 - Appeared in a 1979 paper
 - Goal: authentication in a network of workstations
 - In 1995, Gavin Lowe discovered unintended property while preparing formal analysis using FDR system
- Background: public-key cryptography
 - Every agent A has a key pair K_a, K_a^{-1}
 - Everybody knows public key K_a and can encrypt messages to A with it (we'll use $\{m\}_{K_a}$ notation)
 - Only A knows secret key K_a^{-1} , therefore, only A can decrypt messages encrypted with K_a

Needham-Schroeder Public-Key Protocol



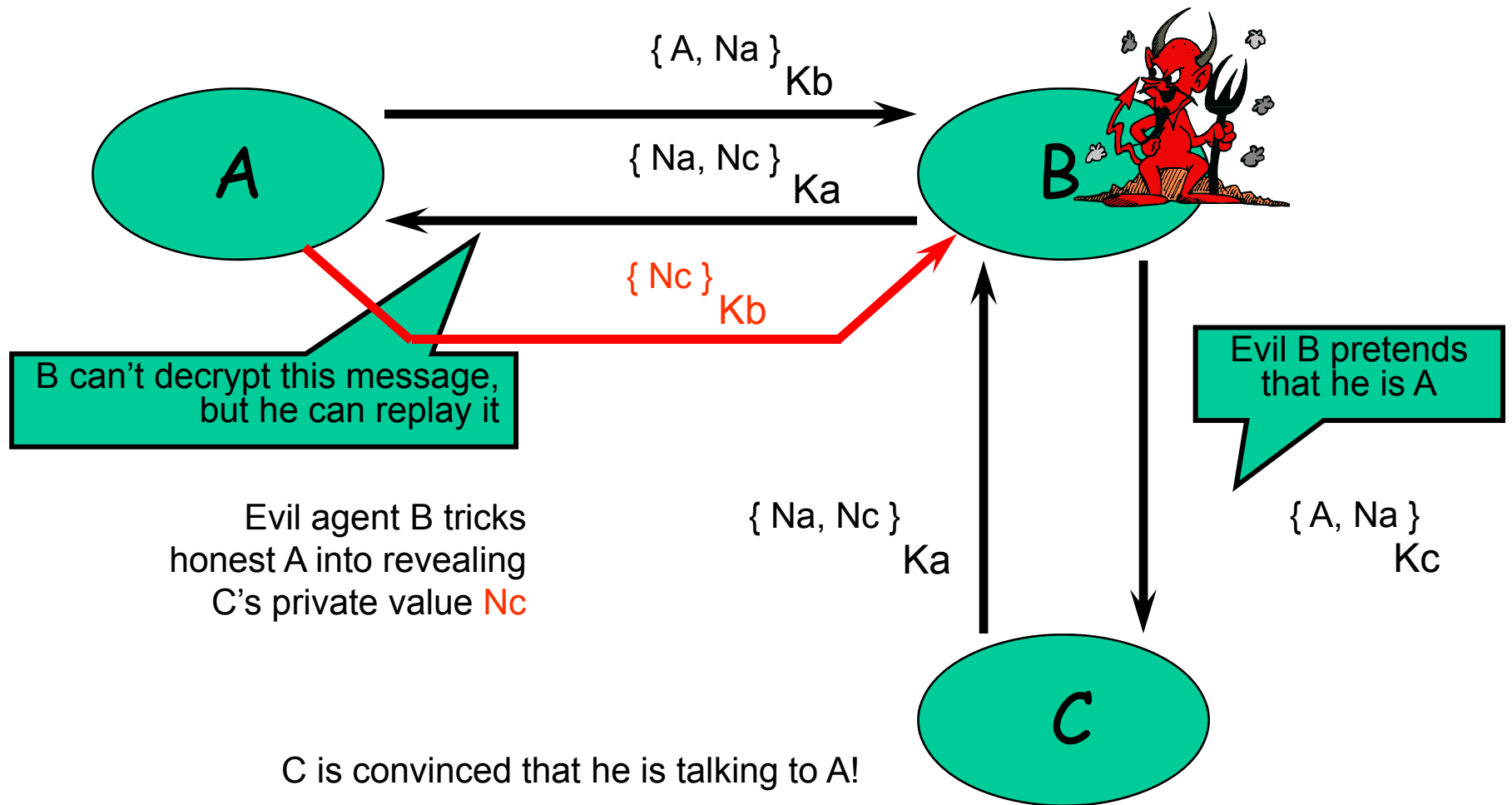
What Does This Protocol Achieve?



- Protocol aims to provide both **authentication** and **secrecy**
- After this the exchange, only A and B know N_a and N_b
- N_a and N_b can be used to derive a shared key

Anomaly in Needham-Schroeder

[published by Lowe]



Lessons of Needham-Schroeder

- Classic man-in-the-middle attack
- Exploits participants' reasoning to fool them
 - A is correct that B must have decrypted $\{A, Na\}_{K_b}$ message, but this does not mean that $\{Na, Nb\}_{K_a}$ message came from B
 - The attack has nothing to do with cryptography!
- It is important to realize limitations of protocols
 - The attack requires that A willingly talk to adversary
 - In the original setting, each workstation is assumed to be well-behaved, and the protocol is correct!
- Wouldn't it be great if one could discover attacks like this automatically?

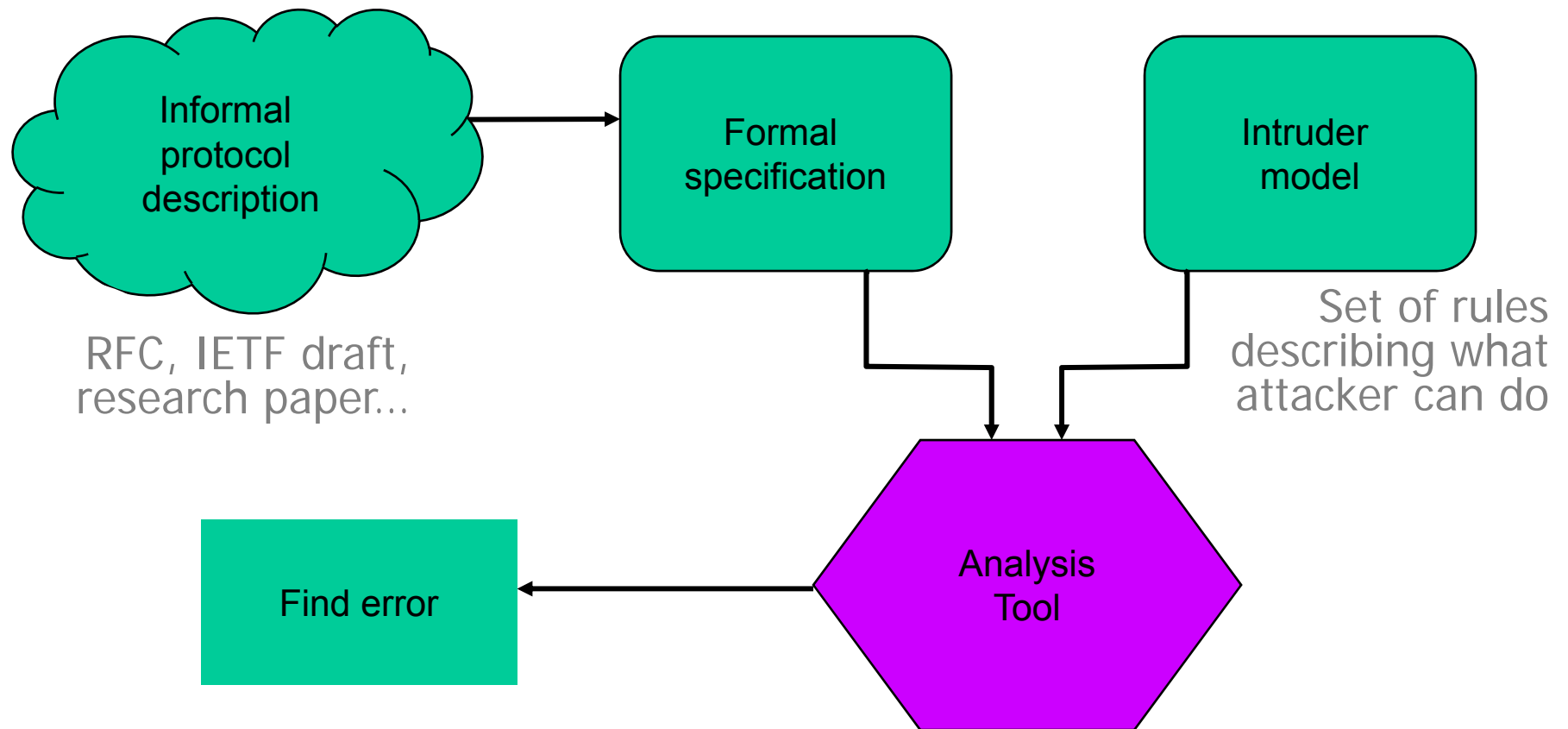
Important Modeling Decisions

- How powerful is the adversary?
 - Simple replay of previous messages
 - Decompose into pieces, reassemble and resend
 - Statistical analysis, partial info from network traffic
 - Timing attacks
- How much detail in underlying data types?
 - Plaintext, ciphertext and keys
 - Atomic data or bit sequences?
 - Encryption and hash functions
 - Perfect (“black-box”) cryptography
 - Algebraic properties: $\text{encr}(x+y) = \text{encr}(x) * \text{encr}(y)$ for RSA
because $\text{encrypt}(k, \text{msg}) = \text{msg}^k \bmod N$

Fundamental Tradeoff

- Formal models are abstract and greatly simplified
 - Components modeled as finite-state machines
 - Cryptographic functions modeled as abstract data types
 - Security property stated as unreachability of “bad” state
- Formal models are tractable...
 - Lots of verification methods, many automated
- ...but not necessarily sound
 - Proofs in the abstract model are subject to simplifying assumptions which ignore some of attacker’s capabilities
- Attack in the formal model implies actual attack

Explicit Intruder Method



Murφ

[Dill et al.]

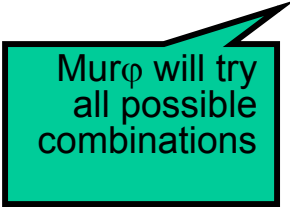
- Describe finite-state system
 - State variables with initial values
 - Transition rules for each protocol participant
 - Communication by shared variables
- Specify security condition as a state invariant
 - Predicate over state variables that must be true in every state reachable by the protocol
- Automatic exhaustive state enumeration
 - Can use hash table to avoid repeating states
- Research and industrial protocol verification

Making the Model Finite

- Two sources of infinite behavior
 - Many instances of participants, multiple runs
 - Message space or data space may be infinite
- Finite approximation
 - Assume finite number of participants
 - For example, 2 clients, 2 servers
 - Mur ϕ is scalable: can choose system size parameters
 - Assume finite message space
 - Represent random numbers by constants r_1, r_2, r_3, \dots
 - Do not allow `encrypt(encrypt(encrypt(...)))`

Applying Mur ϕ to Security Protocols

- Formulate the protocol
 - Define a datatype for each message format
 - Describe finite-state behavior of each participant
 - If received message M3, then create message M4, deposit it in the network buffer, and go to state WAIT
 - Describe security condition as state invariant
- Add adversary
 - Full control over the “network” (shared buffer)
 - Nondeterministic choice of actions
 - Intercept a message and split it into parts; remember parts
 - Generate new messages from observed data and initial knowledge (e.g., public keys)



Mur ϕ will try all possible combinations

Needham-Schroeder in Mur ϕ (1)

const

```
NumInitiators:  1;    -- number of initiators
NumResponders:  1;    -- number of responders
NumIntruders:   1;    -- number of intruders
NetworkSize:    1;    -- max. outstanding msgs in network
MaxKnowledge:   10;   -- number msgs intruder can remember
```

type

```
InitiatorId:  scalarset (NumInitiators);
ResponderId:  scalarset (NumResponders);
IntruderId:   scalarset (NumIntruders);

AgentId:      union {InitiatorId, ResponderId, IntruderId};
```

Needham-Schroeder in Mur ϕ (2)

```
MessageType : enum {                -- types of messages
    M_NonceAddress,                -- {Na, A}Kb  nonce and addr
    M_NonceNonce,                  -- {Na,Nb}Ka  two nonces
    M_Nonce                        -- {Nb}Kb      one nonce
};

Message : record
    source:   AgentId;              -- source of message
    dest:     AgentId;              -- intended destination of msg
    key:      AgentId;              -- key used for encryption
    mType:    MessageType;          -- type of message
    nonce1:   AgentId;              -- nonce1
    nonce2:   AgentId;              -- nonce2 OR sender id OR empty
end;
```


Needham-Schroeder in Mur ϕ (3)

```
-- intruder i sends recorded message
ruleset i: IntruderId do                -- arbitrary choice of
  choose j: int[i].messages do          -- recorded message
    ruleset k: AgentId do               -- destination
      rule "intruder sends recorded message"
        !ismember(k, IntruderId) &    -- not to intruders
        multisetcount (l:net, true) < NetworkSize
      ==>
      var outM: Message;
      begin
        outM          := int[i].messages[j];
        outM.source := i;
        outM.dest   := k;
        multisetadd (outM,net);
      end;
    end;
  end;
end;
```

Game-Based Verification of Security Protocols

Alternating Transition Systems

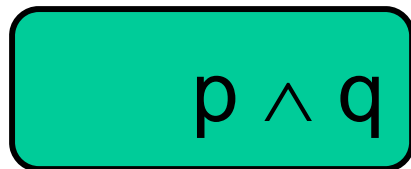
- Game variant of Kripke structures
 - *R. Alur, T. Henzinger, O. Kupferman. “Alternating-time temporal logic”. FOCS 1997.*
- Start by defining state space of the protocol
 - Π is a set of propositions
 - Σ is a set of players
 - Q is a set of states
 - $Q_0 \subseteq Q$ is a set of initial states
 - $\pi: Q \rightarrow 2^\Pi$ maps each state to the set of propositions that are true in the state
- So far, this is very similar to $\text{Mur}\phi$

Transition Function

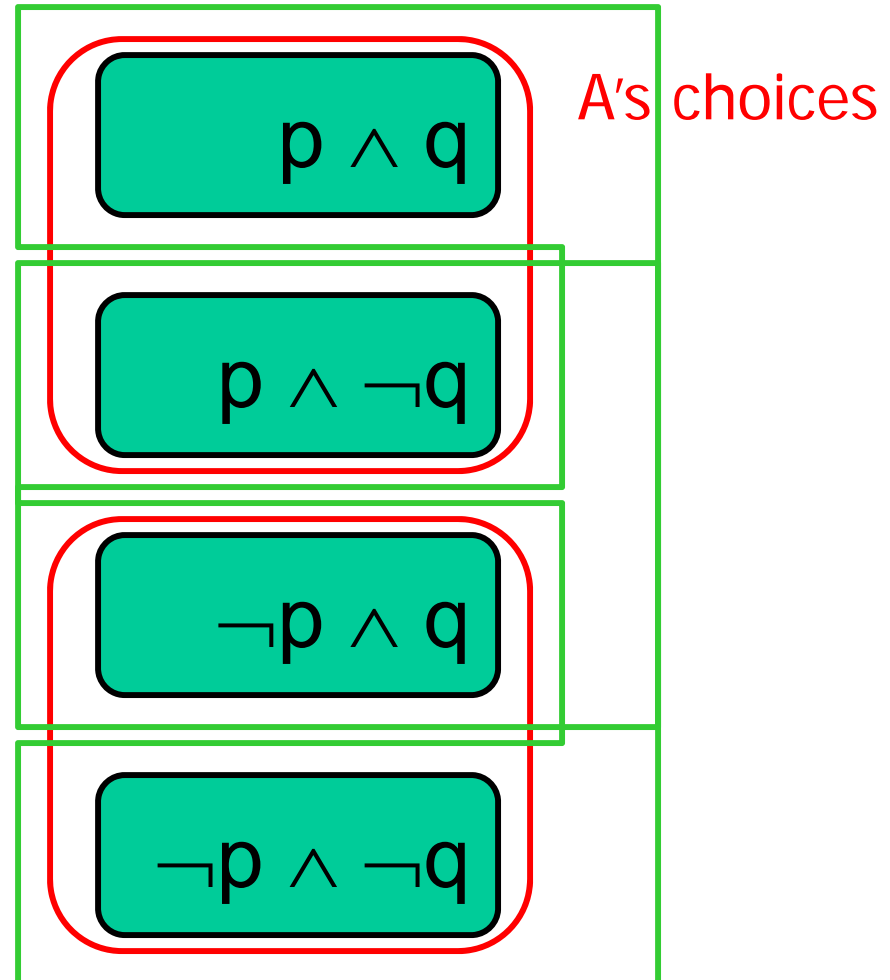
- $\delta: Q \times \Sigma \rightarrow 2^{2^Q}$ maps a state and a player to a nonempty set of choices, where each choice is a set of possible next states
 - When the system is in state q , each player chooses a set $Q_a \in \delta(q, a)$
 - The next state is the intersection of choices made by all players $\bigcap_{a \in \Sigma} \delta(q, a)$
 - The transition function must be defined in such a way that the intersection contains a unique state
- Informally, a player chooses a set of possible next states, then his opponents choose one of them

Example: Two-Player ATS

$\Sigma = \{\text{Alice}, \text{Bob}\}$



B's choices

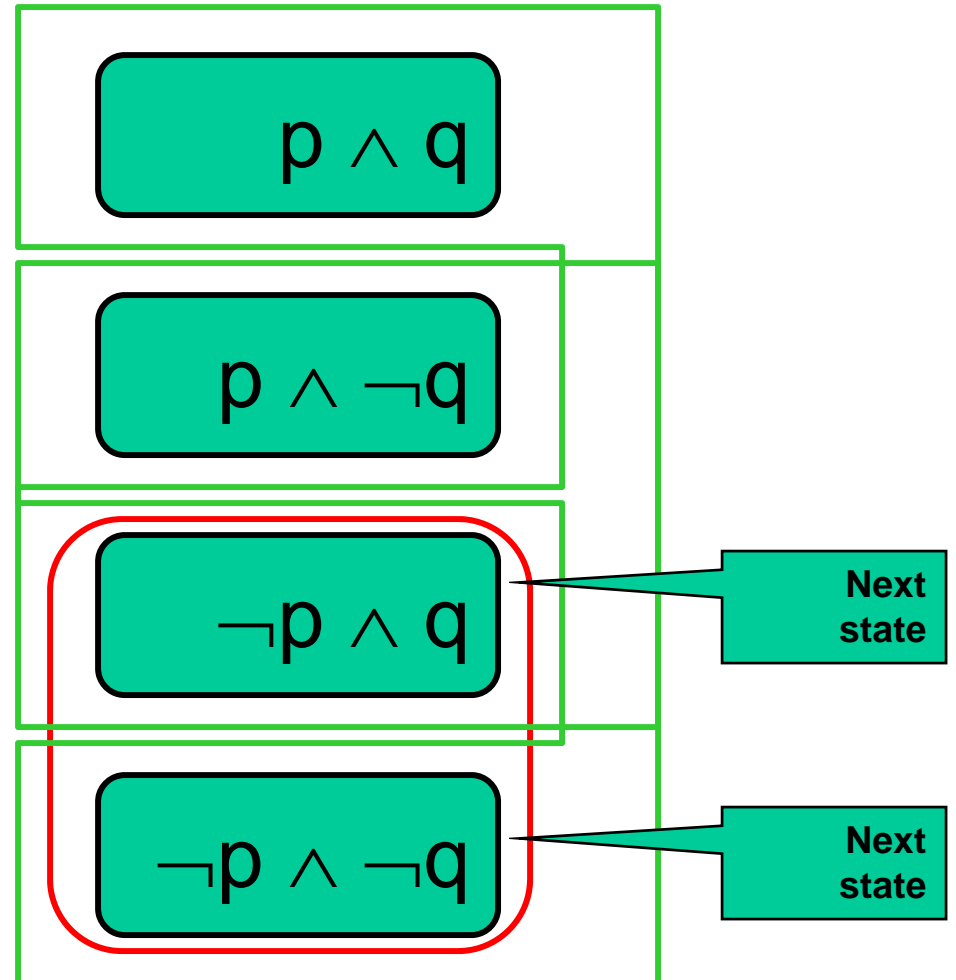


Example: Computing Next State

$\Sigma = \{\text{Alice}, \text{Bob}\}$



If A chooses this set...
... B can choose either state



Alternating-Time Temporal Logic

- Propositions $p \in \Pi$
- $\neg\varphi$ or $\varphi_1 \vee \varphi_2$ where $\varphi, \varphi_1, \varphi_2$ are ATL formulas
- $\langle\langle A \rangle\rangle \bigcirc \varphi$, $\langle\langle A \rangle\rangle \Box \varphi$, $\langle\langle A \rangle\rangle \varphi_1 \mathbf{U} \varphi_2$ where $A \subseteq \Sigma$ is a set of players, $\varphi, \varphi_1, \varphi_2$ are ATL formulas
 - These formulas express the ability of coalition A to achieve a certain outcome
 - \bigcirc , \Box , \mathbf{U} are standard temporal operators (similar to what we saw in PCTL)
- Define $\langle\langle A \rangle\rangle \Diamond \varphi$ as $\langle\langle A \rangle\rangle \text{true} \mathbf{U} \varphi$

Strategies in ATL

- A **strategy** for a player $a \in \Sigma$ is a mapping $f_a: Q^+ \rightarrow 2^Q$ such that for all prefixes $\lambda \in Q^*$ and all states $q \in Q$, $f_a(\lambda \cdot q) \in \delta(q, a)$
 - For each player, strategy maps any sequence of states to a set of possible next states
- Informally, the strategy tells the player in each state what to do next
 - Note that the player cannot choose the next state. He can only choose a set of possible next states, and opponents will choose one of them as the next state.

Temporal ATL Formulas (I)

- $\langle\langle A \rangle\rangle \bigcirc \varphi$ iff there exists a set F_a of strategies, one for each player in A , such that **for all future executions** $\lambda \in \text{out}(q, F_a)$ **φ holds in first state** $\lambda[1]$
 - Here $\text{out}(q, F_a)$ is the set of all future executions assuming the players follow the strategies prescribed by F_a , i.e., $\lambda = q_0 q_1 q_2 \dots \in \text{out}(q, F_a)$ if $q_0 = q$ and $\forall i \ q_{i+1} \in \bigcap_{a \in A} f_a(\lambda[0, i])$
- Informally, $\langle\langle A \rangle\rangle \bigcirc \varphi$ holds if coalition A has a strategy such that φ always holds in the next state

Temporal ATL Formulas (II)

- $\langle\langle A \rangle\rangle \Box \varphi$ iff there exists a set F_a of strategies, one for each player in A , such that **for all future executions** $\lambda \in \text{out}(q, F_a)$ **φ holds in all states**
 - Informally, $\langle\langle A \rangle\rangle \Box \varphi$ holds if coalition A has a strategy such that φ holds in every execution state
- $\langle\langle A \rangle\rangle \Diamond \varphi$ iff there exists a set F_a of strategies, one for each player in A , such that **for all future executions** $\lambda \in \text{out}(q, F_a)$ **φ eventually holds in some state**
 - Informally, $\langle\langle A \rangle\rangle \Diamond \varphi$ holds if coalition A has a strategy such that φ is true at some point in every execution

Protocol Description Language

◆ Guarded command language

◆ Each action described as `[] guard → command`

- `guard` is a boolean predicate over state variables
- `command` is an update predicate

```
[ ]SigM1B ∧ ¬SendM2 ∧ ¬StopB -> SendMrB1' := true;
```

MOCHA Model Checker

- Model checker specifically designed for verifying alternating transition systems
 - System behavior specified as guarded commands
 - Essentially the same as PRISM input, except that transitions are nondeterministic (as in in Mur ϕ), not probabilistic
 - Property specified as ATL formula
- Slang scripting language
 - Makes writing protocol specifications easier
- Try online implementation!

Formal verification: The AES story

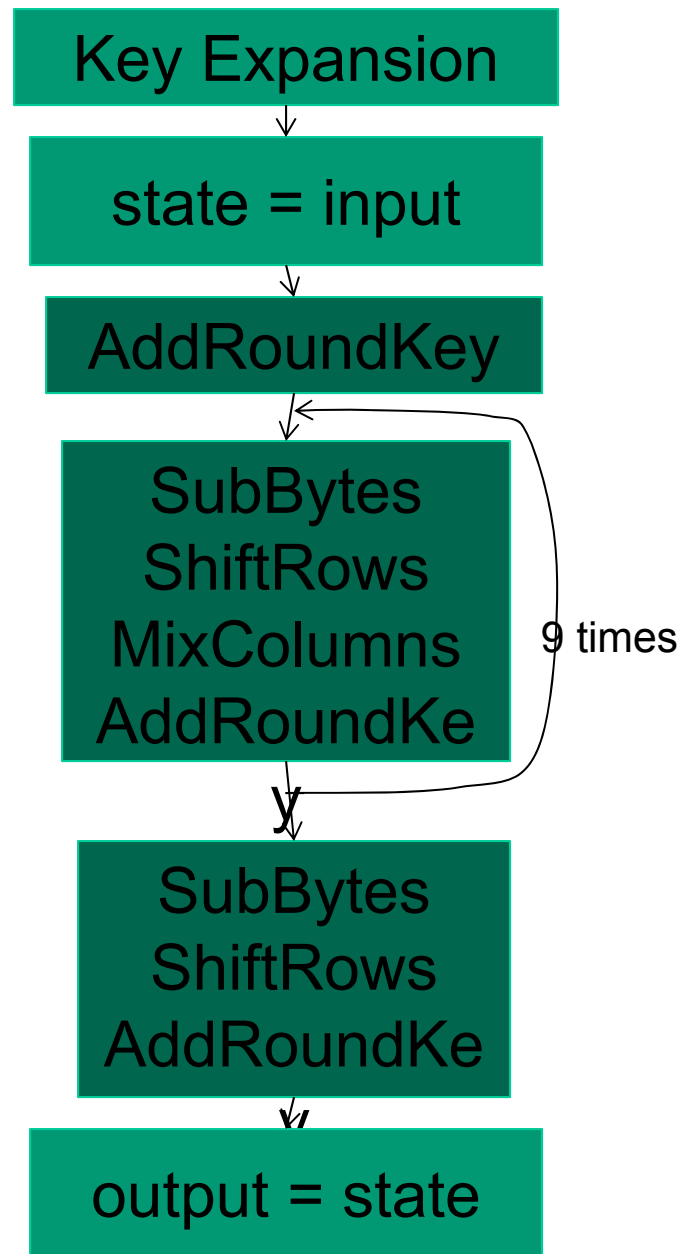
Advanced Encryption Standard

- Adopted by National Institute of Standards and Technology (NIST) on May 26, 2002.
- simple design
- high speed algorithm
- low memory costs.
- Symmetric block cipher
- byte-oriented operations
- Blocksize - 128 bits, 192 bits or 256 bits

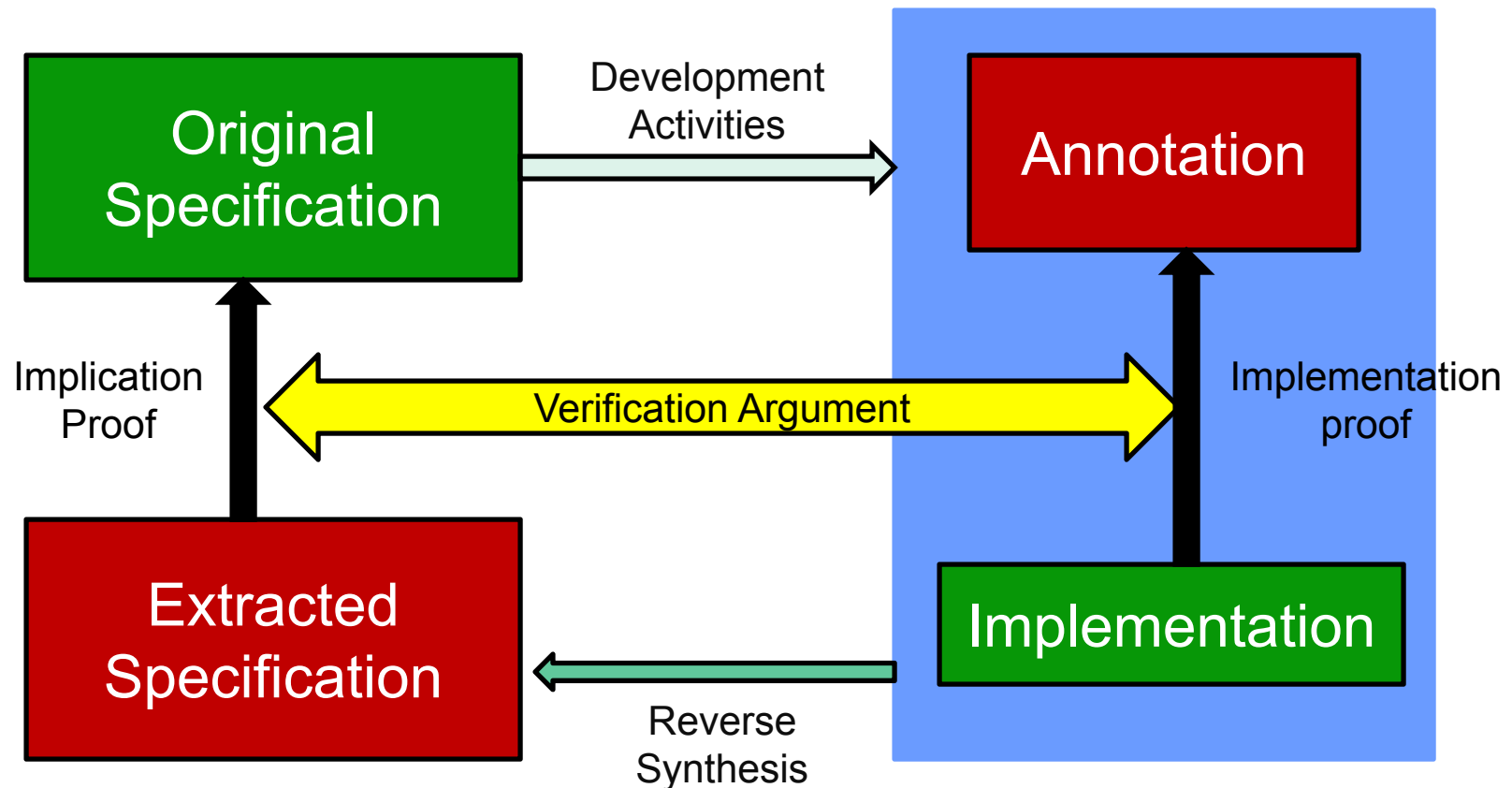
Key-Block-Round Combinations for AES

	Key Length <i>(N_k words)</i>	Block Size <i>(N_b words)</i>	Number of Rounds <i>(N_r)</i>
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

AES-128



AES Experience 1: Verification using Reverse Synthesis



Reverse Synthesis

Specification Extraction using Reverse Synthesis

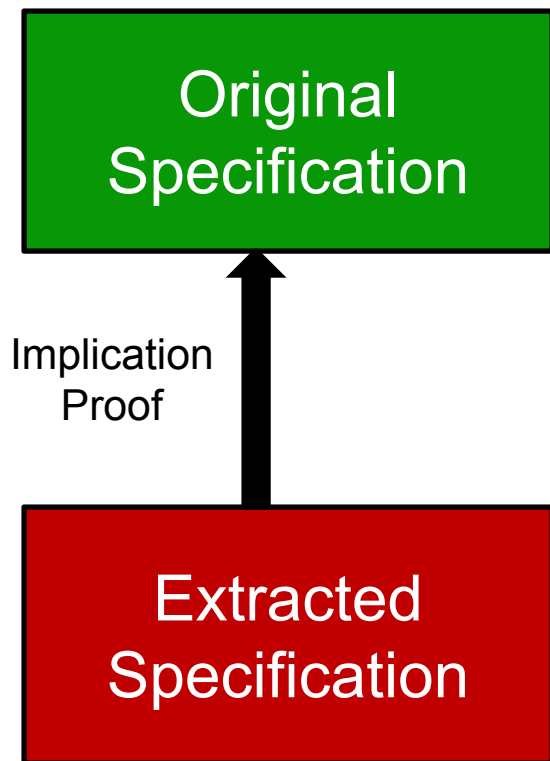
- Architectural and direct mapping
- Component reuse
- Model synthesis

Refactoring

Refactor a program

- to reduce complexity
- reduce its efficiency
- does not change its functionality
- Two stages to use refactoring-
 - Implementation proof
 - Implication Proof

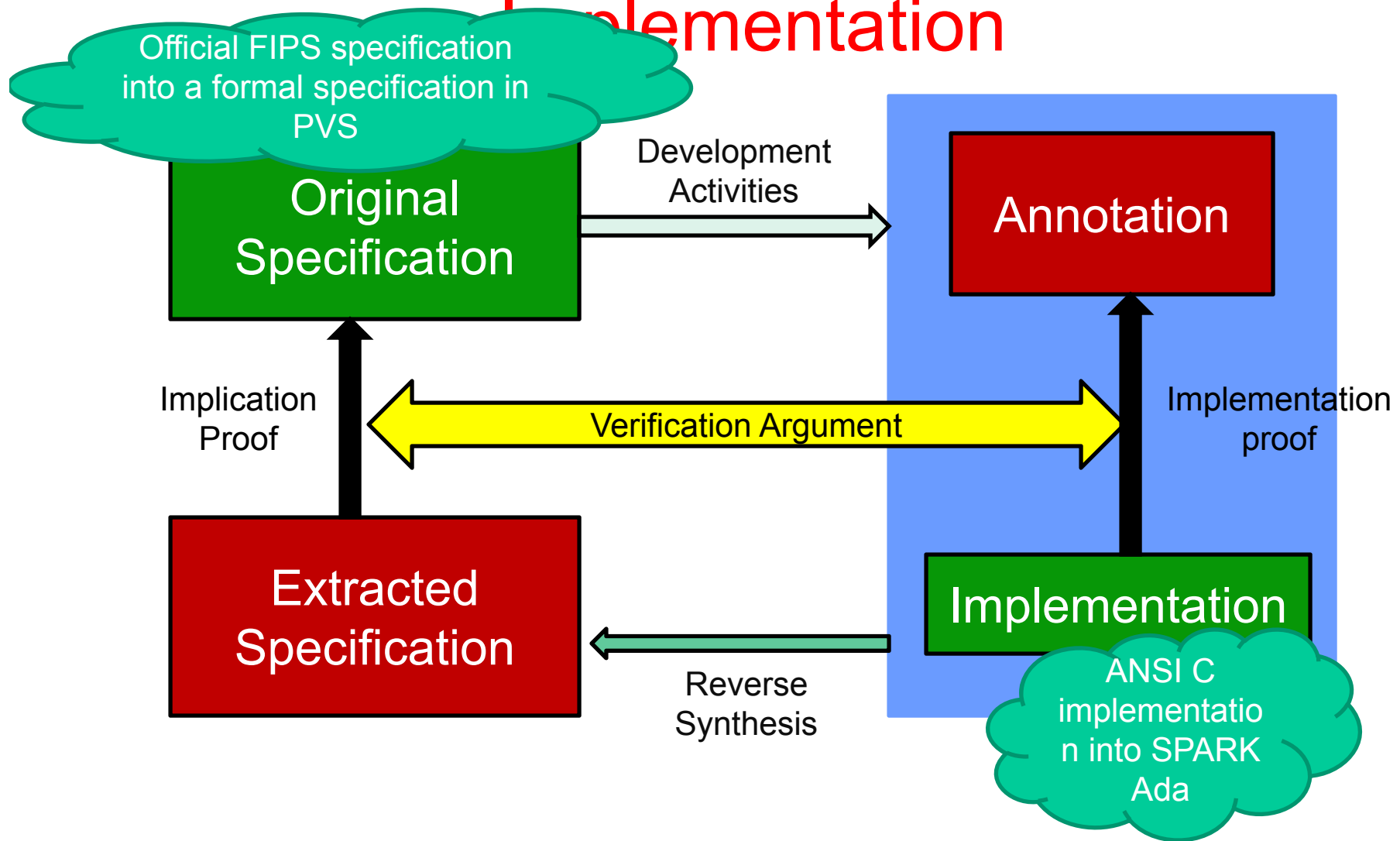
Implication Proof



Extracted Specification
→ Original
Specification

- $\text{Pre}_{\text{Original}} \Rightarrow \text{Pre}_{\text{Extracted}}$
- $\text{Post}_{\text{Extracted}} \Rightarrow \text{Post}_{\text{Original}}$

Verification of the AES Implementation



Refactoring for Implication proof

- Identify optimizations
- template defining the refactoring transformation to reverse the optimization
- proved them to be semantics-preserving
- applied the transformations

Refactoring process

Optimizations in AES to create implementation

- Loop unrolling
- Word packing
- Table lookup
- Function inlining

Loop unrolling

```
Cipher(word in[4], word out[4], word  
      w[4*(11)])
```

```
Begin
```

```
  word state[4]
```

```
  state = in
```

```
  AddRoundKey(state, w[0, 3])
```

```
  SubBytes(state)
```

```
  ShiftRows(state)
```

```
  MixColumns(state)
```

```
  AddRoundKey(state, w[4,7])
```

```
  SubBytes(state)
```

```
  ShiftRows(state)
```

```
  MixColumns(state)
```

```
  AddRoundKey(state, w[8,11])
```

```
  ...
```

```
  SubBytes(state)
```

```
  ShiftRows(state)
```

```
  AddRoundKey(state, w[40,43])
```

```
  out = state
```

```
end
```

```
Cipher(word in[4], word out[4], word  
      w[4*(11)])
```

```
begin
```

```
  byte state[4,Nb]
```

```
  state = in
```

```
  AddRoundKey(state, w[0, 3])
```

```
  for round = 1 step 1 to 9
```

```
    SubBytes(state)
```

```
    ShiftRows(state)
```

```
    MixColumns(state)
```

```
    AddRoundKey(state, w[round*4,  
      (round+1)*4-1])
```

```
  end for
```

```
  SubBytes(state)
```

```
  ShiftRows(state)
```

```
  AddRoundKey(state, w[40,43])
```

```
  out = state
```

```
end
```


Word packing

```
Cipher(word in[4], word out[4], word w[4*(11)])
begin
    word state[4]
    state = in
    AddRoundKey(state, w[0, 3])
    for round = 1 step 1 to Nr-1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*4,
            (round+1)*4-1])
    end for

    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[40,43])
    out = state
end
```

```
Cipher(byte in[4*4], byte out[4*4], word w[4*(11)])
begin
    byte state[4,4]
    state = in
    AddRoundKey(state, w[0, 3])
    for round = 1 step 1 to 9
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*4,
            (round+1)*4-1])
    end for

    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[40,43])
    out = state
end
```

Table lookup

```
SubBytes(byte state [4*4])  
{  
    for i = 0 to 15  
        State[i] = SBox[i]  
}
```

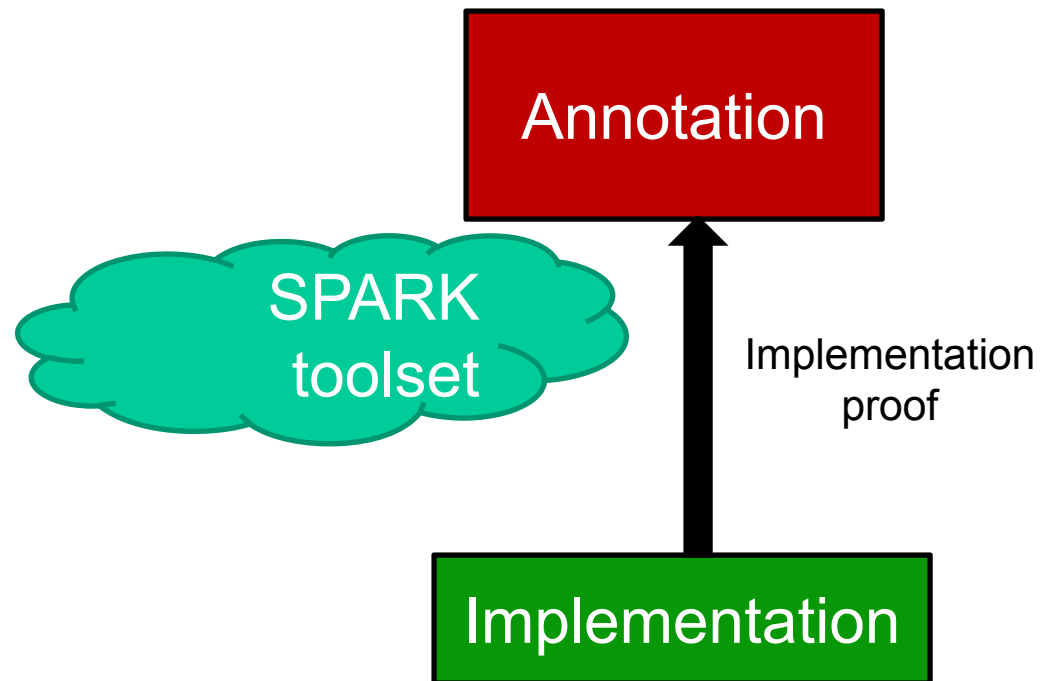
```
SubBytes(byte state [4*4])  
{  
    for i = 0 to 15  
        State[i] = compute(i);  
}
```

Function inlining

- Finding cloned code fragments - removed replicated or similar proof obligations in the implementation proof.
- Aligned the code structure
- implication proof was easier to be constructed.
- Factored nine specified functions, each of which was quite small.
- source code size increased
- conceptual complexity was reduced

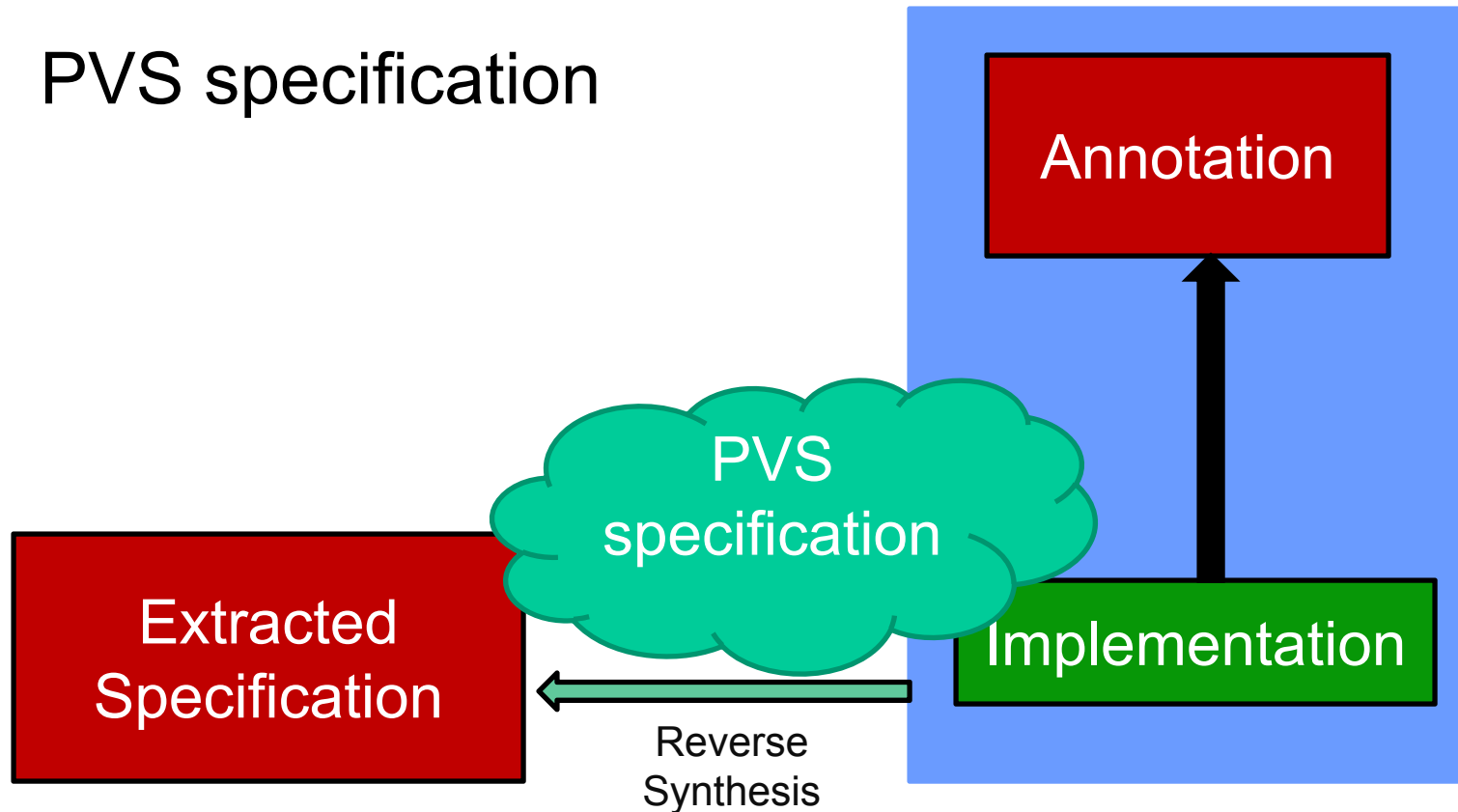
Implementation Proof

- SPARK toolset



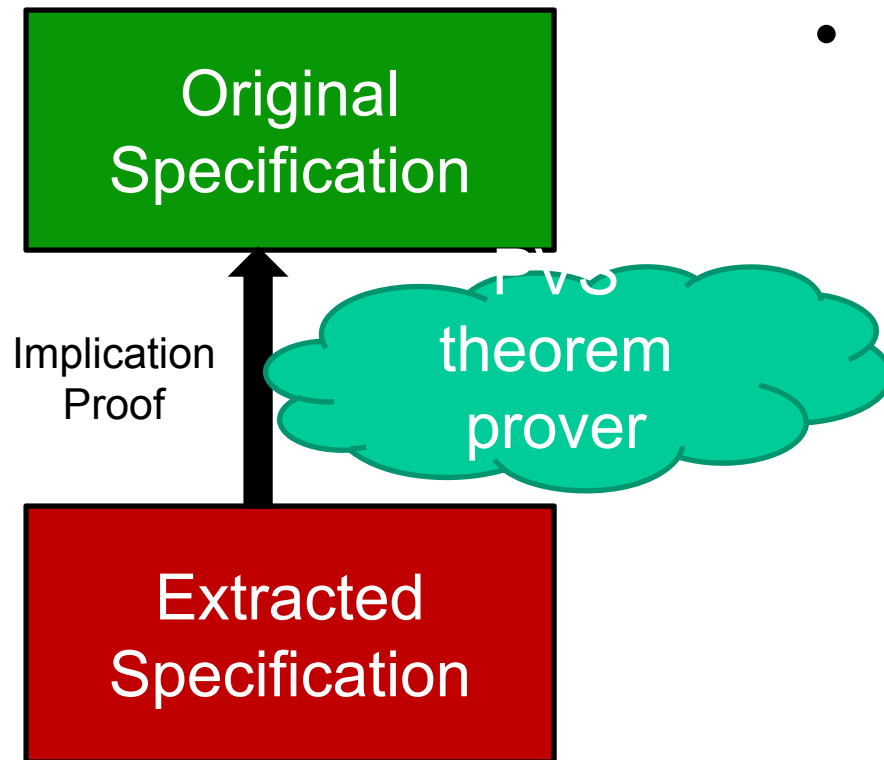
Specification Extraction

- PVS specification



Implication Proof

- PVS theorem prover



AES Experience 2: Verifying Functional Equivalence of two AES Implementations

- For low level software the following do not perform well
 - data-slicing
 - data-abstraction
- Bit-sensitive techniques provide a good alternative.
 - Bounded Model
- The usual problem is that bit-sensitive verification approaches
 - Do not scale well
 - State-space explosion

CBMC

- CBMC is a bounded software model checking tool for ANSI-C programs
- memory locations - modelled by finite bit-vectors.
- The resulting program has a finite number of statements.
- Resulting stateless bit-vector formulas to CNF
- boolean satisfiability decision procedure
 - Safety properties hold or not using Minisat2

CBMC

- built-in checks for several common runtime errors.
- assert statements
- Assume statement
- In order to check equivalence of two C functions
 - wrapper program.
 - Input parameters - equal.
 - outputs - checked for equivalence

Equivalence of two implementations

- mapping inputs from one implementation to the other
- In cases of AES where the standard defines values of constants
 - merge tables and arrays from both implementations
 - the computation of the look-up done once

Equivalence of two implementations

- Verification of three parts of AES independently
- Key Generation
 - Mapping between different bits of round key array
 - round keys generated is input for both implementations

Assert (fkey[r*4 +j] == res)

Reference impl.

00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33



Mike Scott's impl.

0	0	0	0	1	11	1	1	2	...
0	1	2	3	0		2	3	0	

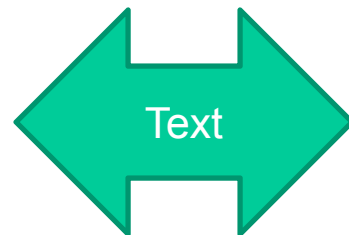
32 bit type

Equivalence of two implementations

- Encryption
 - Mapping of input encoding
 - one round of encryption for both algorithms.
 - outputs should be equal.
 - number of rounds is iteratively increased to up to 4
 - an inductive schema was used:
 - The base - get equal inputs
 - The inductive step - equal up to the i -th round \rightarrow produce equal results in round $i+1$.

Reference impl.

00	01	02	03
10	11	12	13
20	21	22	23
30	31	32	33



Mike Scott's impl.

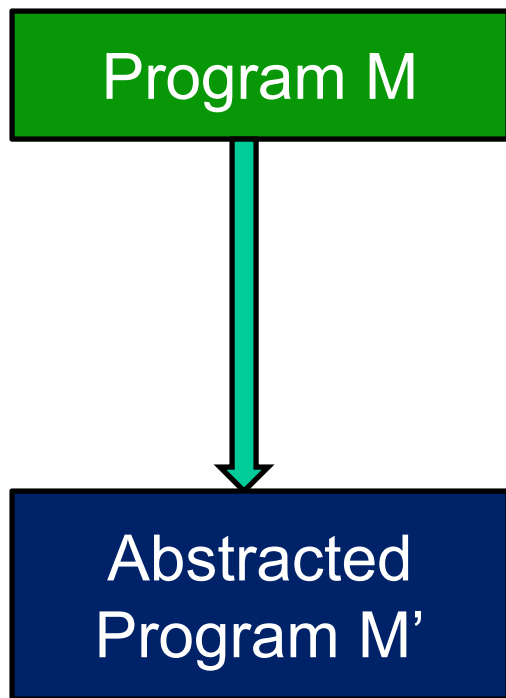
0	0	0	0	1	11	1	1	2	...
0	1	2	3	0		2	3	0	

8 bit type

Equivalence of two implementations

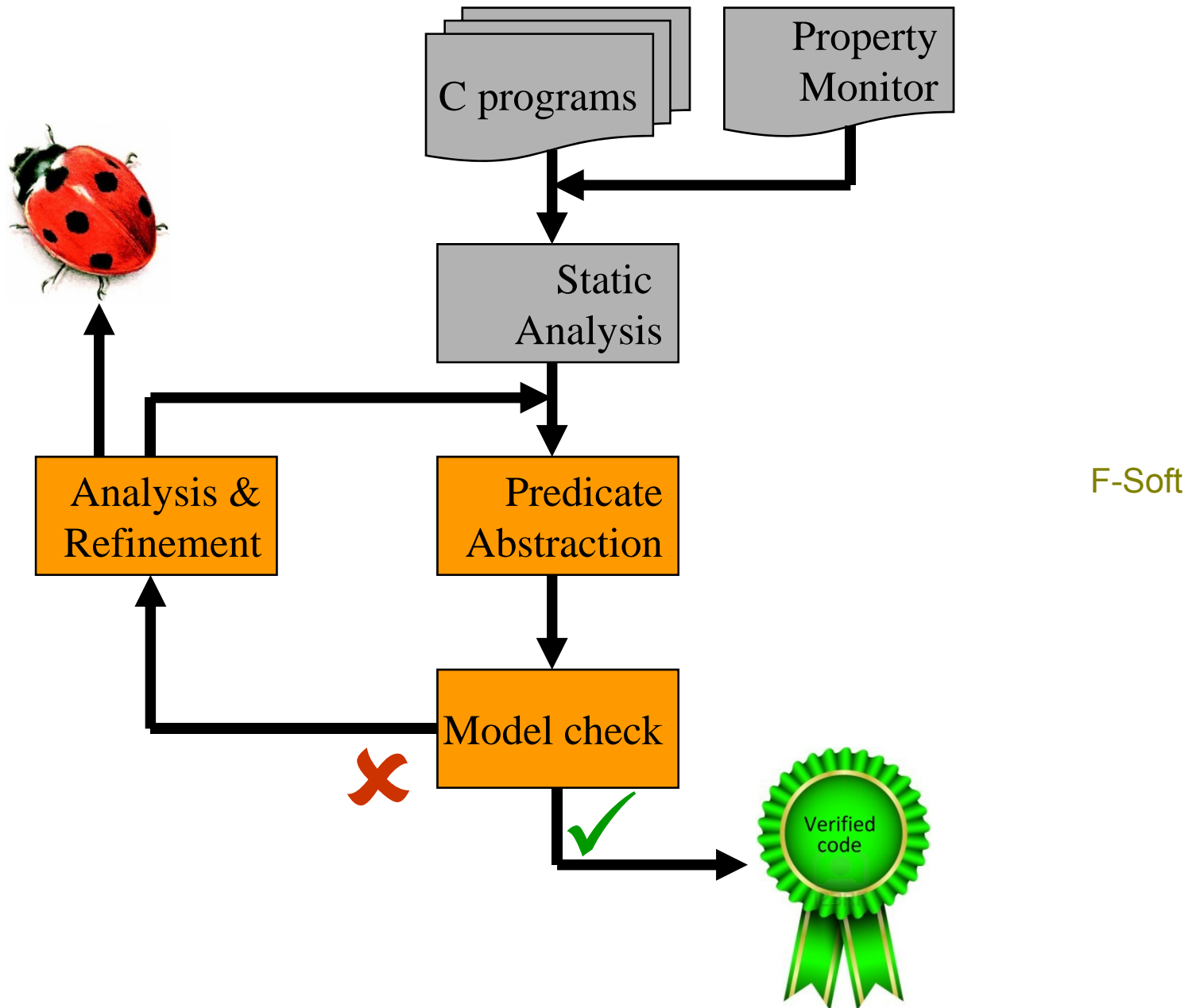
- Decryption
 - structural dissimilarity
 - Generation of backward round keys - expensive

AES Experience 3: The CEGAR attempt

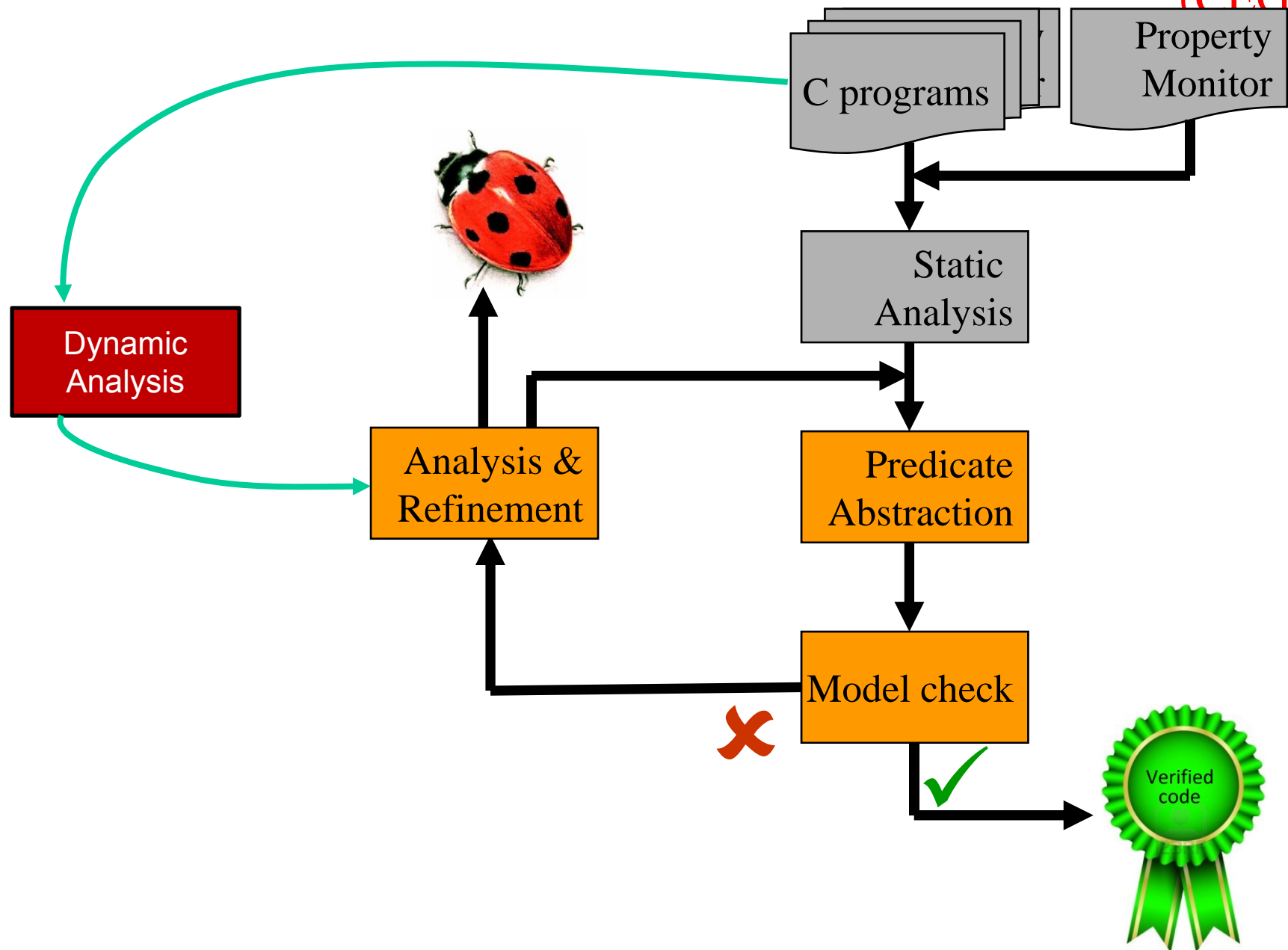


- Predicate Abstraction Reminder
 - Abstracts data by keeping track of certain predicates
 - Each predicate given a Boolean variable in abstract model
 - $M \models p \rightarrow M' \models p$

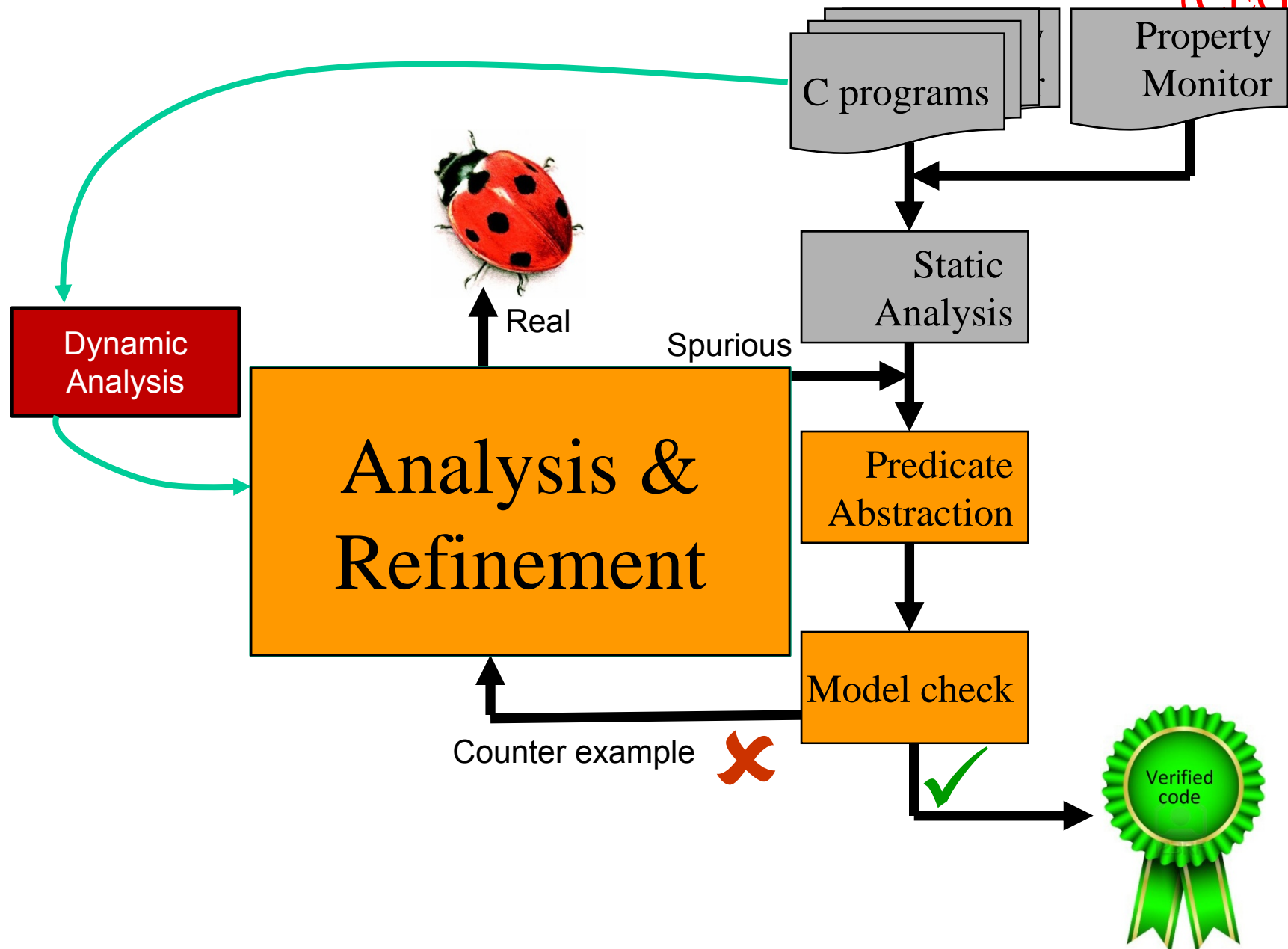
Counterexample Guided Abstraction and Refinement Loop (CEGAR)



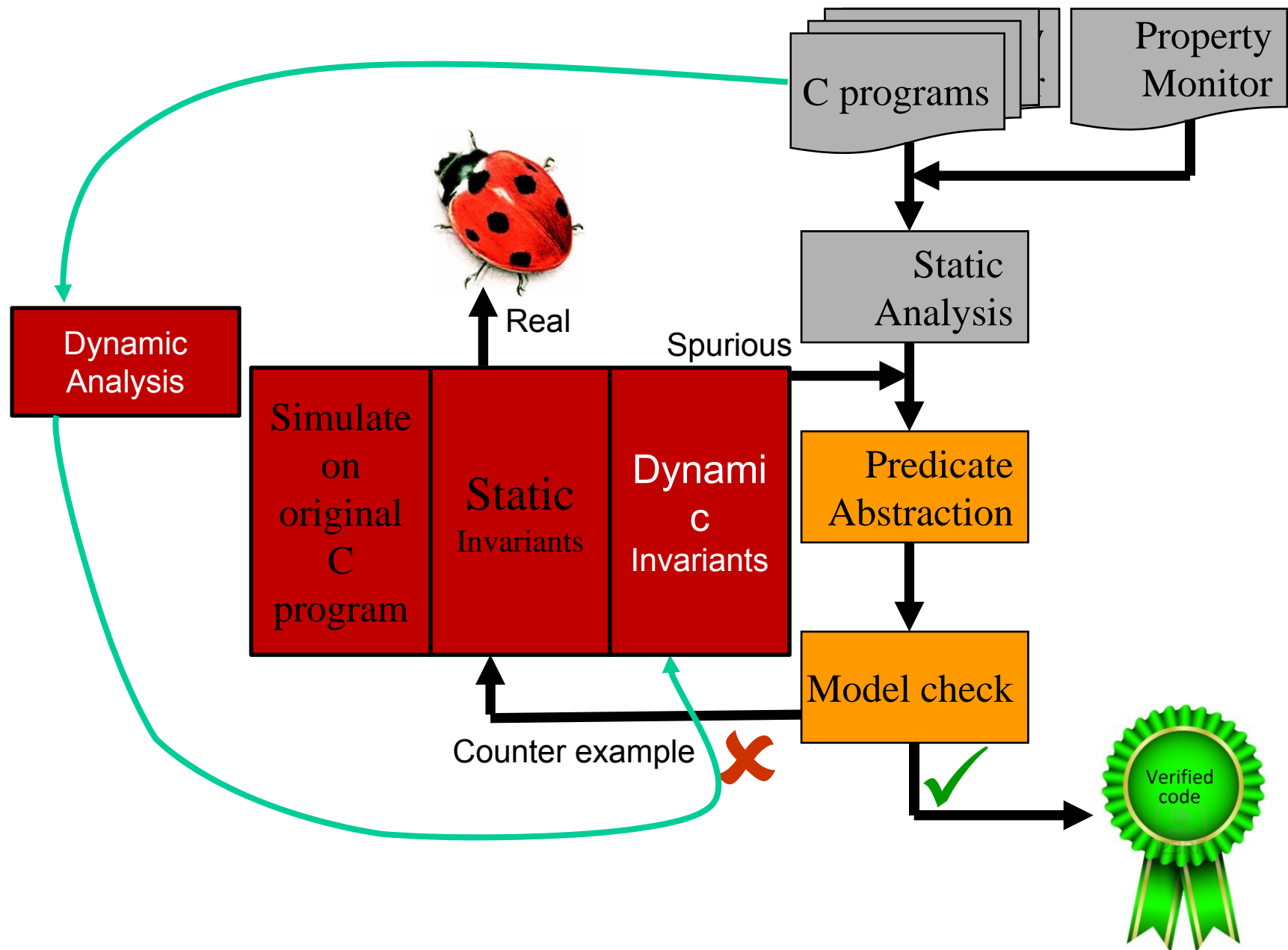
Counterexample Guided Abstraction and Refinement Loop (CEGAR)



Counterexample Guided Abstraction and Refinement Loop (CEGAR)



Counterexample Guided Abstraction and Refinement Loop (CEGAR)



Current Research

- Dynamic Invariant based verification of AES
 - Using Daikon to generate invariants
 - Daikon uses machine learning to generate invariants from program traces
 - Invariants are expressed as preconditions and post-conditions on procedures
 - Using SATABS for CEGAR using the invariants generated by Daikon

Backup Slides

Example-Simulation

```
if ( a[1] < 0 ||  
    a[0]%1000 )  
    {  
        convert(a);  
    }  
  
    sort(a);  
    if( a[0] < 0 )  
        printf("error");  
    assert( a[0] >= 0 );
```

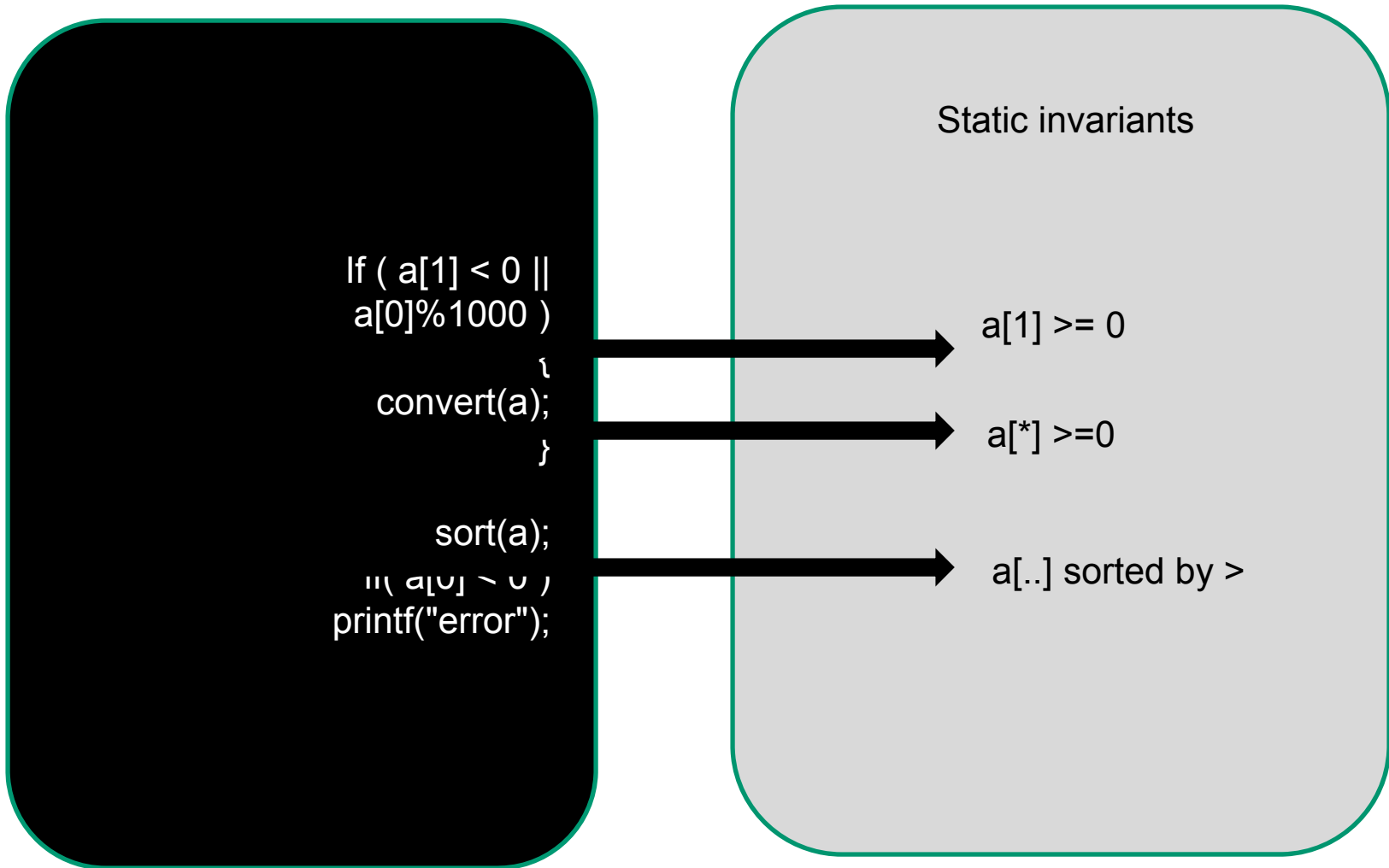
Simulate on original C program

Simulate on original C program

SATABS - 96 iterations

Failed to verify.

Example-Static invariants



Example-Dynamic invariants

```
if ( a[1] < 0 ||  
    a[0]%1000 )  
{  
    convert(a);  
}  
  
sort(a);  
if( a[0] < 0 )  
printf("error");
```

Dynamic invariants

size(a[..]) == 5 {1+}

a[..] >= orig(a[..]) (elementwise) {0.9995+}
a[..] % orig(a[..]) == 0 (elementwise) {1+}

size(a[..]) == 5 {1+}

a[..] sorted by > {0.9995+}

AES Algorithm

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)],Nk)
```

```
Cipher(byte in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)])
```

```
begin
```

```
    byte state[4,Nb]
```

```
    state = in
```

```
    AddRoundKey(state, w[0, Nb-1])
```

```
    for round = 1 step 1 to Nr-1
```

```
        SubBytes(state)
```

```
        ShiftRows(state)
```

```
        MixColumns(state)
```

```
        AddRoundKey(state,w[round*Nb,(round+1)*Nb-1])
```

```
    end for
```

```
    SubBytes(state)
```

```
    ShiftRows(state)
```

```
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
```

```
    out = state
```

```
end
```


AES Algorithm

KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)],Nk)

```
Cipher(byte in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)])  
begin  
    byte state[4,Nb]  
    state = in  
    AddRoundKey(state, w[0, Nb-1])  
    for round = 1 step 1 to Nr-1  
        SubBytes(state)  
        ShiftRows(state)  
        MixColumns(state)  
        AddRoundKey(state,w[round*Nb,(round+1)*Nb-1])  
    end for  
    SubBytes(state)  
    ShiftRows(state)  
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])  
    out = state  
end
```

AES Algorithm - Key Expansion

SubWord()

- four-byte input word
- applies the S-box

RotWord()

- $[a_0, a_1, a_2, a_3] \rightarrow [a_1, a_2, a_3, a_0]$.

Rcon[i]

- $[x_{i-1}, \{00\}, \{00\}, \{00\}]$,

AES Algorithm - Key Expansion

```
for i ← 0 to 3
  do w[i] ← (key[4i], key[4i+1], key[4i+2], key[4i+3])
for i ← 4 to 43
  temp ← w[i-1]
  if i ≡ 0 (mod 4)
    then temp ← SubWord(RotWord(temp)) + Rcon[i/4]
  w[i] ← w[i-4] + temp
return(w[0]..w[43])
```

AES Algorithm

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)],Nk)
```

```
Cipher(byte in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)])
```

```
begin
```

```
    byte state[4,Nb]
```

```
    state = in
```

```
    AddRoundKey(state, w[0, Nb-1])
```

```
    for round = 1 step 1 to Nr-1
```

```
        SubBytes(state)
```

```
        ShiftRows(state)
```

```
        MixColumns(state)
```

```
        AddRoundKey(state,w[round*Nb,(round+1)*Nb-1])
```

```
    end for
```

```
    SubBytes(state)
```

```
    ShiftRows(state)
```

```
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
```

```
    out = state
```

```
end
```

AES Algorithm

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)],Nk)
```

```
Cipher(byte in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)])
```

```
begin
```

```
    byte state[4,Nb]
```

```
    state = in
```

```
    AddRoundKey(state, w[0, Nb-1])
```

```
    for round = 1 step 1 to Nr-1
```

```
        SubBytes(state)
```

```
        ShiftRows(state)
```

```
        MixColumns(state)
```

```
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
```

```
    end for
```

```
    SubBytes(state)
```

```
    ShiftRows(state)
```

```
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
```

```
    out = state
```

```
end
```

AES Algorithm - AddRoundKey

State

41	45	49	4D
42	46	4A	4E
43	47	4B	4F
44	48	4C	50

Expanded Key w[0] → w[4]

11	22	33	44
55	66	77	88
99	00	AA	BB
CC	DD	EE	FF

After AddRoundKey

41 ↗ 11	45 ↗ 55	49 ↗ 99	4D ↗ CC
42 ↗ 22	46 ↗ 66	4A ↗ 00	4E ↗ DD
43 ↗ 33	47 ↗ 77	4B ↗ AA	4F ↗ EE
44 ↗ 44	48 ↗ 88	4C ↗ BB	50 ↗ FF



50	10	D0	81
60	20	4A	93
70	30	E1	A1
00	C0	F7	AF

AES Algorithm

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)],Nk)
```

```
Cipher(byte in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)])
```

```
begin
```

```
    byte state[4,Nb]
```

```
    state = in
```

```
    AddRoundKey(state, w[0, Nb-1])
```

```
    for round = 1 step 1 to Nr-1
```

```
        SubBytes(state)
```

```
        ShiftRows(state)
```

```
        MixColumns(state)
```

```
        AddRoundKey(state,w[round*Nb,(round+1)*Nb-1])
```

```
    end for
```

```
    SubBytes(state)
```

```
    ShiftRows(state)
```

```
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
```

```
    out = state
```

```
end
```

AES Algorithm - SubBytes

- SubBytes is the SBOX for AES
- For every value of b there is a unique value for b'
 - It is faster to use a substitution table (and easier).

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

AES Algorithm - SubBytes

State

50	10	D0	81
60	20	4A	93
70	30	E1	A1
00	C0	F7	AF



Sbox(50)	Sbox(10)	Sbox(D0)	Sbox(81)
Sbox(60)	Sbox(20)	Sbox(4A)	Sbox(93)
Sbox(70)	Sbox(30)	Sbox(E1)	Sbox(A1)
Sbox(00)	Sbox(C0)	Sbox(F7)	Sbox(AF)

$$b'_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i$$



= {01100011}

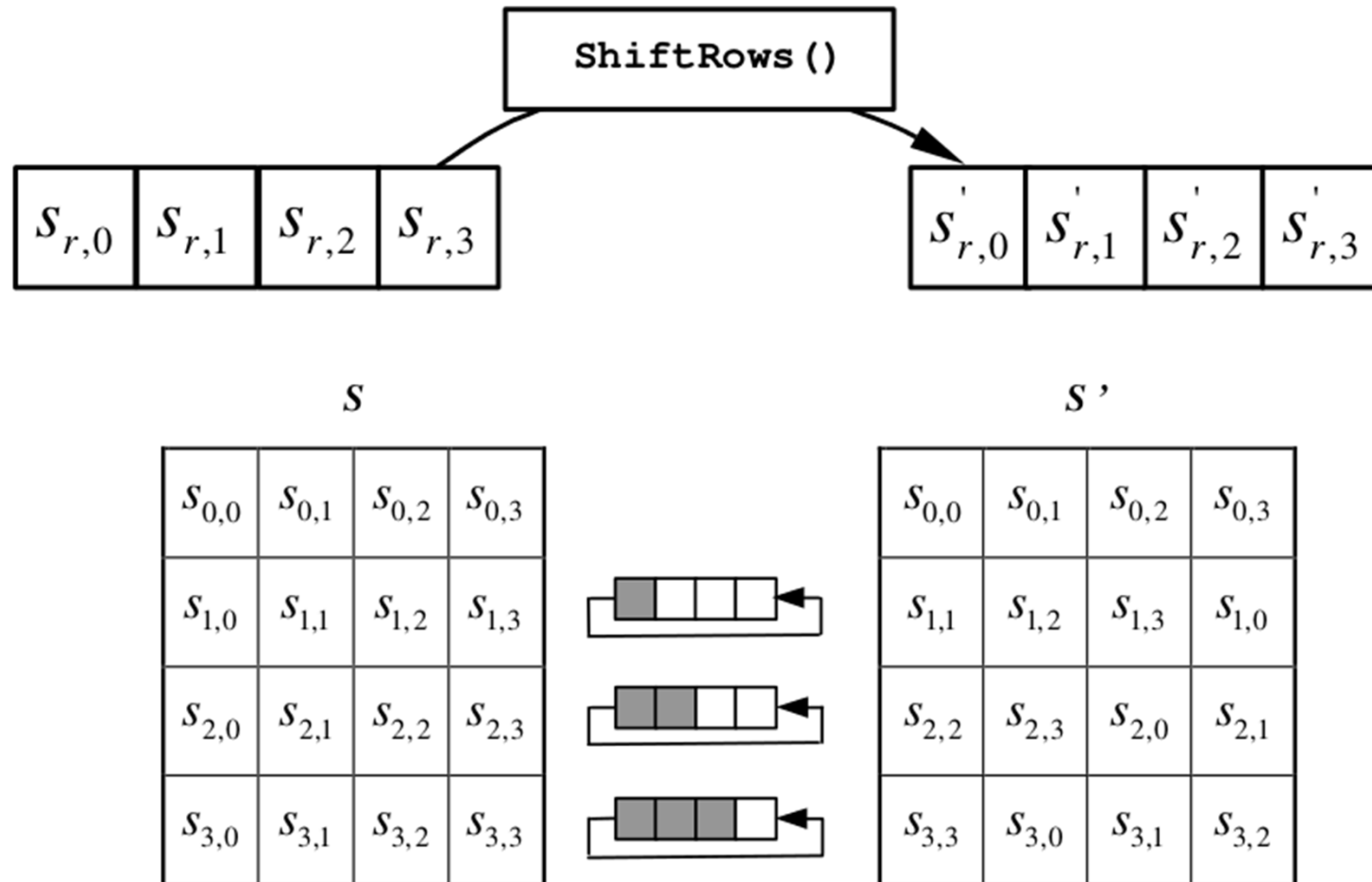
53	CA	70	0C
D0	B7	D6	DC
51	04	F8	32
63	BA	68	79

AES Algorithm

```
KeyExpansion(byte key[4*Nk],word w[Nb*(Nr+1)],Nk)

Cipher(byte in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]
    state = in
    AddRoundKey(state, w[0, Nb-1])
    for round = 1 step 1 to Nr-1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state,w[round*Nb,(round+1)*Nb-1])
    end for
    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
    out = state
end
```

AES Algorithm - ShiftRows



AES Algorithm - ShiftRows

- Simple routine which performs a left shift rows 1, 2 and 3 by 1, 2 and 3 bytes respectively

Before Shift Rows

53	CA	70	0C
D0	B7	D6	DC
51	04	F8	32
63	BA	68	79



After Shift Rows

53	CA	70	0C
B7	D6	DC	D0
F8	32	51	04
79	63	BA	68

AES Algorithm

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)],Nk)
```

```
Cipher(byte in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)])
```

```
begin
```

```
    byte state[4,Nb]
```

```
    state = in
```

```
    AddRoundKey(state, w[0, Nb-1])
```

```
    for round = 1 step 1 to Nr-1
```

```
        SubBytes(state)
```

```
        ShiftRows(state)
```

```
        MixColumns(state)
```

```
        AddRoundKey(state,w[round*Nb,(round+1)*Nb-1])
```

```
    end for
```

```
    SubBytes(state)
```

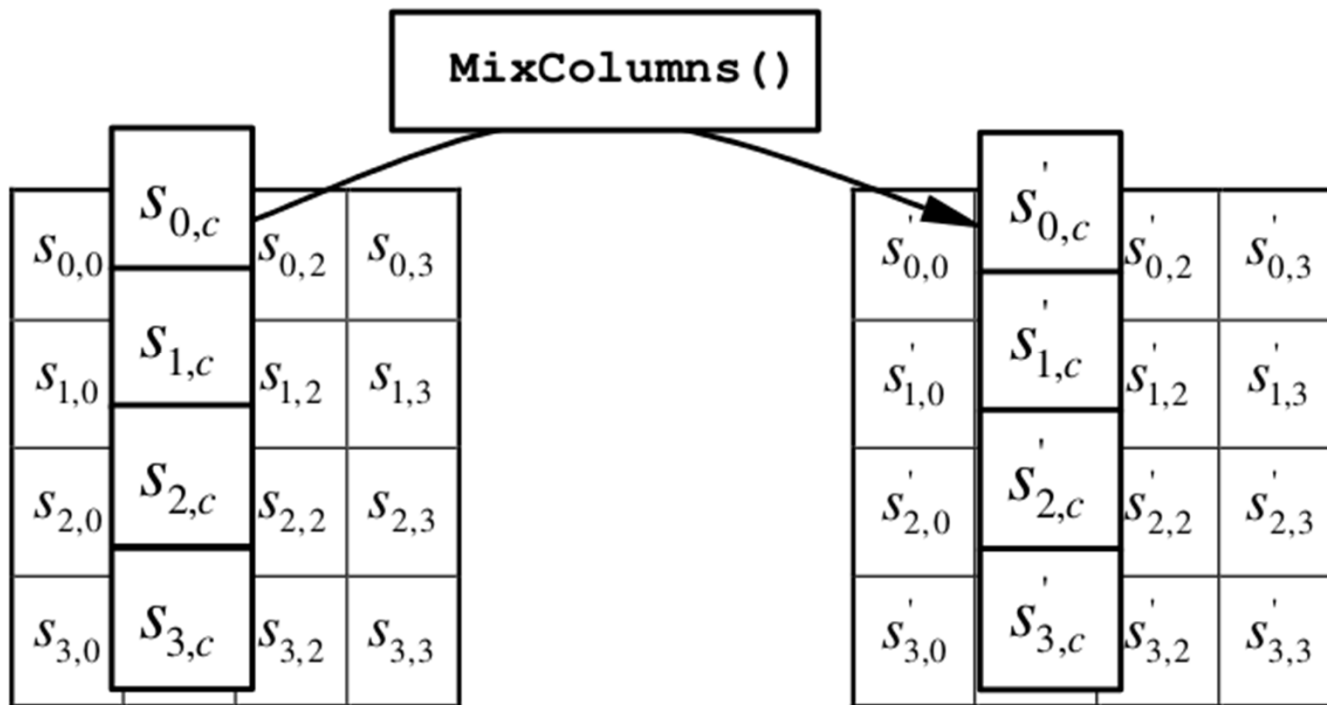
```
    ShiftRows(state)
```

```
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
```

```
    out = state
```

```
end
```

AES Algorithm - MixColumns



AES Algorithm - MixColumns

$$\begin{bmatrix} a'_0 \\ a'_1 \\ a'_2 \\ a'_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \left\{ \begin{array}{l} a'_0 = 2a_0 + 3a_1 + a_2 + a_3 \\ a'_1 = a_0 + 2a_1 + 3a_2 + a_3 \\ a'_2 = a_0 + a_1 + 2a_2 + 3a_3 \\ a'_3 = 3a_0 + a_1 + a_2 + 2a_3 \end{array} \right.$$

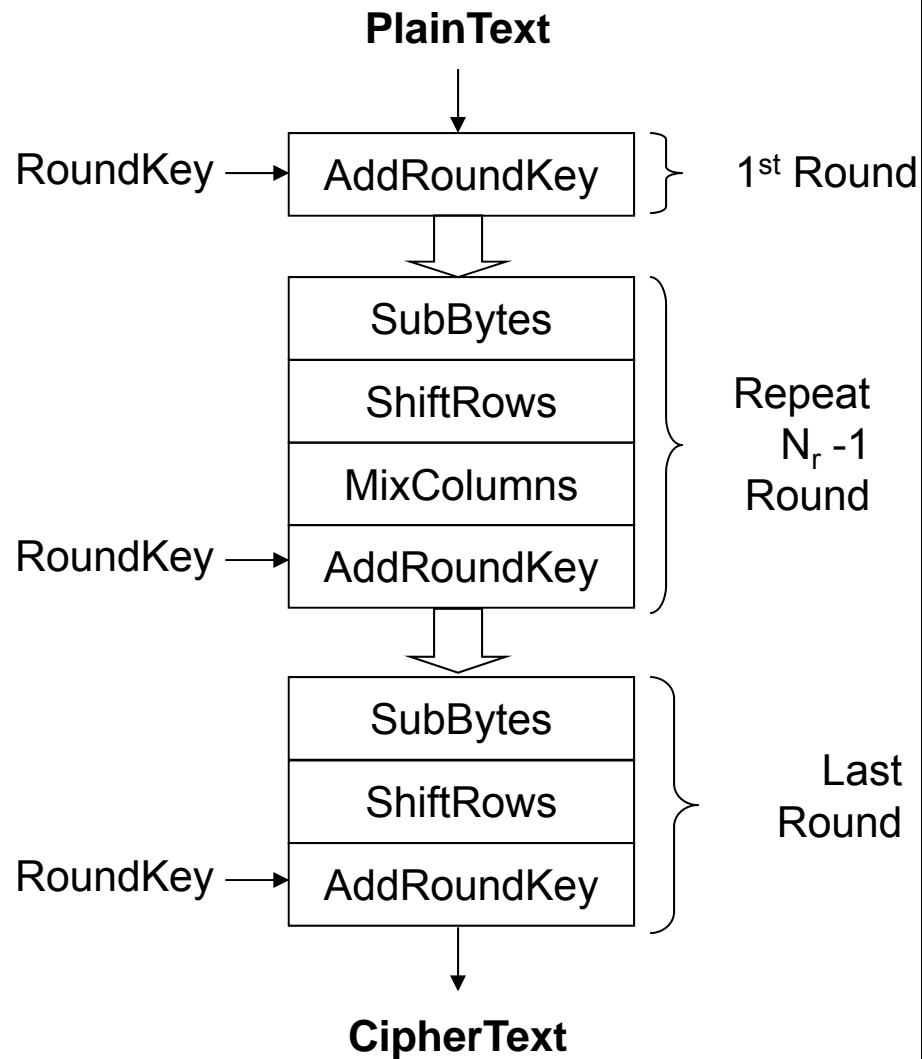
AES Algorithm

```
KeyExpansion(byte key[4*Nk], word w[Nb* (Nr+1)],Nk)

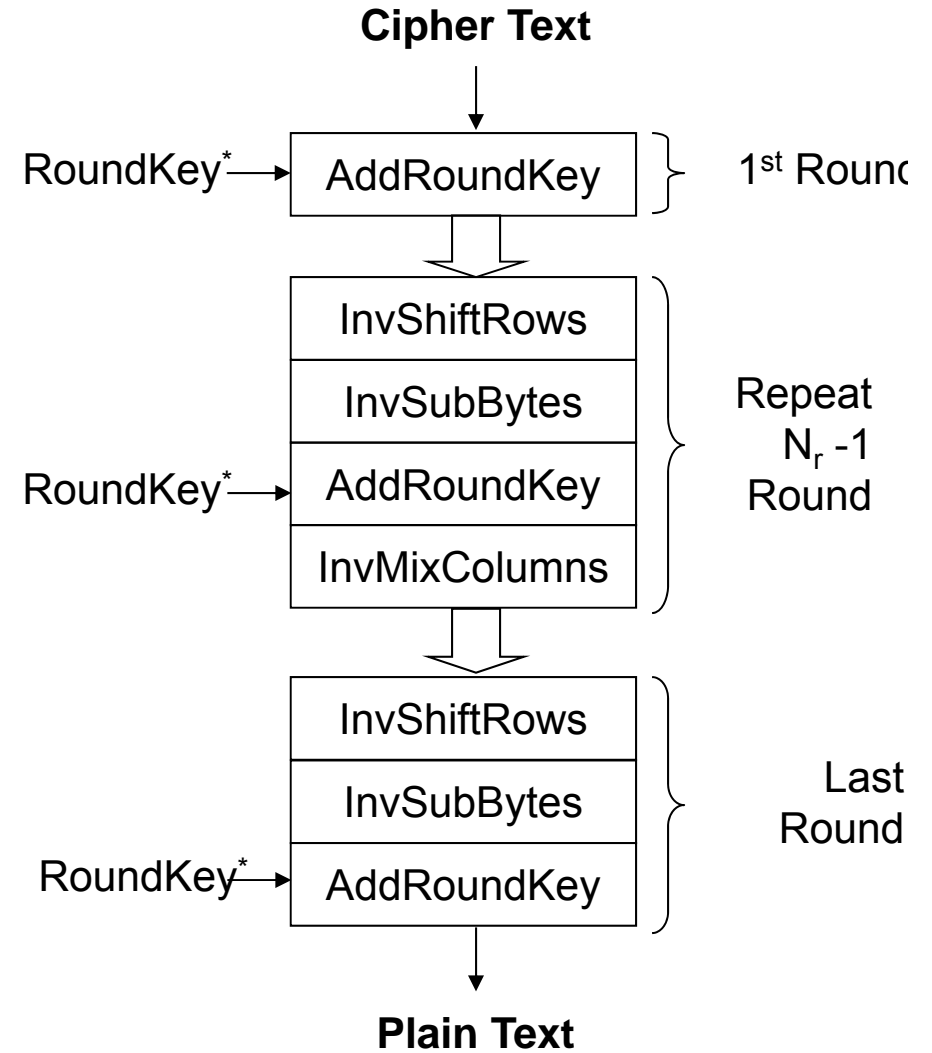
Cipher(byte in[4*Nb],byte out[4*Nb],word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]
    state = in
    AddRoundKey(state, w[0, Nb-1])
    for round = 1 step 1 to Nr
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    end for
    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
    out = state
end
```


AES Algorithm

Encryption



Decryption



* RoundKey Added in reverse order

Slide sources

- Edmund Clarke's course:

<http://www.cs.cmu.edu/~emc/15414-f11/lecture/>

- Vitaly Shmatikov's course:

http://www.cs.utexas.edu/~shmat/courses/cs395t_fall04/cs395t_home.html

- Tom Chotia's course:

<http://www.cs.bham.ac.uk/~tpc/cwi/Teaching/index.html>