Verification Everywhere: Security, Dependability, Reliability

Lenore D. Zuck

Usable Verification, May 25, 2011

“Trustworthy” Protocols: NTLM

- A suite of Microsoft security protocols
- Proves authentication, integrity, confidentiality
- Had been replaced by Kerberos unless it can’t:
  - domain controller unavailable/unreachable
  - client is not Kerberos capable
  - user remotely authenticating over the web
  - ...
- Vulnerable to a credential forwarding attack

“Trustworthy” Protocols

Chrome File Edit View History Bookmarks Window Help

The NTLM Authentication Pr | cn.cs.unibas.ch/people/tm/ | ה הלאומית הישראלית למדעים | Downloads

davenport.sourceforge.net/ntlm.html

4 Unknown 0x0000000f

The operating system build can be found by running "winver.exe"; it should give a string similar to:

```
Version 5.1 (Build 2600.xpsp_sp2_gdr.050301-1519 : Service Pack 2)
```

This yields an OS Version structure of "0x0501280a0000000f":

0x05	(major version 5)
0x01	(minor version 1; Windows XP)
0x280a	(build number 2600 in hexadecimal little-endian)
0x0000000f	(unknown/reserved)

Note that the OS Version structure and the supplied domain/workstation are optional fields. There are three versions of the Type 1 message that have been observ

1. Version 1 -- The Supplied Domain and Workstation security buffers and OS Version structure are omitted completely. In this case the message ends after t in the Open Group's ActiveX reference documentation (Section 11.2.2).
2. Version 2 -- The Supplied Domain and Workstation buffers are present, but the OS Version structure is not. The data block begins immediately after the se
3. Version 3 -- Both the Supplied Domain/Workstation buffers are present, as well as the OS Version structure. The data block begins after the OS Version st 2000, Windows XP, and Windows 2003.

The "most-minimal" well-formed Type 1 message, therefore, would be:

```
4e544c4d53535000010000000202000 0
```

This is a "Version 1" Type 1 message containing only the NTLMSSP signature, the NTLM message type, and the minimal set of flags (Negotiate NTLM and N

And th

Researcher Warns of NTLM Security Vulnerability

August 16, 2010
By eSecurityPlanet Staff
Submit Feedback »
More by Author »

Security researcher Marsh Ray says a 15-year-old vulnerability in NT LAN Manager (NTLM) and NTLMv2 is continuing to put organizations at risk.

"Awareness of the protocol vulnerability dates back to 1996 and it has been the topic of several presentations over the years at various Black Hat security conferences, Ray says," writes The Register's Dan Goodin.

"But a raft of software packages, including WebKit, Samba, and Mozilla titles, continue to be plagued by the problems, in large part because fixes tend to limit themselves to specific attack vectors at the expense of comprehensiveness," Goodin writes. "And that means that the flaw is likely of benefit to black-hat hackers."

Mozilla Seamonk
Reflection
Securityt

based buffer if given a too long
you enable NTLM authentication
er are longer
ate
user name

emote Users Conduct Authentication

uct authentication reflection attacks.
get user, will cause the target user's browser to forward NTLM authenticated requests to another application.
ded by the target user, will cause the target user's browser to forward NTLM authenticated requests to another application.

g/security/announce/2009/mfsa2009-68.html
www.mozilla.org/security/announce/2009/mfsa2009-68.html (Links to External Site)
Access control error
erlying OS: Linux (Any), UNIX (Any), Windows (Any)

Affe
Not af

Also note th
advertised as

3. RECOMMENDA

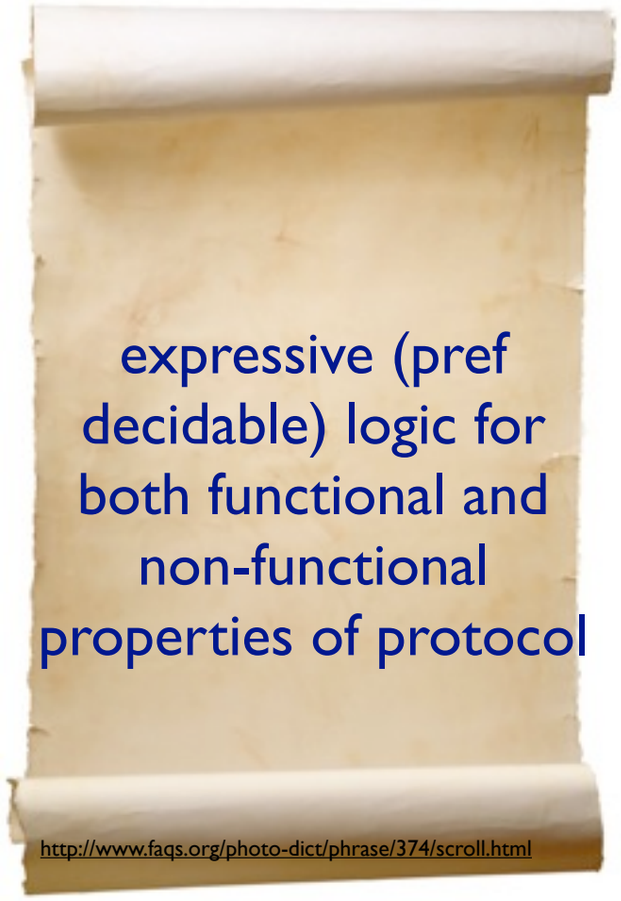
Why??

- Protocols not carefully designed (hard to obtain exact specs from English description)
- Protocols not formally verified ("we may never get a secure system, and we surely won't unless we verify it")
- Bugs take a long time to identify (usually long after deployment)
- Patching breaks backward compatibility
- and we lack ...

We lack

- Integration of verification methodologies
  - that operate on networks
  - that incorporate functional and non-functional properties
  - that implementations follow specifications
  - that check backward compatibility
- Agreed upon language(s) to formally specify the security properties we require from systems that can be verified
- Formal assumptions of attacker and attack models

Road Map



expressive (pref decidable) logic for both functional and non-functional properties of protocol

http://www.faqs.org/photo-dict/phrase/374/scroll.html

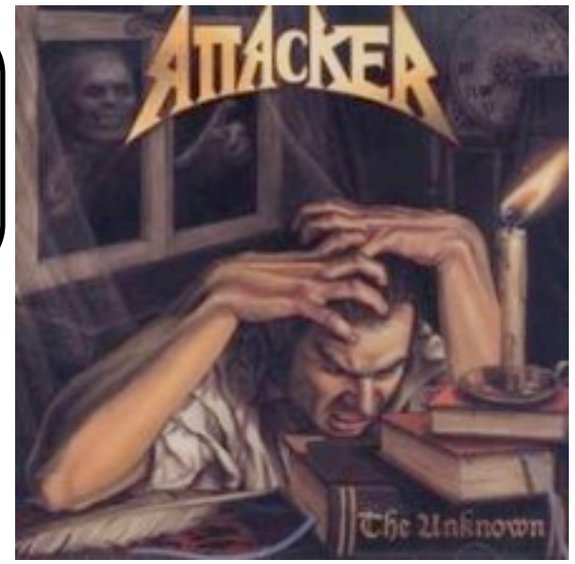
+

http://www.gilad.co.uk/writings/the-protocols-of-the-elders-of-zion-verse-2-by-gilad-atzmon.html

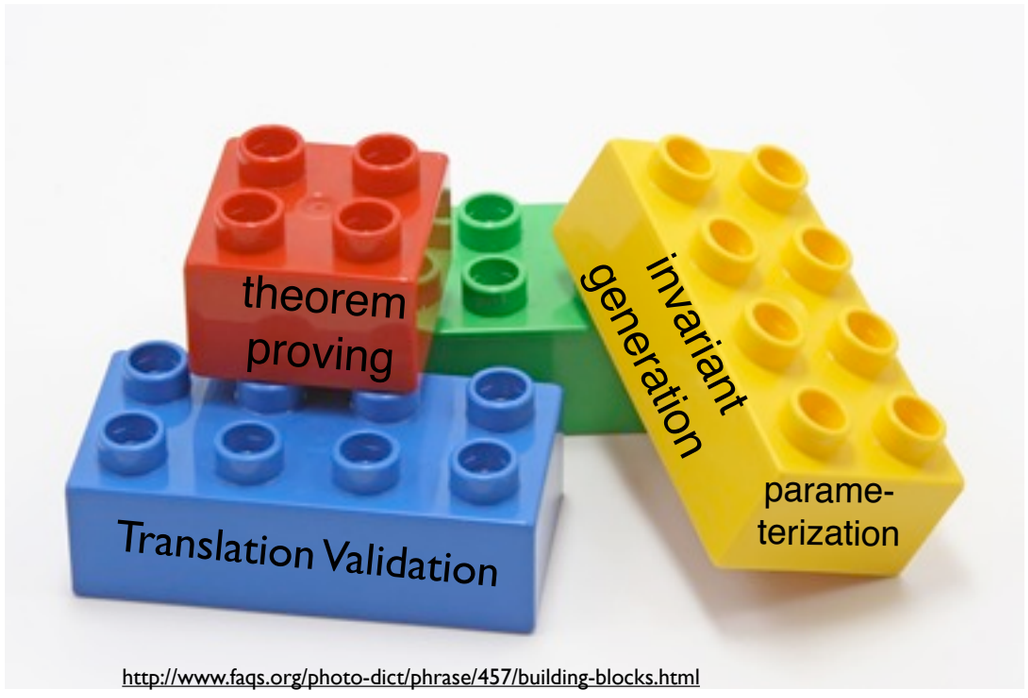
Note: Attackable!

+

http://www.alternative-zine.com/interviews/en/88



+



http://www.faqs.org/photo-dict/phrase/457/building-blocks.html

+

http://www.1stpositionmarketing.com/blog/?Tag=Twitter%20tools

=

Fully verified protocols robust against security attacks

On What There Is

- Tools to verify security protocols (Avispa, Athena, Scyther, ProVerif)
  - Cannot be easily accommodated to work on arbitrary topologies and arbitrarily large messages
- Handcrafted tools for particular protocols [Pereira, Paulson]
- Bugs found even on verified protocols (TLS)
- Implementation sometimes break security (side-channel attacks) [Bleichbaher, Kocher]

Missing: General tools to verify protocols on any topology, careful specifications of protocol requirements and attack model, proofs that implementations do not introduce new flaws

What Formal Methods offer

A variety of methodologies to help verification of:

- Protocols (arbitrary, even dynamic, topology and number of participants) even in case of attacks on network*
- Stepwise refinement (functional and non-functional properties)*

Theorem provers that allow integration of proofs about mathematics with proofs about software

On Refinement

- Techniques apply to high-level abstractions
- But it's actual code we want to verify
- Existing techniques can help verify that properties are preserved at refined code, but
- Unlike many properties, security flaws can pop up at the lower level implementation (e.g., Kocher attack on RSA) and may require new methodologies (to show that security is preserved)

Preview

- Similarities between fundamental problems in FM and security
- Verification of network security protocols is (probably) attainable using more research and a combination of (numerous!) existing tools
- The lack of formal requirements and formal (executable) specification is a major obstacle
- The model of the attacker needs be clearly defined (per protocol)
- (As in SE) Combining development and verification processes facilitates correct verified design (Design for Verification)
- Tools for security verification need to be integrated with “standard” verification tools if we are to obtain a full-fledged verification tool

An Example (where everything is needed...)

Secure RideSharing (Iskander, Lee, Mossé)

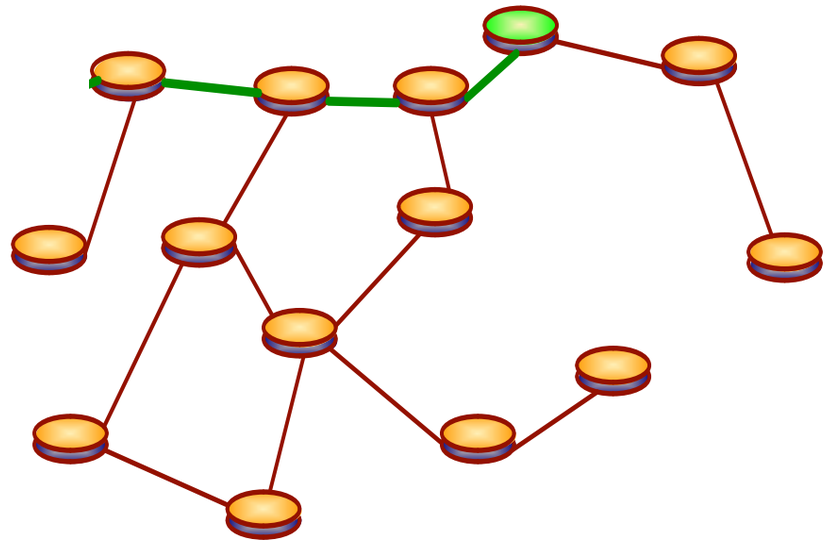
A protocol for dynamic Wireless Sensor Networks to carry out secure data aggregation to a sink node.

Protocol should satisfy:

- Privacy
- Fault tolerance
- Exact aggregation during fault-free operation

Secure RideShare

- Ad-hoc sensor network
  - Sensors are limited
    - ✦ bandwidth
    - ✦ storage
    - ✦ power
    - ✦ computation



Goal

Formal verification permeates system:

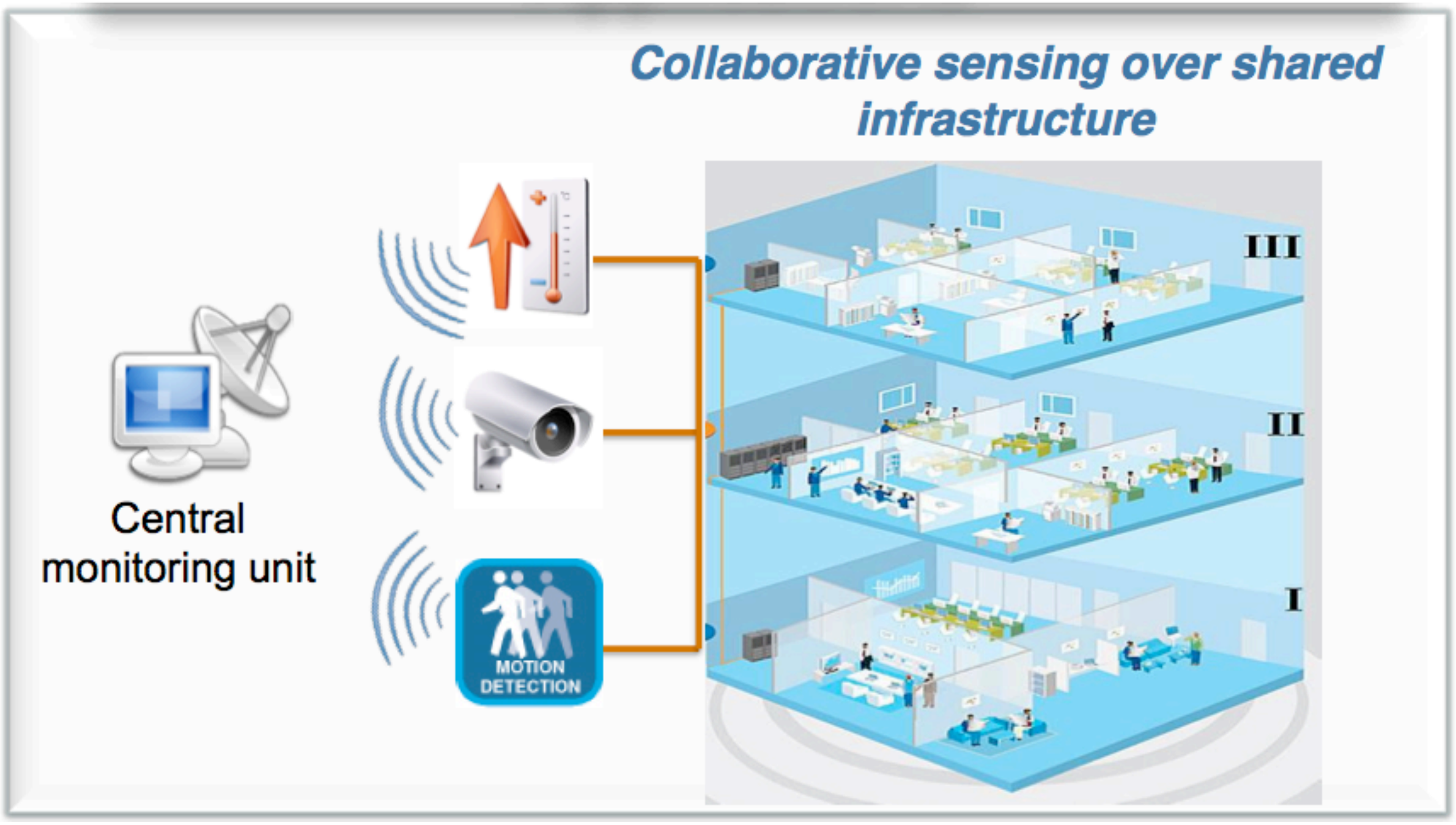
distributed resource allocation
security sensing
power management
real-time
trusted join/leave
(In)voluntary promotion/demotion
reliable delivery of data & results

Building Blocks:

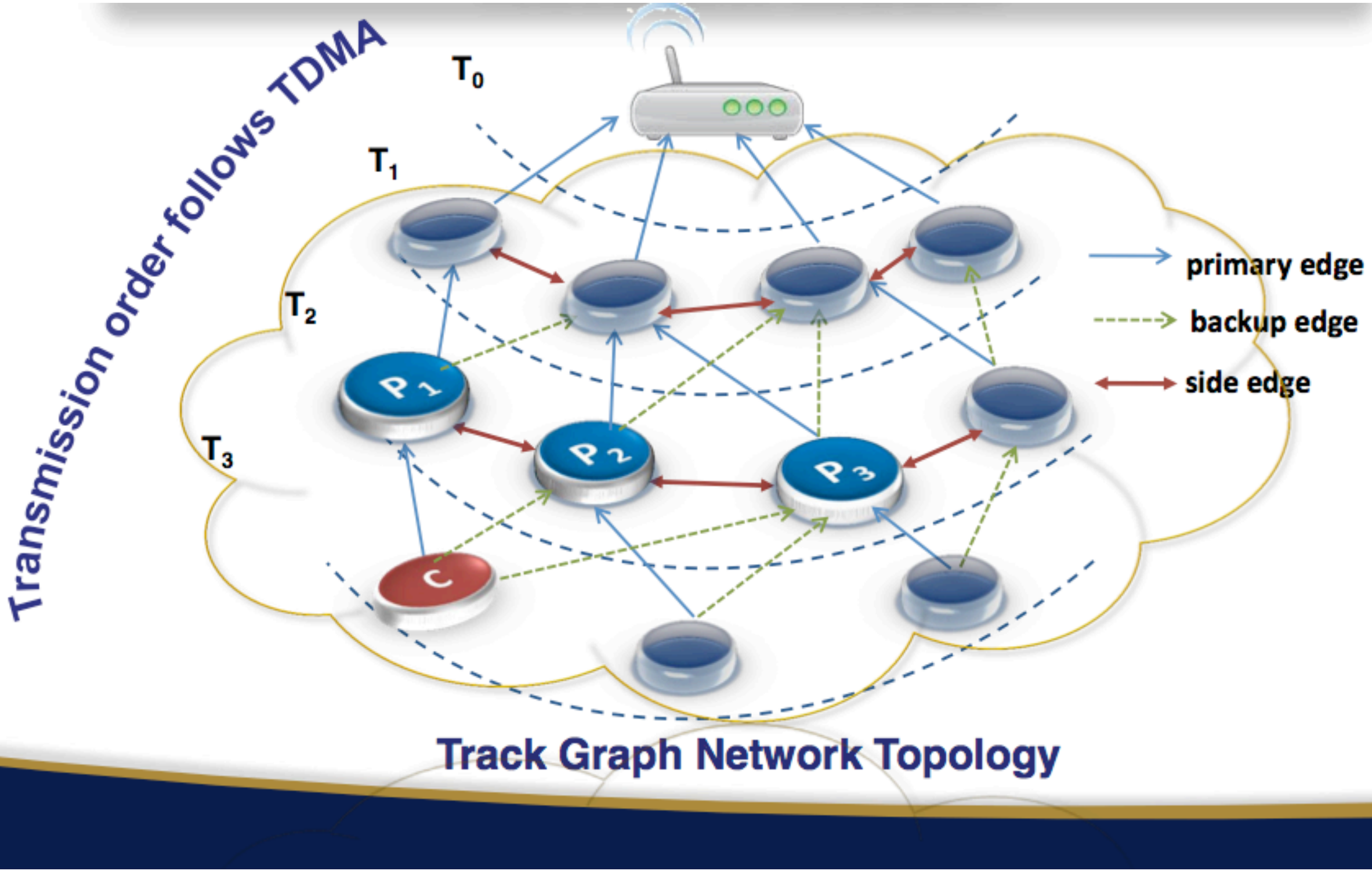
Privacy Preservation	Additively homomorphic stream cipher [1]
Robustness	Cascaded Ridesharing [2]

[1] Casteluccia, Chan, Mykletun, Tsudik 2009
[2] Gobriel, Khatab, Mossé, Brustoloni, Melhem 2006

Applications



Network Model



Attack Model

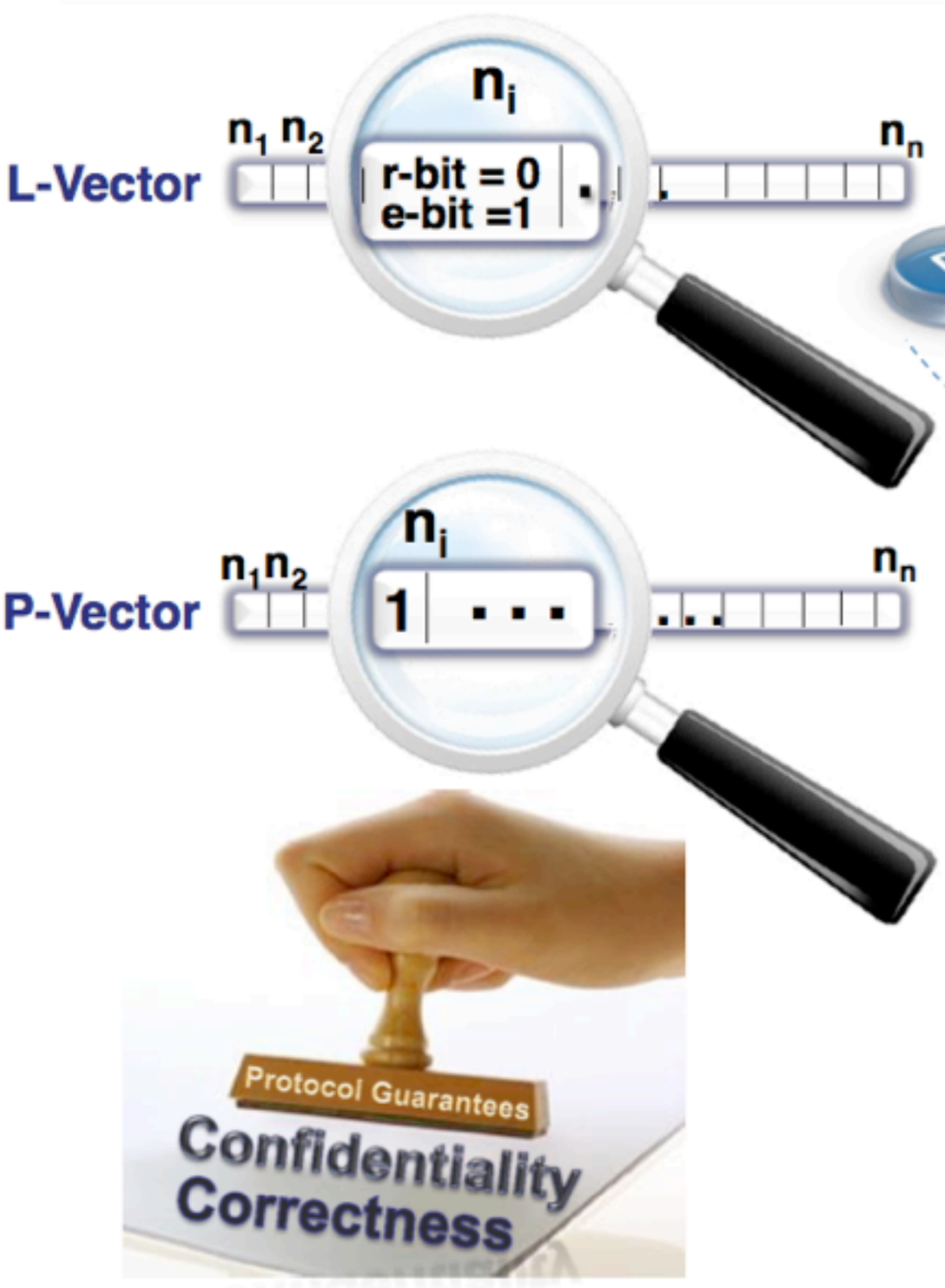
HONEST-BUT-CURIOUS

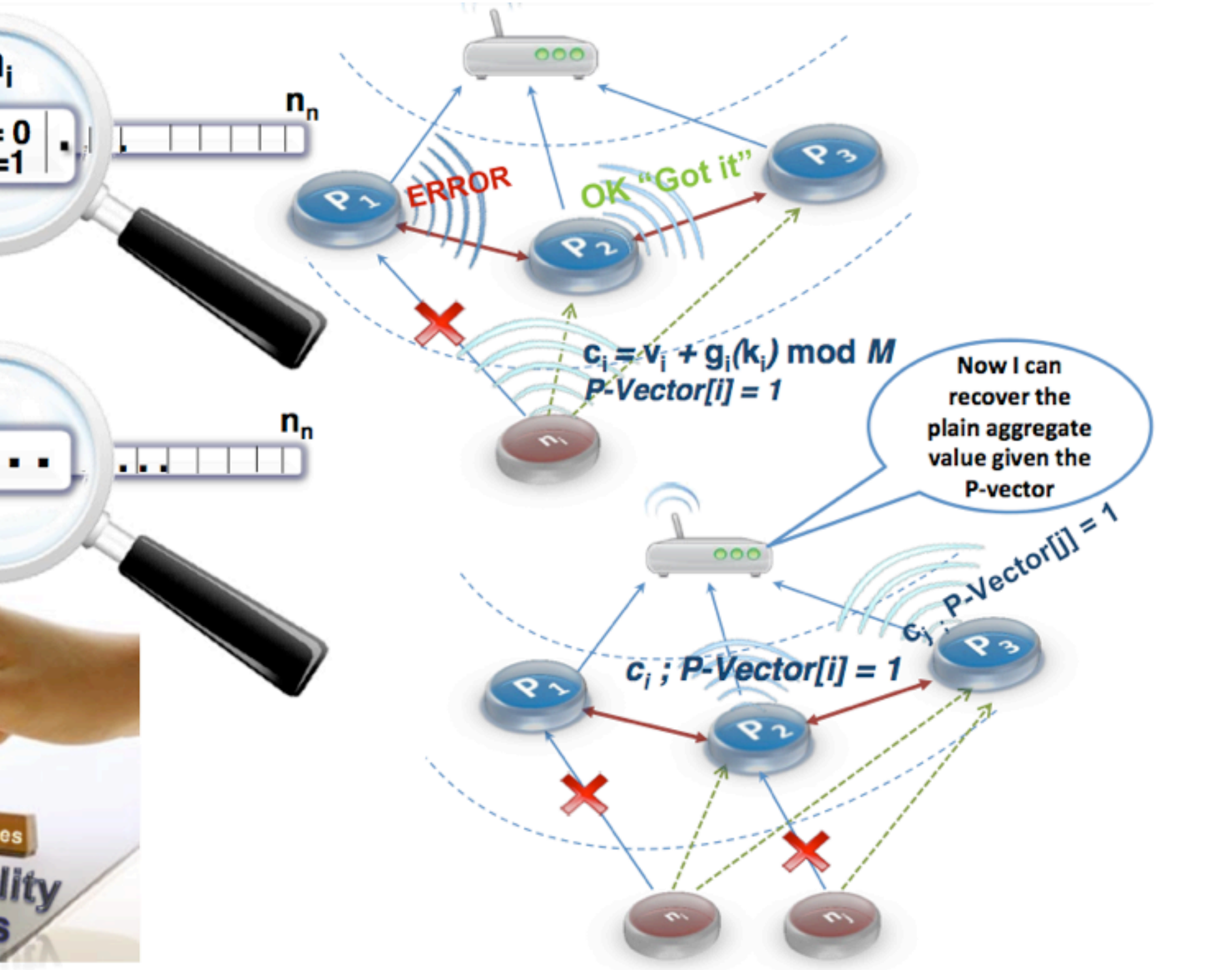
correctly aggregate, but eavesdrop

QUIET INFILTRATORS

stealthily infiltrate the network to eavesdrop

Protocol (rough outline)





Protocol (rough outline)

1. Each sensor n_i encrypts its value v_i as $c_i = v_i + g_i(k_i) \bmod M$, and sets its corresponding bit in the P-Vector

2. The resulting c_i values are aggregated using the Cascaded RideSharing protocol, which results in the sink receiving the value $C = \Sigma_i c_i \bmod M$

3. The sink computes the aggregate key value $K=\Sigma_i g_i(k_i) \bmod M$ for each i in P-vector

4. The sink extracts the final aggregate value $V = \Sigma_i v_i = C - K \bmod M$

And it started here...

Algorithm 1: Aggregation and routing algorithm run by sensors within the network

input : PC, BC, SP, v
$A := 0$;
$P := \bar{0}$;
$L.r := \bar{0}$;
$L.e := \bar{0}$;
if v *NOT NULL* **then** `// Aggregate own value`
  $A := A + v + g_{ID}(k_{ID}) \bmod M$;
  P[ID] := 1;
end
$L :=$ rcvL(SP);
foreach *Child* C *in* $PC \cup BC$ **do**
  if *rcv*(A_c,P_c) *from Child* C **then**
    if $C \in PC$ *OR* ($C \in BC$ *AND* $L[C].e = 1$ *AND* $L[C].r = 0$) **then** `// Aggregate the received values`
      $A := A + A_C \bmod M$;
      $P := P$ OR P_c;
      $L[C].e := 1$;
    end
    else `// Propagate the error signal`
      $L[C].e := 1$;
    end
  end
end
Transmit(A,P,L);

Algorithm 2: Final aggregation and decryption algorithm used by the data sink

input : PC
output: $FinalA$
$A := 0$;
$P := \bar{0}$;
$K := 0$;
$FinalA := 0$;
foreach *Child* C *in* PC **do**
  if *rcv*(A_c,P_c) *from Child* C **then**
    $A := A + A_C \bmod M$;
    $P := P$ OR P_c;
  end
end
foreach *bit set to '1' in* P **do**
  $K := K + g_i(k_i) \bmod M$;
end
$FinalA := A - K \bmod M$;

With the properties:

Theorem 1 (Confidentiality): During the execution of the protocol described by Algorithms 1 and 2, no sensor (except the sink) can learn the value of the readings reported by any other sensor, nor the value of any intermediate aggregate value.

Theorem 1 follows directly from the semantic security of the cipher used by Algorithms 1 and 2 and the fact that each sensor node shares a unique key with the sink.

Theorem 2 (Correctness): Under the assumption of "honest but curious" or "quiet infiltrators" attack nodes, the protocol described by Algorithms 1 and 2 includes each sensor reading at most one time during the aggregation process. Further, the sink node is able to correctly identify the sensors that contributed to this aggregate, generate the resulting aggregate key, and recover the correct result.

What does a proof look like?

Formalize (abstract, if needed) program/protocol [transition system]

Formalize property [Temporal Logic]

Invent (divine?) auxiliary constructs  ...

Prove:

1. ...
2. ...

⋮

k. ...

] logical formulae over system description and  ...

Conclude: Property

Safety

- System often associated with “good” and “bad” states
- Safety: the system is always in a good state
- Invariance: safety properties that can be described by “state assertions”

- every safety property can be reduced to an invariance property
- Invariance properties are perhaps the most important properties one may wish to prove on systems!
- They capture properties like “within a given amount of time, something good must occur”, “there is no security violation”, &c

How to prove Invariance

- Inductive invariant: true at initial state, and preserved in every step
- **INV** rule: construct an auxiliary invariant, show that it's inductive and implies "good"

- (Why do we need the auxiliary invariant? because inductiveness may be hard to show on real system)
- (Auxiliary inductive invariants can be viewed as an abstraction of the reachable states)

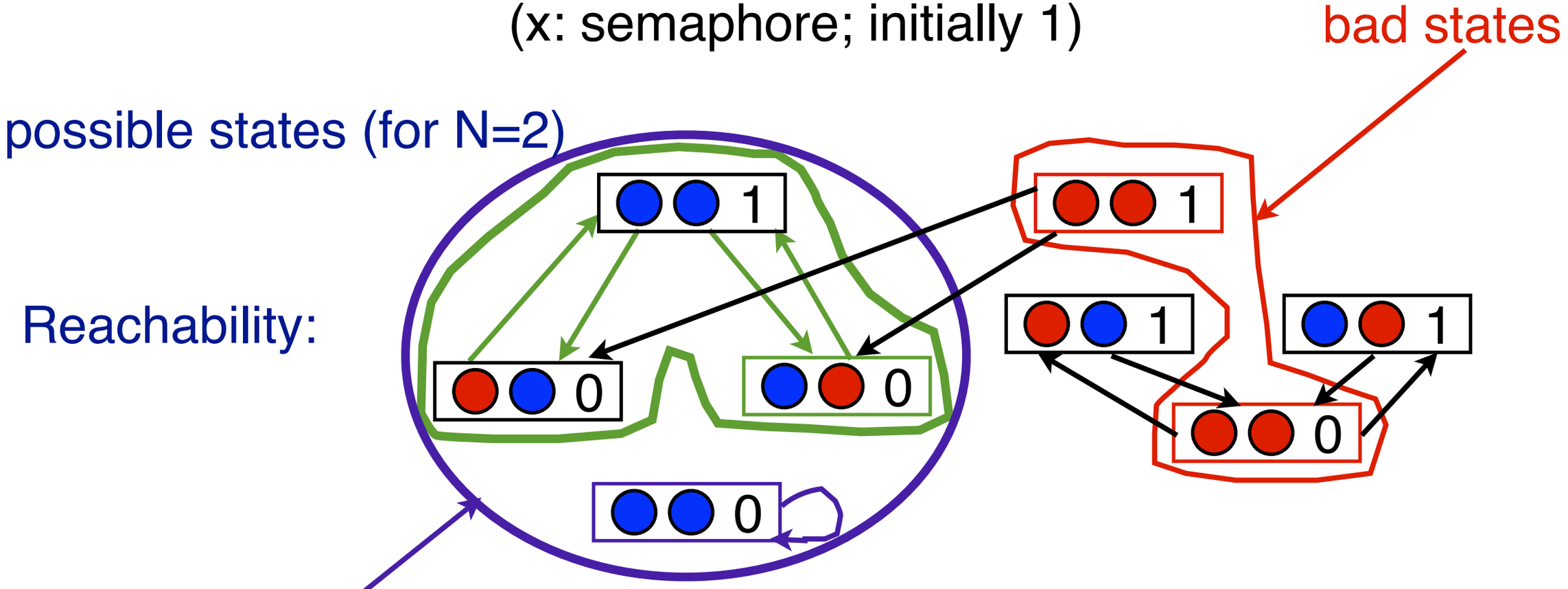
Example

P[i]:

req(x)

rel(x)

(x: semaphore; initially 1)



Reachability:

states covered by auxiliary invariant

auxiliary invariant:

$$\sum_{i=1}^{N} (P[i] \text{ is in red state}) + x \;\leq\; 1$$

Back to RideShare

Fault tolerance, power/timing, and correct aggregation of the P-vectors are all safety properties

Sample property: If

1. attackers can only eavesdrop
2. for sufficiently long time (so that information can propagate from leaves to sink) there are no changes in topology (but possibly for a single link failure)

   Then within a given time bound the sink node receives the “correct” information

Parameterized Systems*

- A parallel composition of N (finite-state) processes where N is unknown

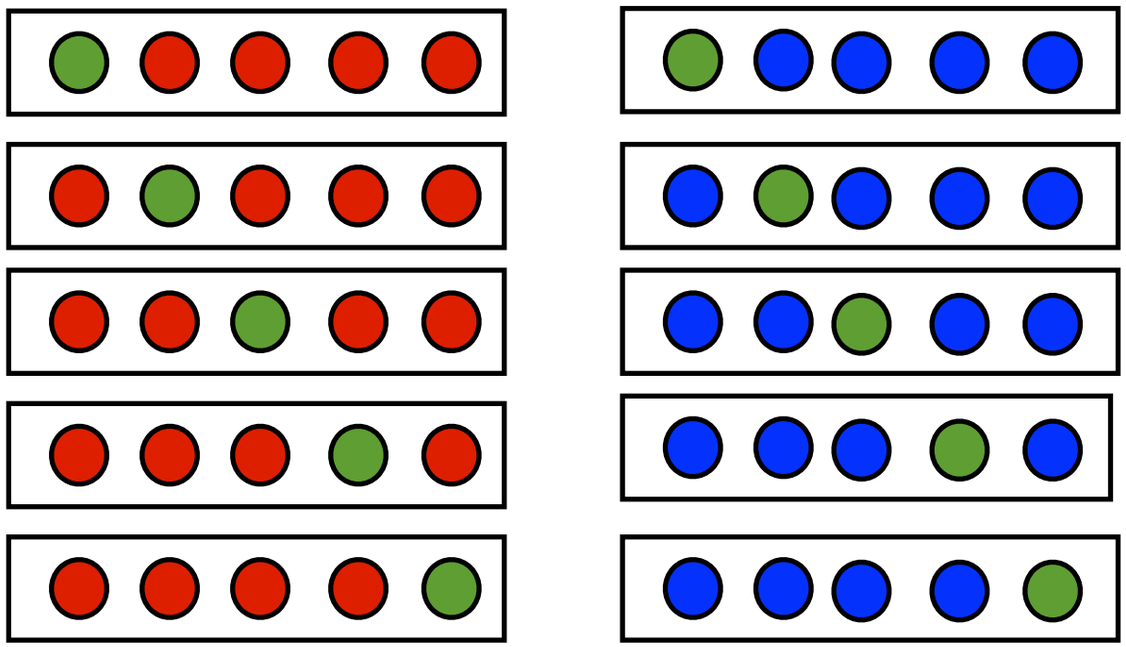
$$P_1 \,||\, P_2 \,||\, P_3 \,||\, \cdots \,||\, P_N$$

- Proofs requires auxiliary constructs parameterized on N
  - For safety, an *inductive invariant*
- Invisible Invariants: derive constructs for general N by abstracting from the mechanical proof of a particular N
  - under-approximation can yield over-approximation
  - Proofs can be done entirely using finite-state model checking, w/o explicitly generating the auxiliary constructs

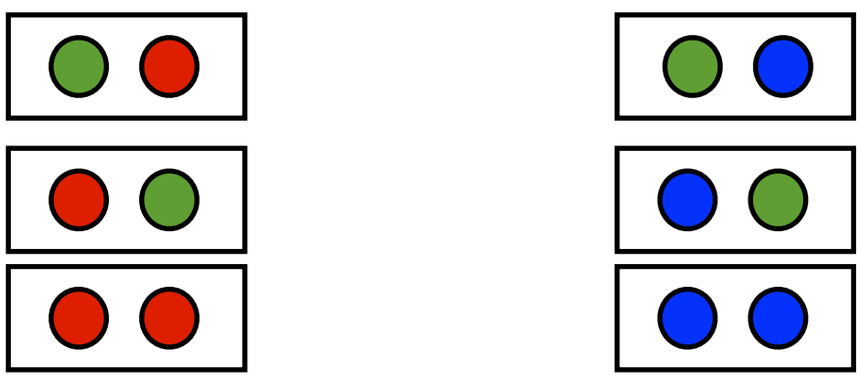
*Joint work with Amir Pnueli and students (main ideas in [ZP04])

Generating an Invariant

1. Compute the reachable states R_N for a fixed N (say N=5)



2. Project onto a small subset of processes (say {1,2})



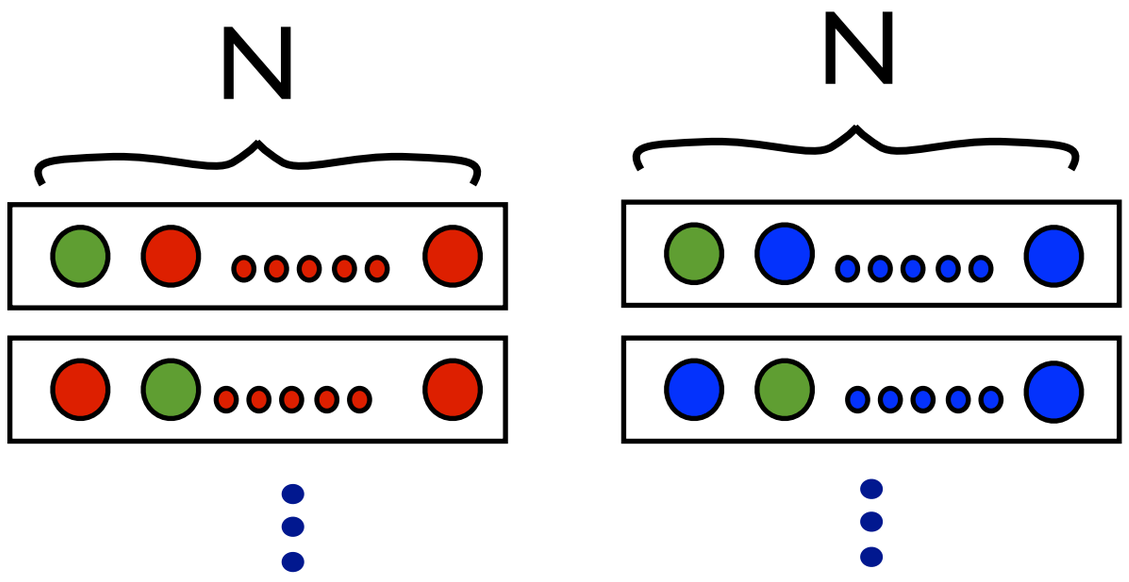
$$\pi = \{(s_1, s_2) \mid (s_1, s_2, \ldots, s_N) \in R_N\}$$

Generating an Invariant

2. Project onto a small subset of processes (say {1,2})

$$\pi = \{(s_1, s_2) \mid (s_1, s_2, \ldots, s_N) \in R_N\}$$

3. Generalize from two to N, to get G_N



$$G_N \;=\; \forall_{i \neq j \in [1..N]}\; \pi(s_i, s_j)$$

4. Test whether G_N is an invariant for all N

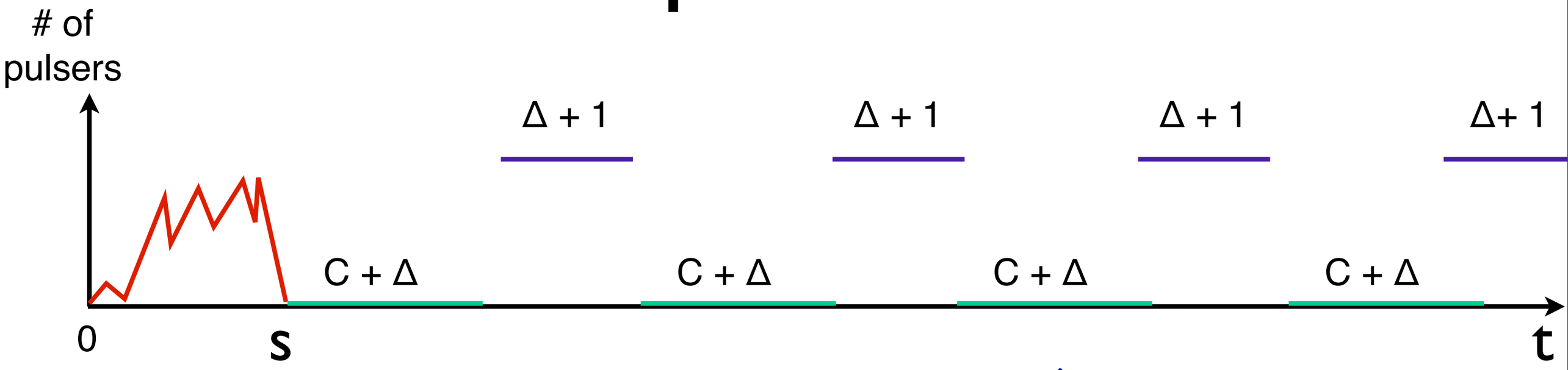
Checking Inductiveness

- Small Model Theorem:
  - If there is a counterexample with N>M there is a counter-model with N≤M
  - Suffices to check inductiveness for N≤M

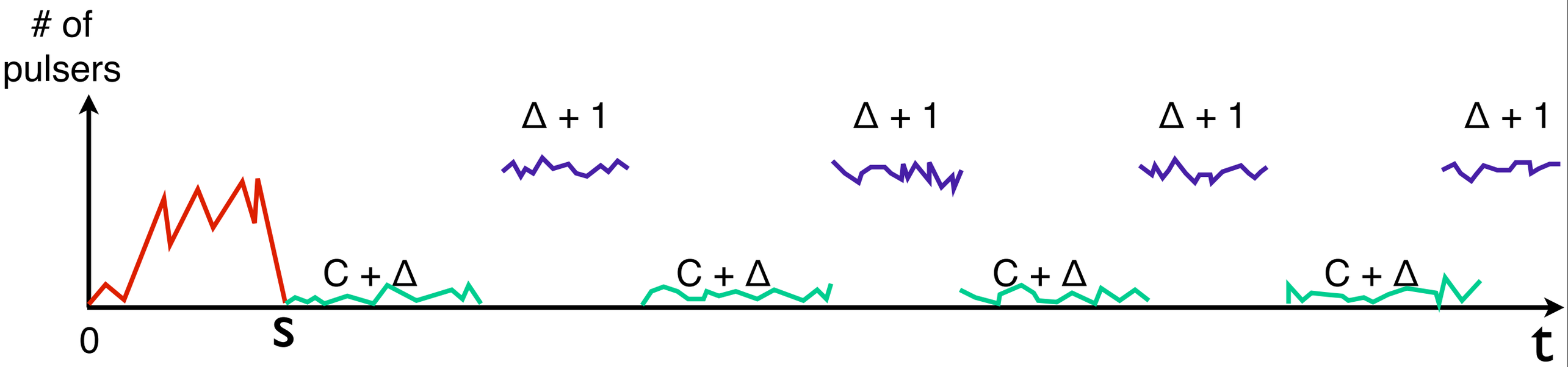
Thus, both invariant generation and invariant checking amount to finite-state model checking

However, doesn't apply to all cases (and Abstract Interpretation does!)

Example: Pulser



if there are no "bad guys"



with (not too many) bad guys

Pulser: Protocol and specs

Protocol:

$$P[i] ::=$$

$$\left[\begin{array}{l} \textbf{If } count[i] > C \ \vee \ BA[Clock] \\ \quad \textbf{then } count[i] := C \\ \quad \textbf{elsif } count[i] > 0 \\ \quad\quad \textbf{then } count[i] = \min(C, count[i] - 1) \end{array}\right]$$

$$G ::=$$

$$\left[\begin{array}{l} Clock := Clock + 1 \bmod \Delta \\ \textbf{If } \forall i.\ \neg fault[i] \rightarrow count[i] \leq 0 \\ \quad \textbf{then } BA[Clock] := \mathrm{T} \\ \textbf{If } \forall i.\ \neg fault[i] \rightarrow count[i] > 0 \\ \quad \textbf{then } BA[Clock] := \mathrm{F} \\ \quad \textbf{else } BA[Clock] := \{\mathrm{T}, \mathrm{F}\} \end{array}\right]$$

Property:

$$\Diamond\,\Box\,((\neg pulse \wedge \bigcirc\, pulse) \rightarrow \bigwedge_{t=1}^{\Delta+1} \bigcirc^{t} pulse \ \wedge \ \bigcirc^{\Delta+2} \neg pulse \quad \wedge$$

$$(pulse \wedge \bigcirc\, \neg pulse) \rightarrow \bigwedge_{t=1}^{C+\Delta} \bigcirc^{t} \neg pulse \ \wedge \ \bigcirc^{C+\Delta+1} pulse)$$

But, it didn't start there...

Algorithm **Large-Cycle-Pulser** ** executed repeatedly at each beat **

1. for each $i \in \{1, .., \Delta\}$ do
   execute the i^{th} round of the $\mathcal{BBB}_i$ protocol;
2. (a) if $Counter > 0$ then
   $Counter := \min\{Counter - 1, Cycle'\}$;
   $WantToPulse := 0$;
   (b) else
   $WantToPulse := 1$;
3. if $\mathcal{V}(\mathcal{BBB}_\Delta) = 1$ then
   (a) do PULSE;
   (b) $Counter := Cycle'$;
4. for each $i \in \{2, ..., \Delta\}$ do
   $\mathcal{BBB}_i := \mathcal{BBB}_{i-1}$;
5. initialize a new instance of $\mathcal{BBB}$, $\mathcal{BBB}_1 = \mathcal{BBB}(WantToPulse)$.

Theorem 1. *The Large-Cycle-Pulser algorithm is a* $[\Delta, \Delta + Cycle']$-PULSER.

How to Verify

- Verification of Protocols (arbitrary, even dynamic, topology and number of participants) even in case of attacks

Parameterized Verification

- Verification of stepwise refinement (of functional and non-functional properties)

Translation Validation

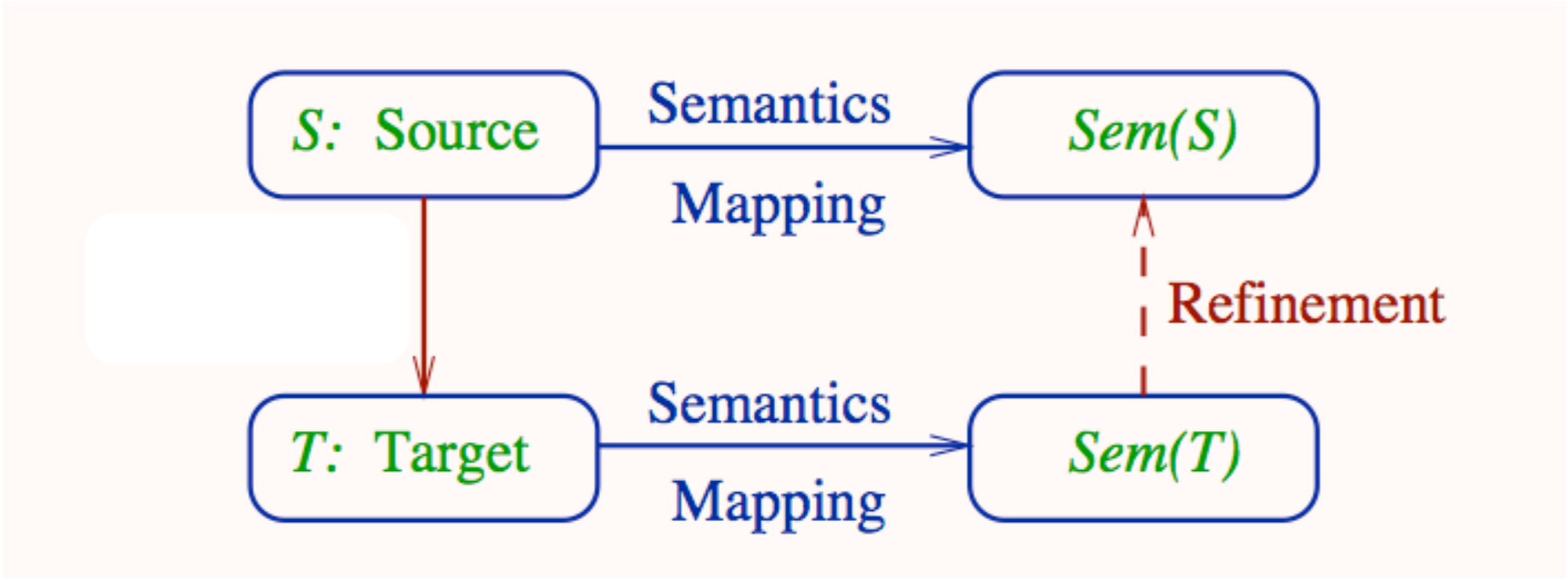
Translation Validation

- Originally proposed as part of the Safety Critical Embedded Systems project (1995-1998), where all validation/verification occurred at high levels and correct design automatically “translated” into C
- Had to verify the translation!
- Evolved to:
  - TV of optimizing compilers for Intel’s ORC compiler
  - MicroFormal @ Intel: tool to verify backward compatibility of microcode

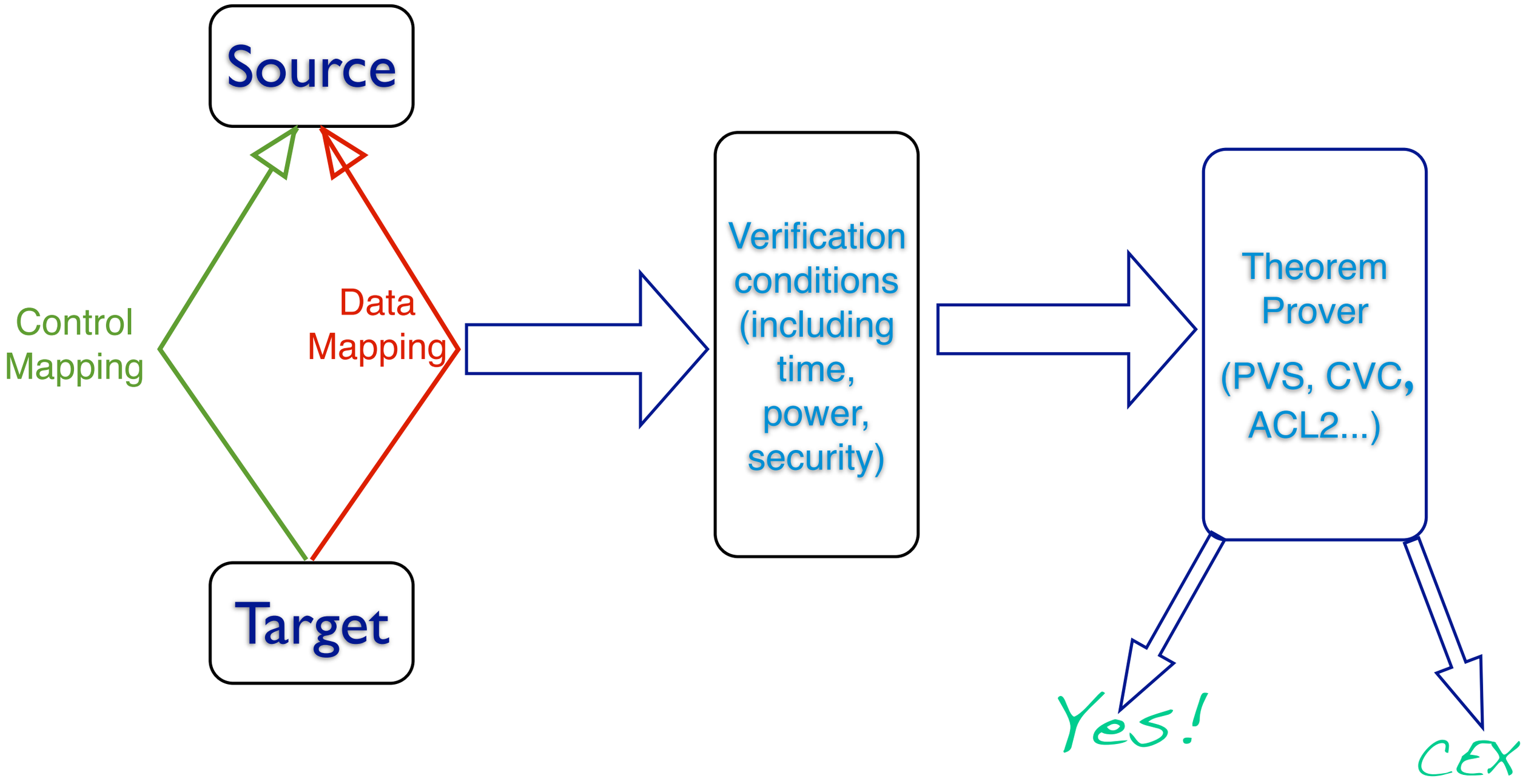
Translation Validation

- Verification of translatOR is often infeasible
  - "translator" (compiler) may be proprietary, compiler may evolve over time (so its verification may become useless), provides an independent cross check
  - "translator" may be human!
- Instead of verifying translator, verify each translation!!
  - has constant run-time additional cost (justifiable)
  - usually doable because source and target are similar (in case of optimizing compilers, there are a known set of optimizations)

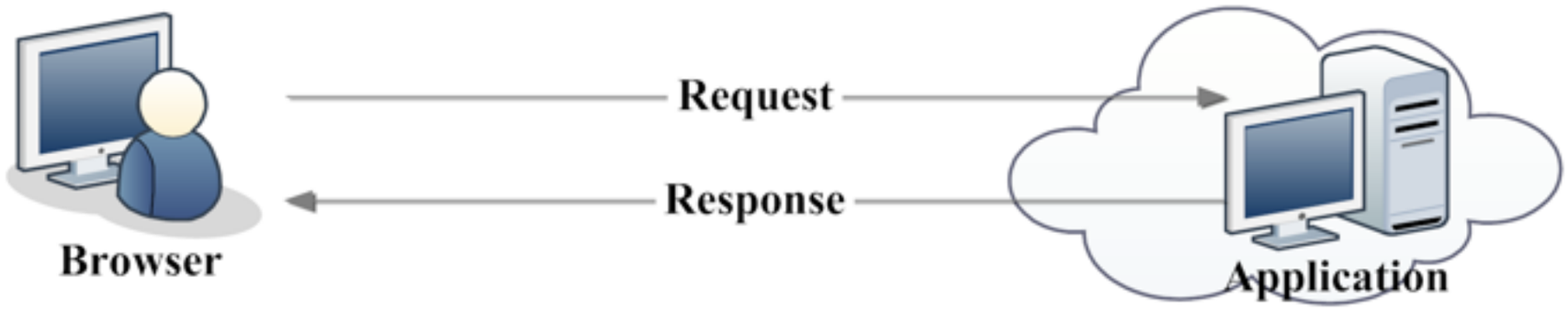
T.V. : Idea (1)


S: Source
Semantics
Mapping
Sem(S)
Refinement
T: Target
Semantics
Mapping
Sem(T)

T.V. : Idea (2)


Source
Control Mapping
Data Mapping
Target
Verification conditions (including time, power, security)
Theorem Prover
(PVS, CVC, ACL2...)
Yes!
CEX

Example: Client/Server

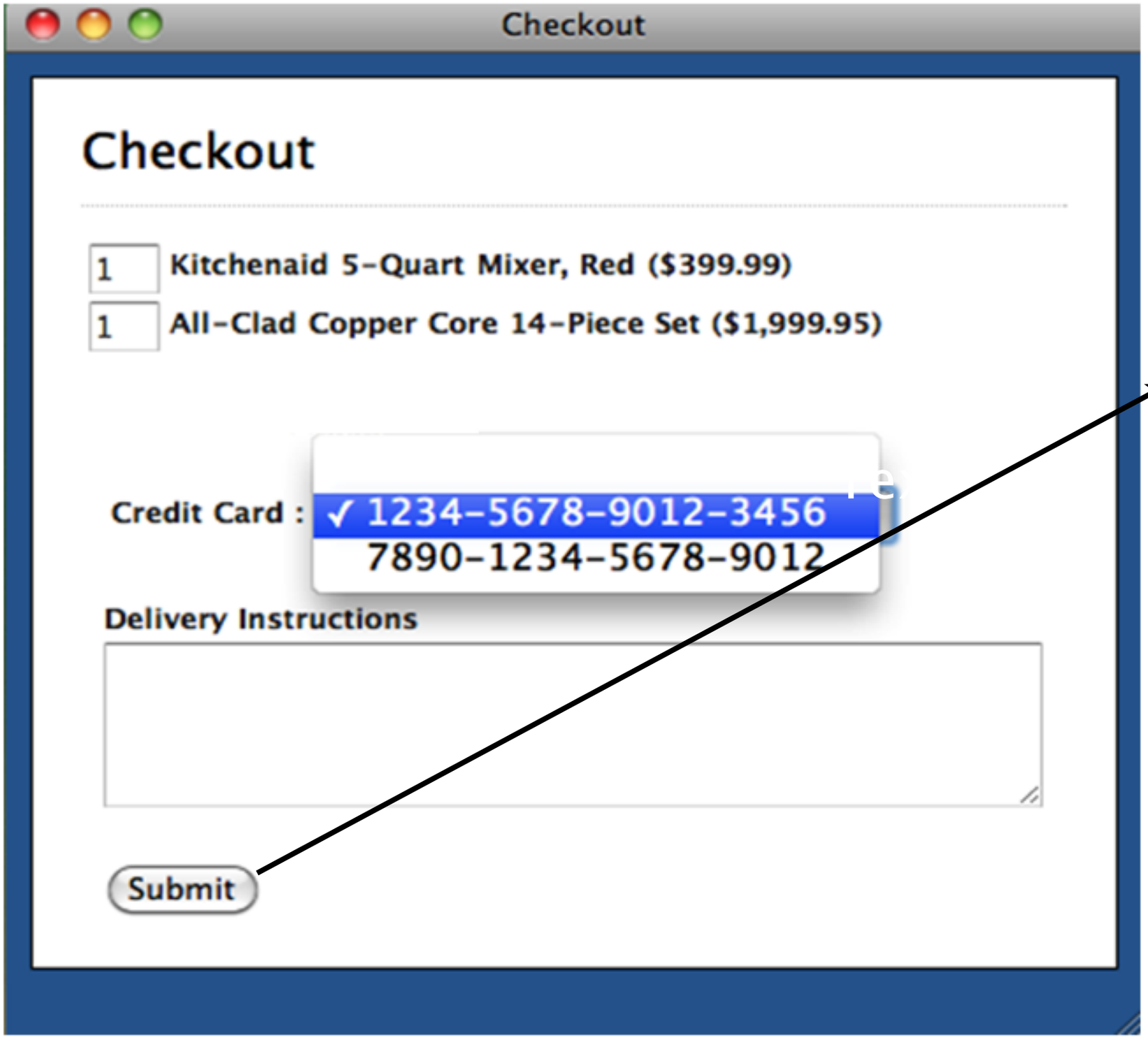


- Web applications need to
- Validate user supplied input
- Reject invalid input

- Examples:
  - "date/month combination is invalid"
  - "Credit card number is exactly 16 digits"
  - "Expiration date of Jan 2009 is not valid"

- Validation traditionally done at server: round trip, load
- Popular trend: Browser (client) validation through JavaScript

Client Side Validation using JavaScript



onSubmit=
validateCard();

Validation Pass?

Yes

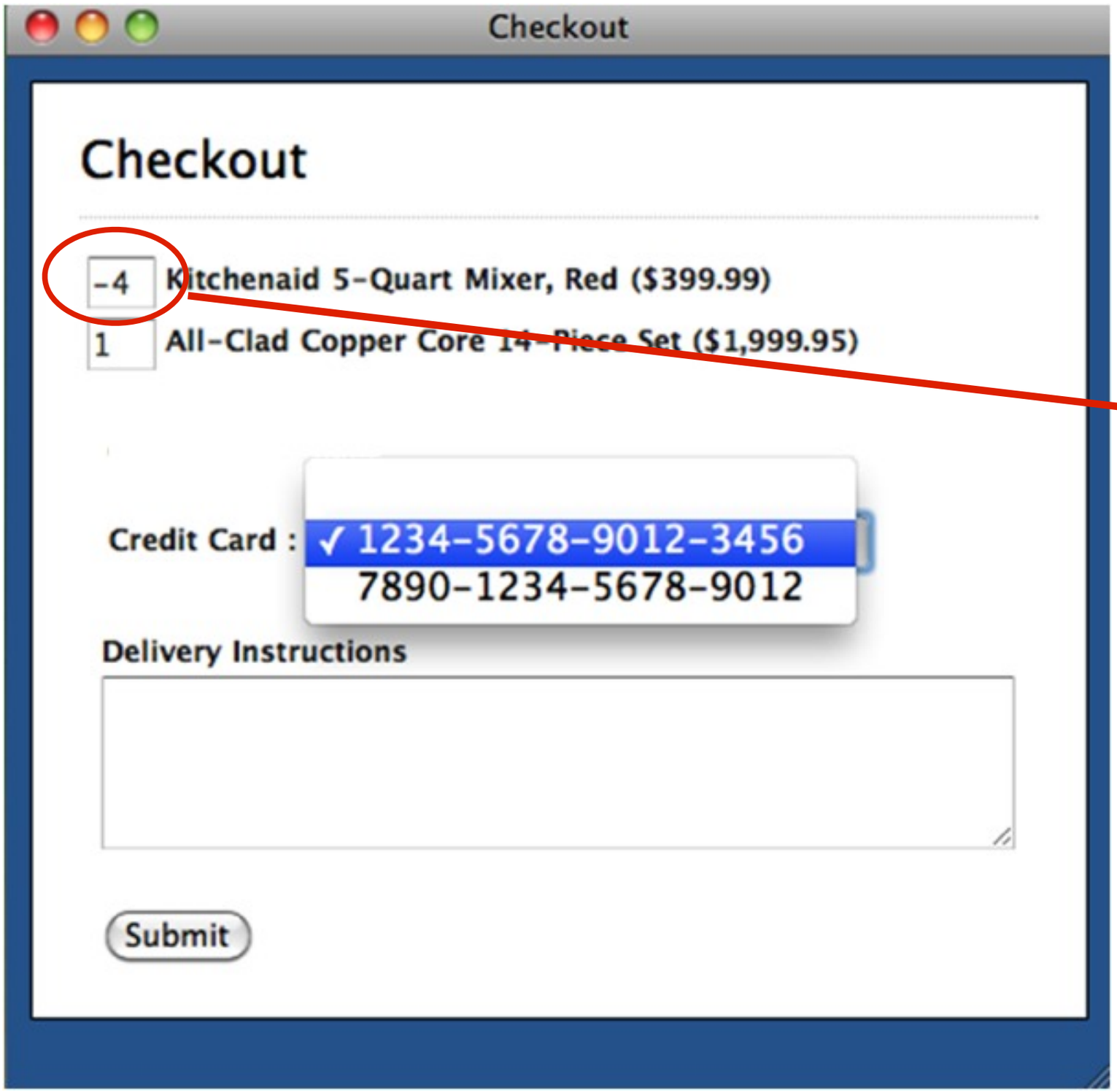
No

Send inputs to Server

Reject inputs

Problem: The user interacting with website can attack the client !

Problem: Client is Untrusted Environment



- Validation can be bypassed

- Previously rejected values, sent to server
  - Invalid quantity: -4

- Ideally: Re-validate at server-side and reject
  - If not, security risks

Applying TV on Example

Client

cost :=0
ok := true
For every i in list
    cost := cost + quan[i] x cost[i]
    if quan[i] < 0
    then ok := false
If cc not in list
then ok := false

Server

Cost := cost
Ok := true
If !valid(cc)
then Ok := false

Need to verify: $\text{Cost} = \text{cost} \ \wedge \ \text{Ok} = \text{ok}$

Counter Examples:

$$\text{quan}[1] < 0 \ \wedge \ \neg\text{ok} \ \wedge \ \text{Ok}$$
$$(\text{cc not on list}) \ \wedge \ \neg\text{ok} \ \wedge \ \text{Ok}$$
$$\neg\text{valid}(\text{cc}) \ \wedge \ \text{ok} \ \wedge \ \neg\text{Ok}$$

How to Overcome?

- Use Translation Validation to show that every check performed by the client is also performed by the server
- (Of course) needs to be done automatically
  - The code of client and server is usually in different languages – we successfully dealt with a similar issue in MicroFormal by translation into an intermediate representation language (IRL)
- Can then automatically (or almost fully-automatically) patch server and repeat TV

Conclusion

- Security NEEDS formal methods
  - The stakes are high: security protocols have bugs
  - Many (some recently developed) FM techniques are not incorporated in security verification:
    - Verification of network protocols
    - Verification of refinement
    - Integration of proofs techniques
- Attacker needs to be formally specified
- We have many techniques in FM that can assist
- Much research is needed to obtain reliable security

What's Next?

- Develop methodologies, supported by tools, to formally verify secure network protocols
- DARPA project:
  - formally verify the secure RideSharing protocol
  - apply to expanded network topologies (using D4V)
- Automatic patching of servers (or cloud)
- Data sanitization (for testing purposes, e.g.)
- Verification of smartgrid controllers