Code Generation for network software with formal safety guarantees

Riccardo Sisto

Dipartimento di Automatica e Informatica

Politecnico di Torino

Outline

- Introduction and motivation
  - Safety and Formal Methods in Network Software
  - Code Generation and formal methods
- Application to packet processing
  - The goals (what safety means here) and the ideas
  - The FSA approach for packet filters
  - Extending the approach to other packet processing applications
- Application to cryprographic protocols
  - The goals
  - The spi2java approach

Motivation

- Safety and security requirements in network software are more and more compelling
  - Increased pervasiveness
  - Increased demand for trustworthyness
  - Increased threats
  - …
- Formal methods can help to increase confidence bad things will not happen in critical applications
  - Introduce mathematical rigour in software development

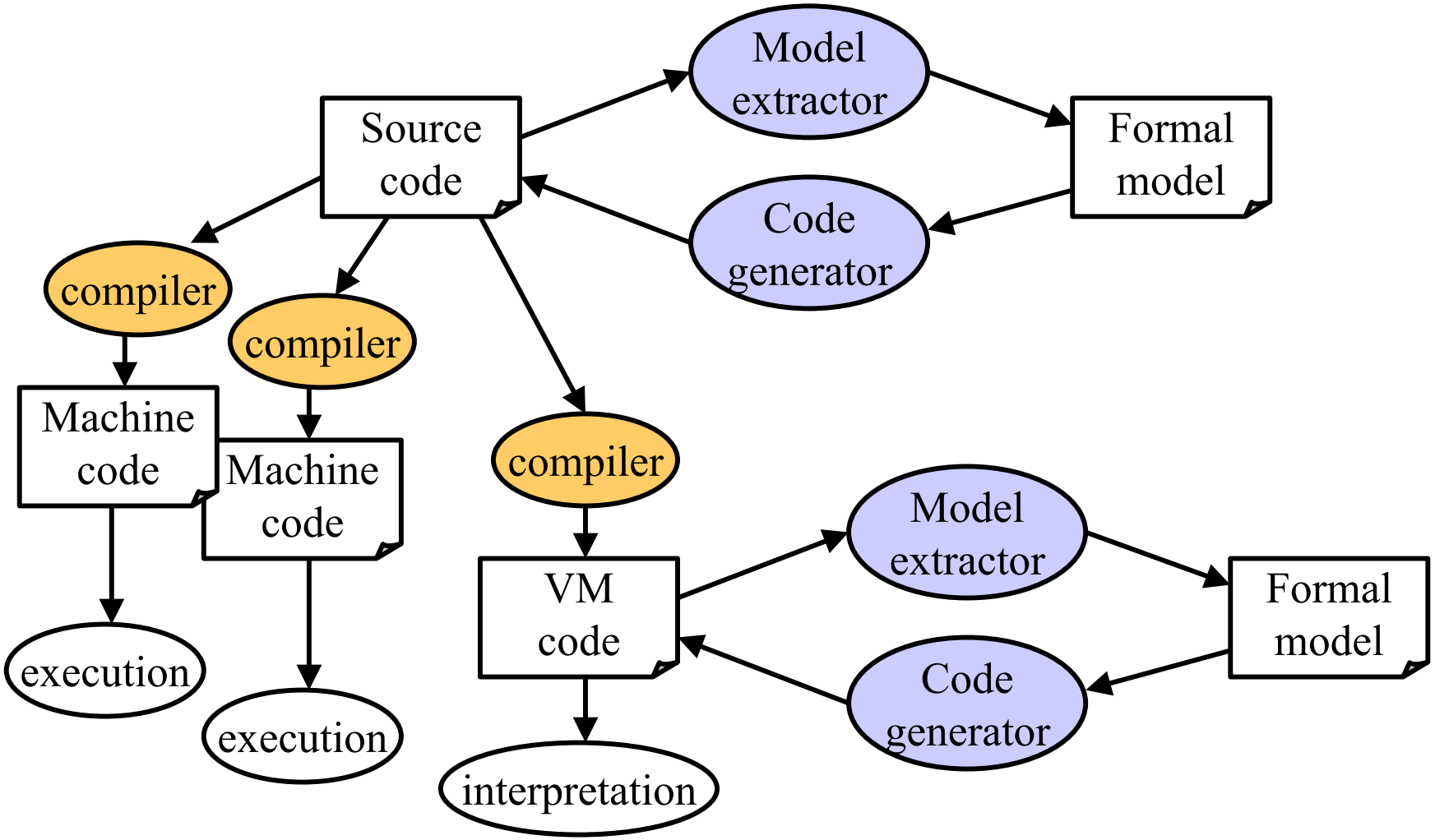
Yes, but can Formal Methods be made acceptable/affordable?

- Key issues for success in practice:
  - Hide mathematical complexity
  - Full automation
- Examples:
  - Java byte code verifier
  - …

How to get affordable FM

- At first sight, success seems impossible
  - Even simple questions (e.g. termination) on Turing complete machines are undecidable
  - Formal specification of properties to be checked may be difficult
- Possible ways out?
  - Approximate (incomplete) models or analysis
    - False poitives / false negatives
  - Exploit domain specific features
  - Use pre-defined properties

The code-model gap

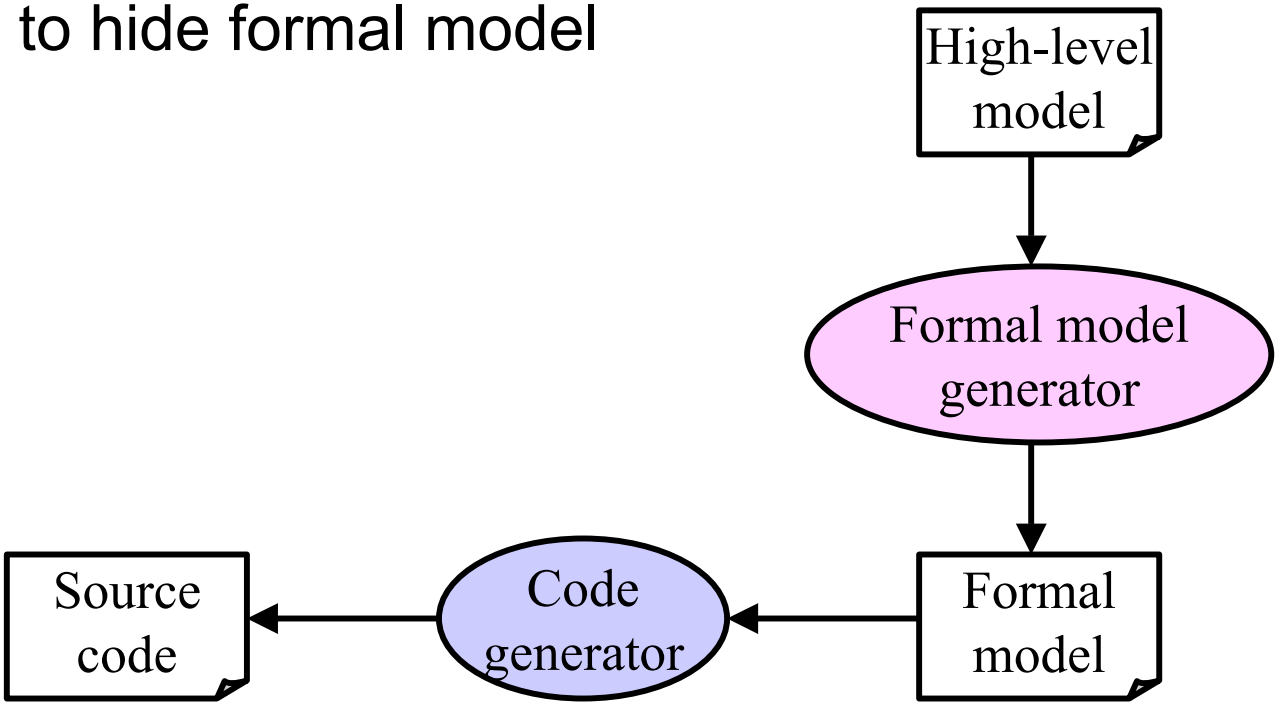


Code Generation

- The way SW is written can be constrained
  - Some “low-level” properties guaranteed by construction
  - or automatically checkable
- Lack of flexibility and performance may be an issue
- The formal model is exposed to the programmer
  - Programmers like programming languages much more than mathematical models

Two-stage Code Generation

- A way to hide formal model



Packet Processing

- Contrasting requirements
  - Must go fast
    - Follow wire speed
  - Must be safe
    - No unpredicted run-time exceptions like out-of-bound memory accesses
    - No infinite loops
- Can be reconciled by using FM and code generation
  - Generate code that is safe by construction or easy to formally check statically

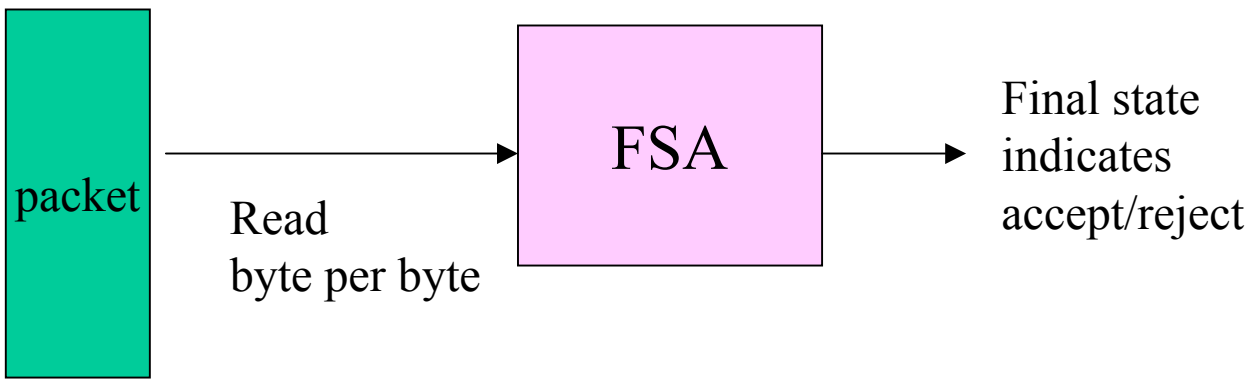
    => Avoid the overhead of run-time checks

Packet Filters

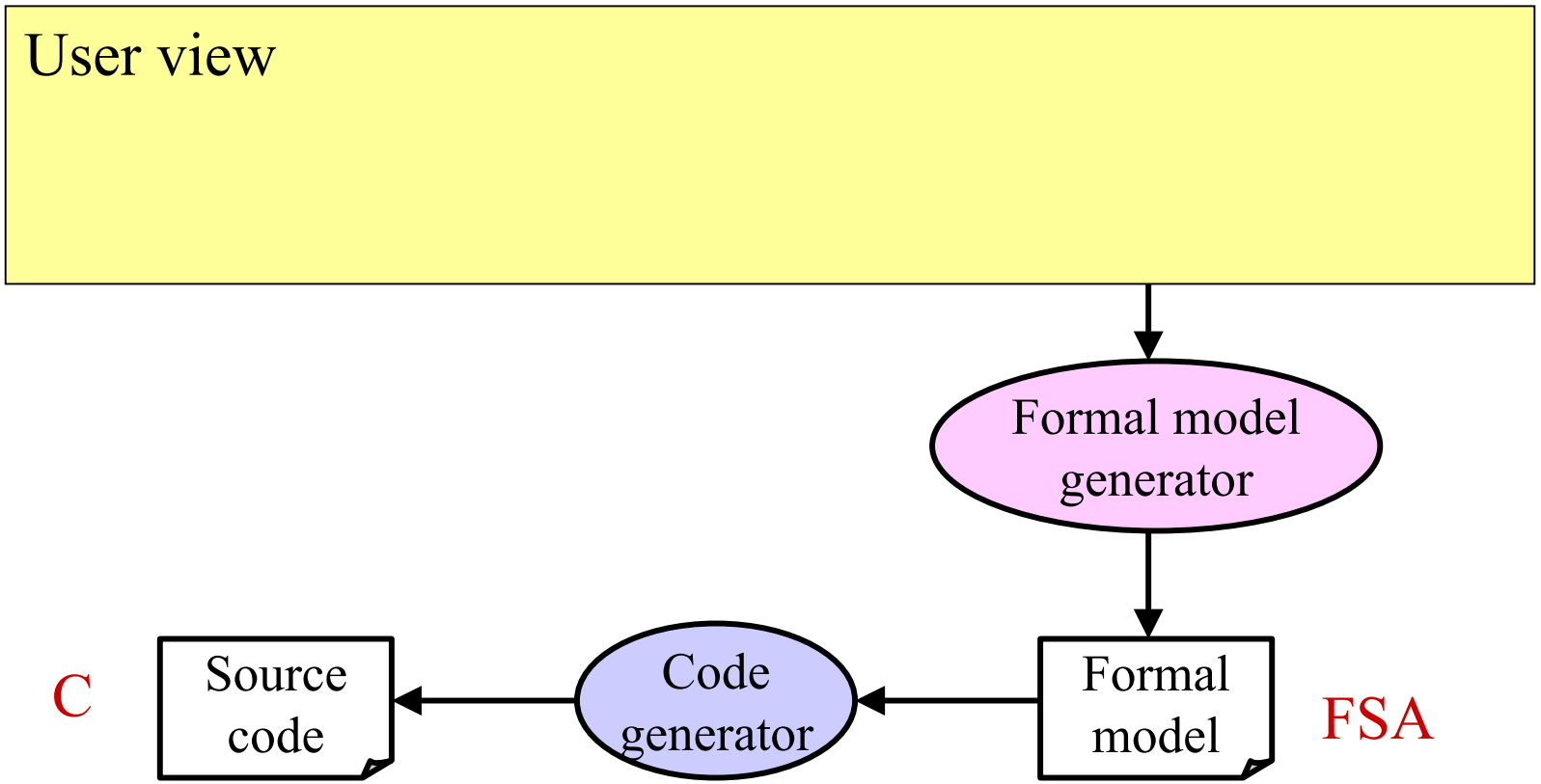
- Our starting case study:
  - Well defined application domain
    - Input: packet (memory buffer) Output: accept, reject
  - code generation already in use
    - BPF, BPF+, NetVM, Pathfinder, DPF,…
- Previous solutions have limitations
  - E.g. termination ensured by disallowing backward jumps
- Aim:
  - Fast and flexible packet filtering
  - Formal guarantee about termination and no out-of-bound memory access

The FSA Packet Filter Idea

- Packet filtering is a language recognition problem
- Language is regular because packets have finite size (true for stateless filtering)
- Formal model: Finite State Automaton (FSA)



…and its implementation



FSA Generation

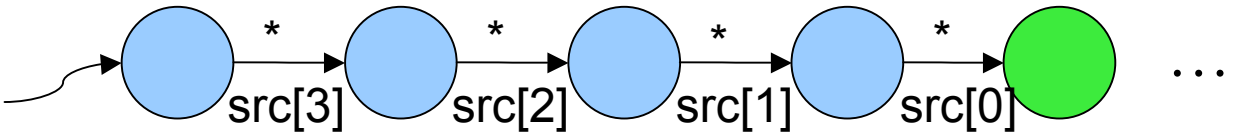
- Key issue
  - Algorithms for FSA combination by standard Boolean and Kleene operators are well known
- Procedure
  1. Generate FSA templates for protocols from NetPDL DB (required only on changes in protocol DB)
  2. Generate FSAs for single predicates in filter description
  3. Combine FSAs according to filter description and generate final (minimized) DFA for filter

Example

NetPDL

```
...
<field type="fixed" name="src" size="4" ... />
...
```

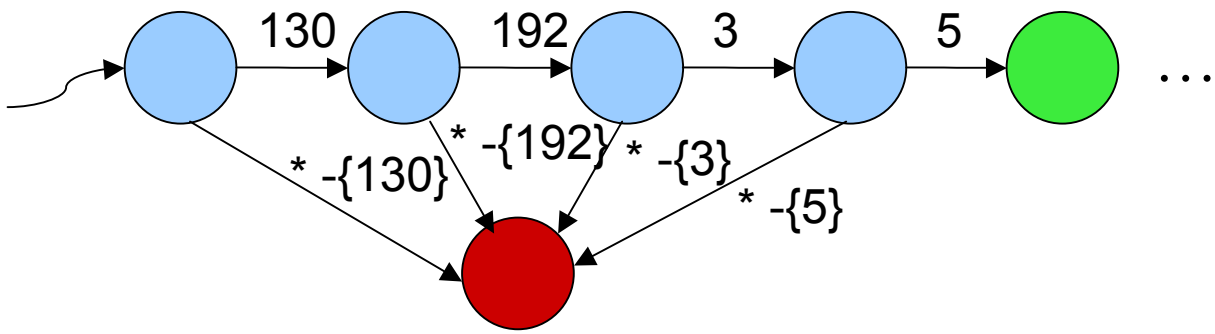
FSA template



FRL predicate

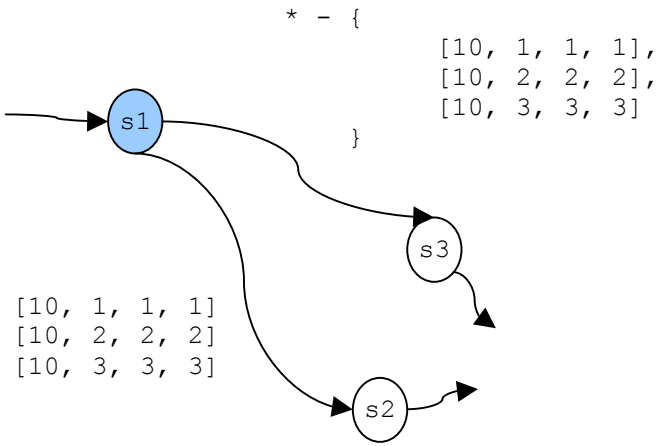
```
...
(ip.src=130.192.3.5)...
...
```

predicate FSA



C Code Generation

- FSA implementation is simple
  - Read input, switch and jump to next state
  - No run-time issues like arithmetic exceptions



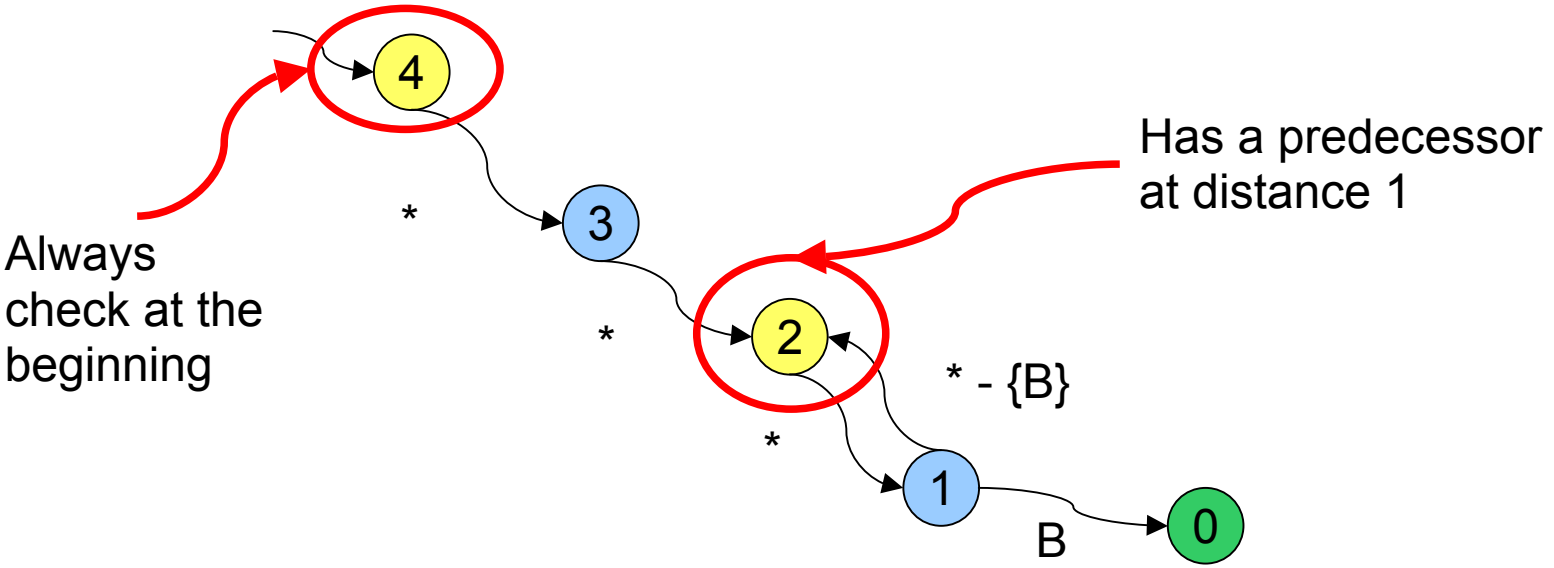

```
s1:
    read4 = get_4_bytes(packet, len, offset);
    offset += 4;
    switch(read4) {
          case ((10 << 8*0) | (1 << 8*1) | (1 << 8*2) | (1 << 8*3)):
          case ((10 << 8*0) | (2 << 8*1) | (2 << 8*2) | (2 << 8*3)):
          case ((10 << 8*0) | (3 << 8*1) | (3 << 8*2) | (3 << 8*3)):
                    goto s2;
                    break;
          default:
                    goto s3;
                    break;
    }
```

Safety

- Termination
  - Simply enforced because **each** step consumes input => eventually (after finite number of steps) input terminates
- Memory access safety
  - Minimization of required run-time checks based on distance-to-success (in most cases 1 check is enough)

Memory Check Minimization

- Memory checks needed only when in state with distance from success equal or greater than (at least) one of its predecessors

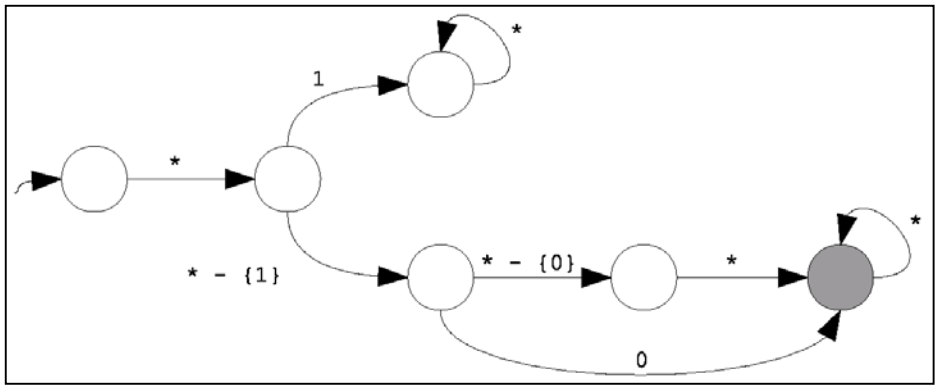


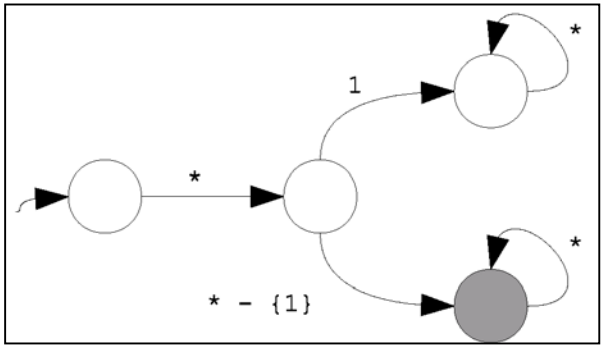
Optimizations I

- Succeed early

  terminate filter as soon as the user-specified rules are matched

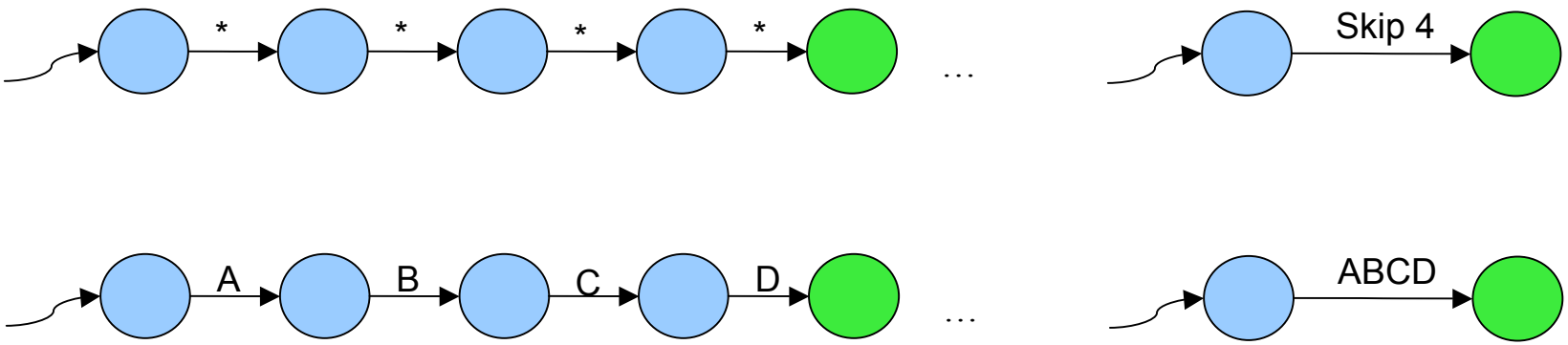
  => mark as final any state that is post-dominated by a final state





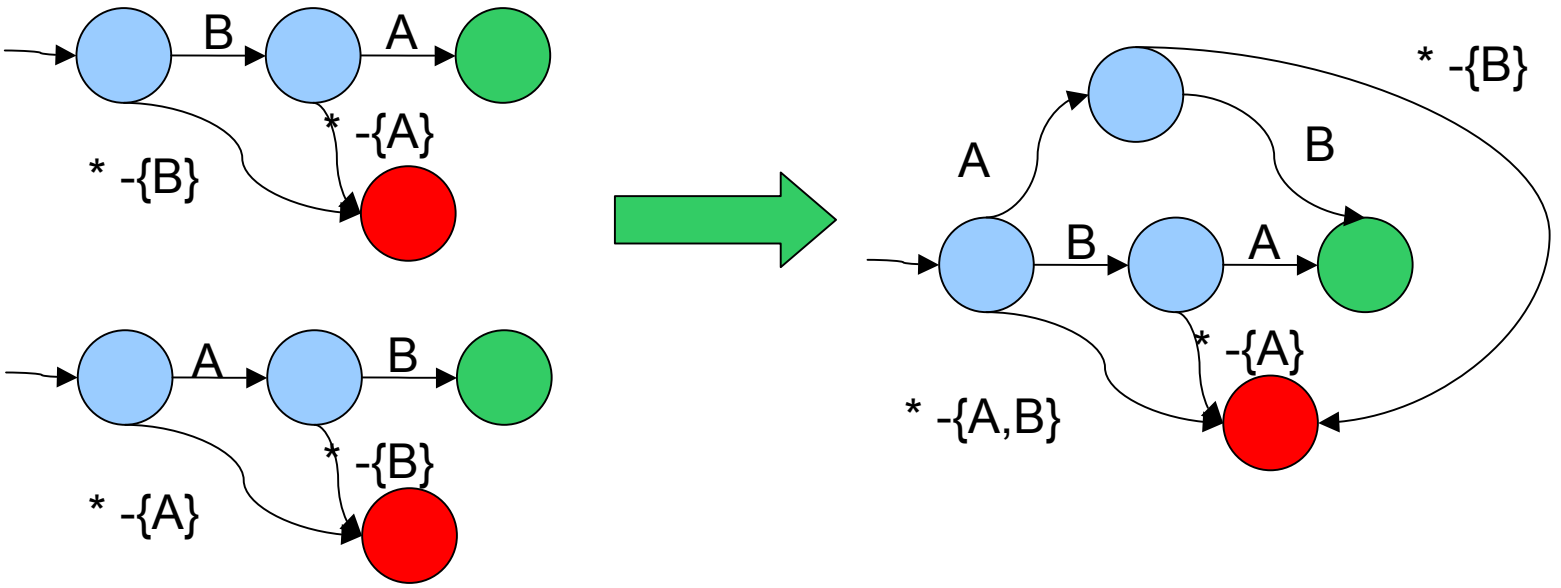
Optimizations II

- Transition compaction
  - Sequences of * transitions compacted into a single * transition
  - Sequences of any transitions compacted into n-byte transitions according to machine word size



Optimizations III

- Ad-hoc optimizations introduced by other filter generators (e.g. BPF+) are automatically exploited in FSA-based filters
- Example: (ip.src = A && ip.dst = B) || (ip.src = B && ip.dst = A)



Performance

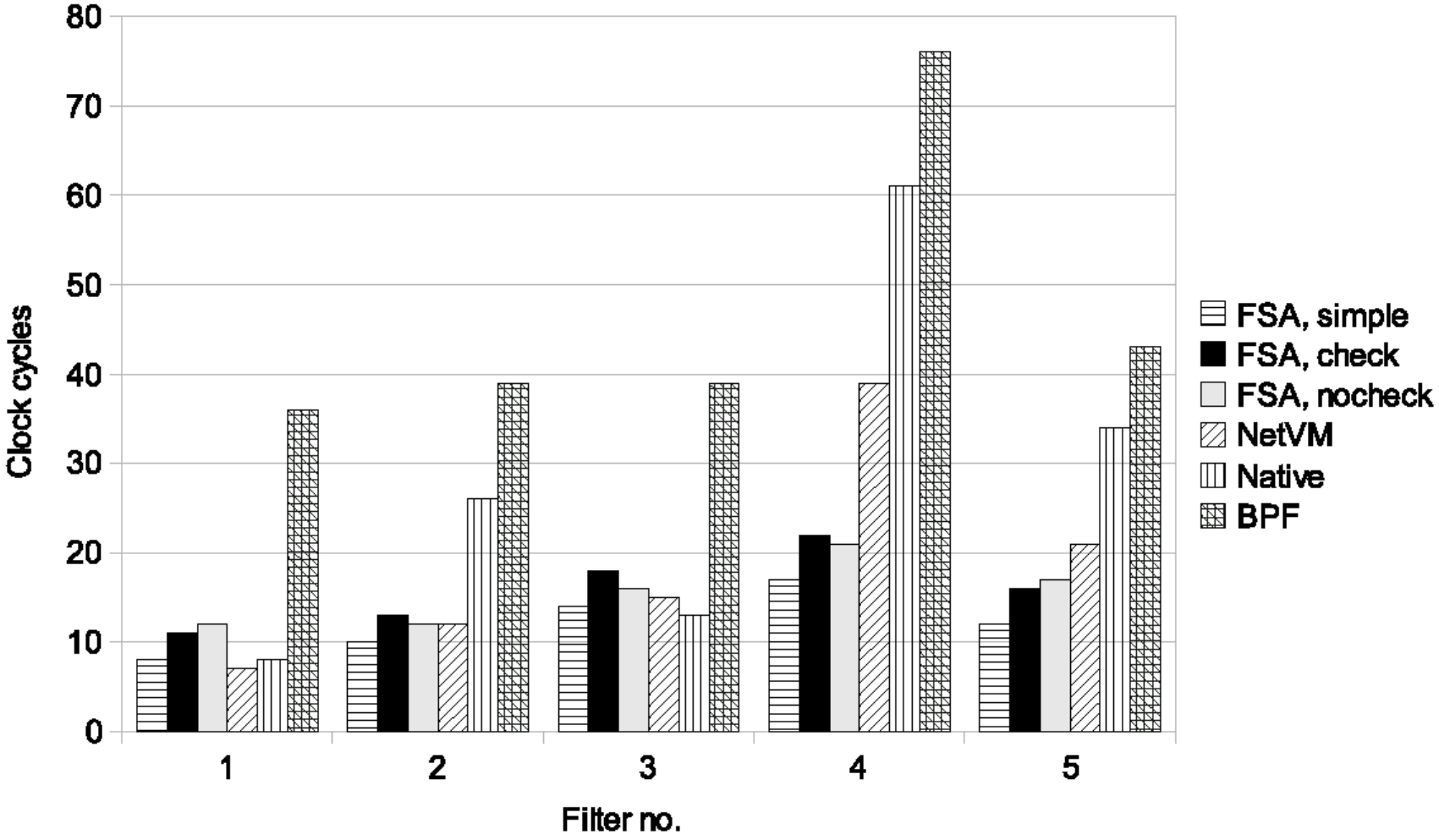
- FSA filters generated by our prototype compared to
  - BPF (jit version)
  - BPF+ (with C backend)
  - NetVM
  - Hand-written
- All run in the same test bench
  - clock cycles measured by RDTSC
  - Linux 2.6.24 Workstation with Intel E8400 Core 2 Duo dual-core processor, 4 GB RAM

Worst-case Filter Performance

filter 1	`ip`
filter 2	`ip.src == 10.1.1.1`
filter 3	`tcp`
filter 4	`ip.src == 10.1.1.1 and ip.dst == 10.2.2.2 and udp.sport === 20 and udp.dport == 30`
filter 5	`ip.src == 10.4.4.4 or ip.src == 10.3.3.3 or ip.src == 10.2.2.2 or ip.src == 10.1.1.1`

- Ad-hoc packet to trigger worst case code path
- No multiple levels of encapsulation
- Cost of VM for NetVM excluded

Worst-case Filter Performance



Expected Scalability

- All operations except the switch case execute in **constant time**
- Switch case with large non-dense case set compiled into binary decision trees (log depth)

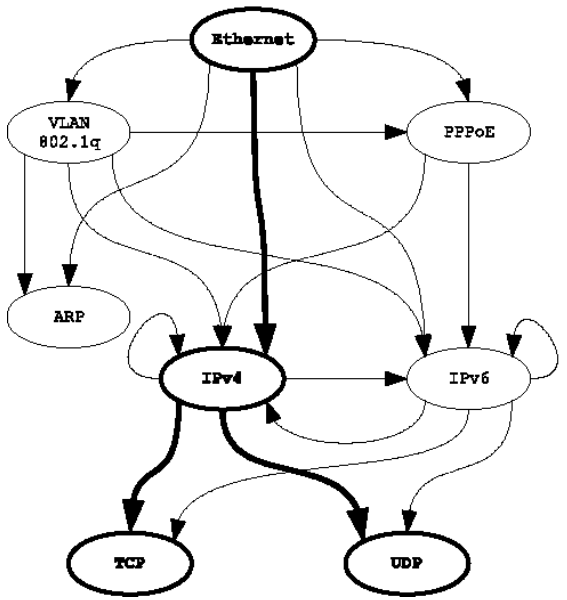
=> Expected asyntotical complexity is **O(log(N))**

Where N: number of cases

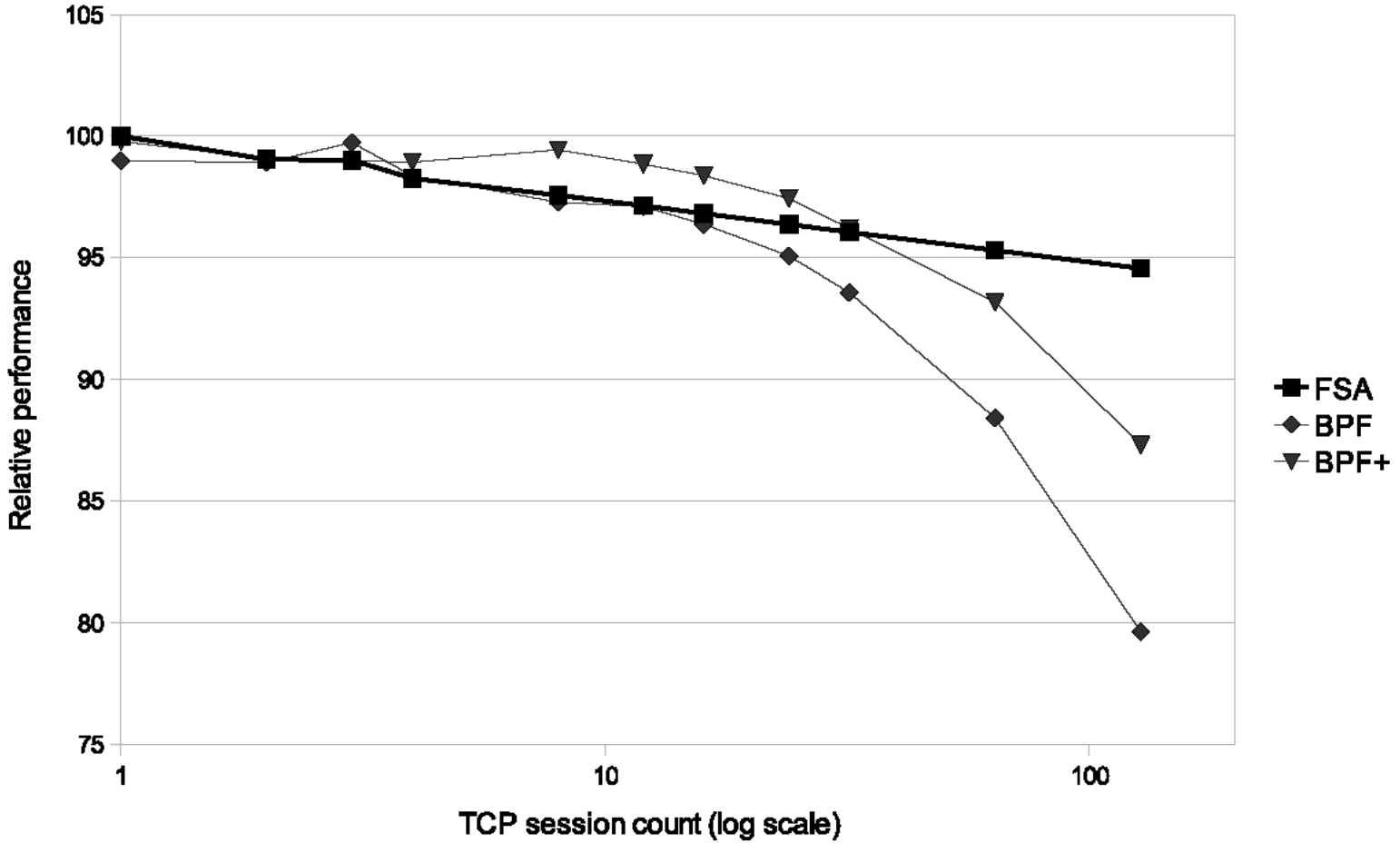
  - Roughly proportional to # filter rules

Scalability Measure

- 1Gb real-world captured packet trace
- 1 Parameterized filtering rule
  - Extract the N most active TCP sessions
- Protocol DB



Measured Scalability

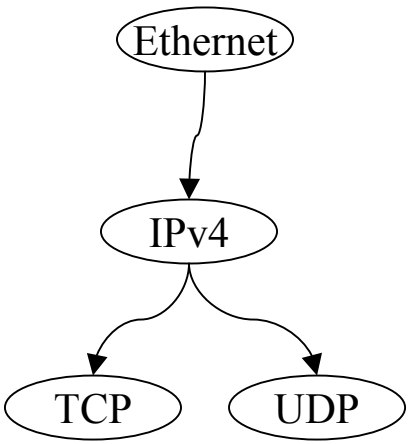


Memory Consumption

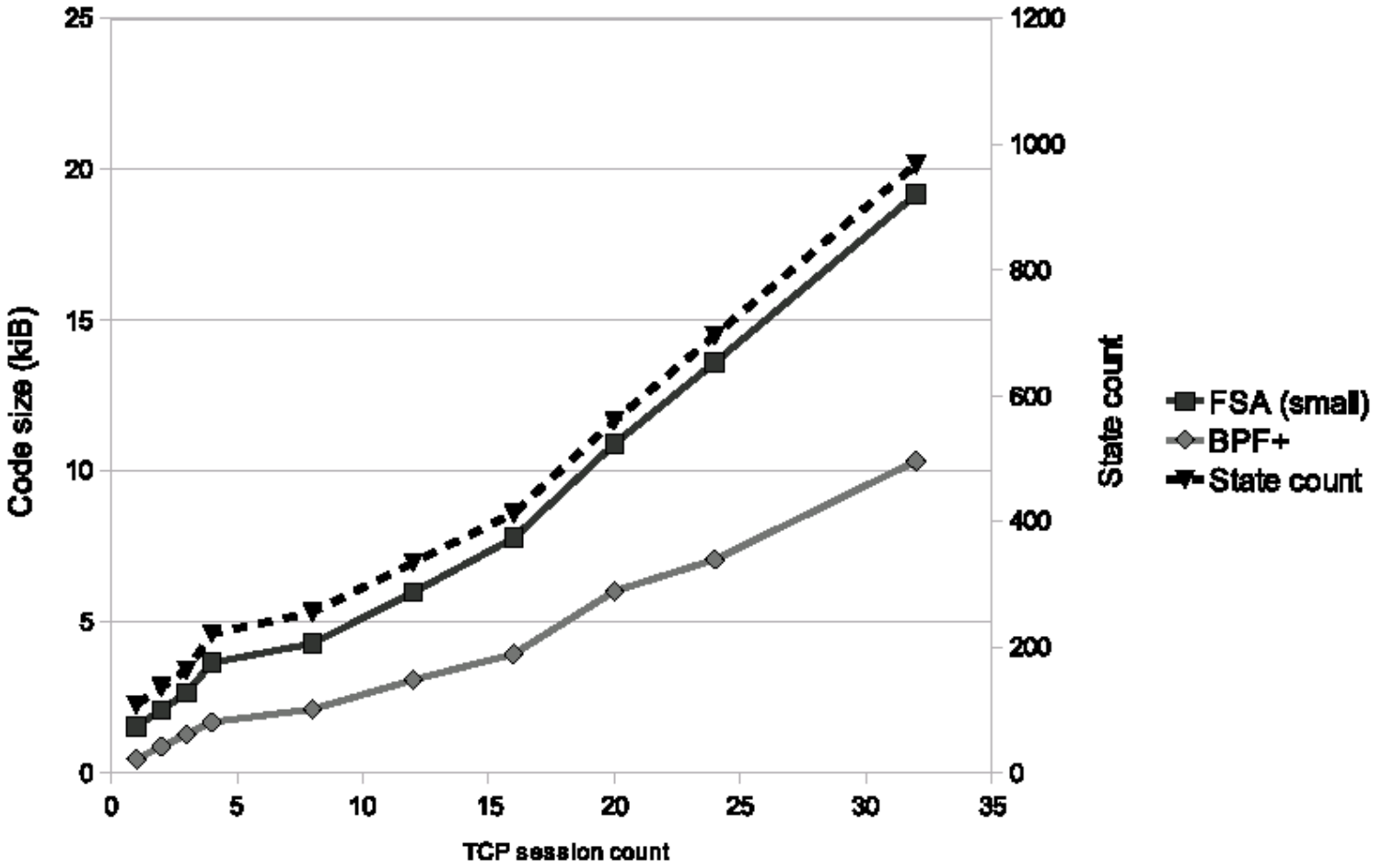
- In theory
  - State space explosion may occur in FSA
    - Example: FSA that recognizes regexp .*s.*s'
    - Determinization of NFA may lead of DFA of $O(2^n)$ states in worst case
- In practice
  - Filters are no free-form as regexp can be
  - Exponential growth found in only 1 case
    - Compare 2 fields against each other

Comparing Memory Consumption

- Filters that recognize increasing number of TCP sessions
- Code size
  - Proportional to state count in FSA filters
- Protocol DB



Memory Consumption Measures



Comments

- Most desirable properties of packet filters achieved by FSA filters
  - High flexibility
    - full support of encapsulation
    - Already tested on protocol DBs with most layer 1-4 protocols
  - Performance similar or higher than best existing approaches
  - Safety guarantee
- Current limitation
  - Generation time does not allow dynamic usage

Future Work

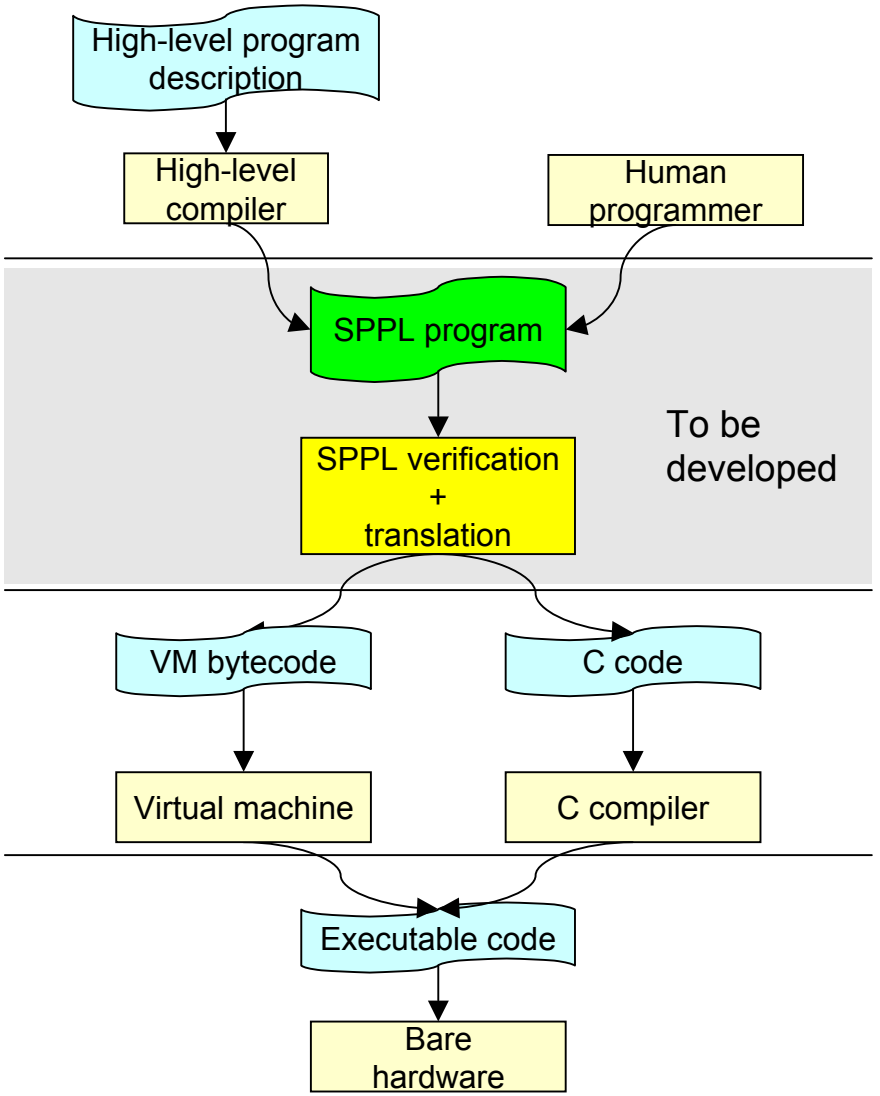
- Improve generation time
  - Better algorithms
  - java->C porting
  - templates for common filter predicates
- Extend the approach to other packet processing applications
  - Demultiplexing already possible in theory
  - Generation time is the most limiting factor

Safe Packet Processing Language

- FSA model good for filters. New model needed for general purpose packet processing
- Goal:
  - Simple language for general purpose packet processing such that some safety properties
    - are easy to check
    - or are automatically guaranteed to hold by construction

  => Safety could be enforced or checked even on untrusted code

Safe Packet Processing Language



How to build SPPL?

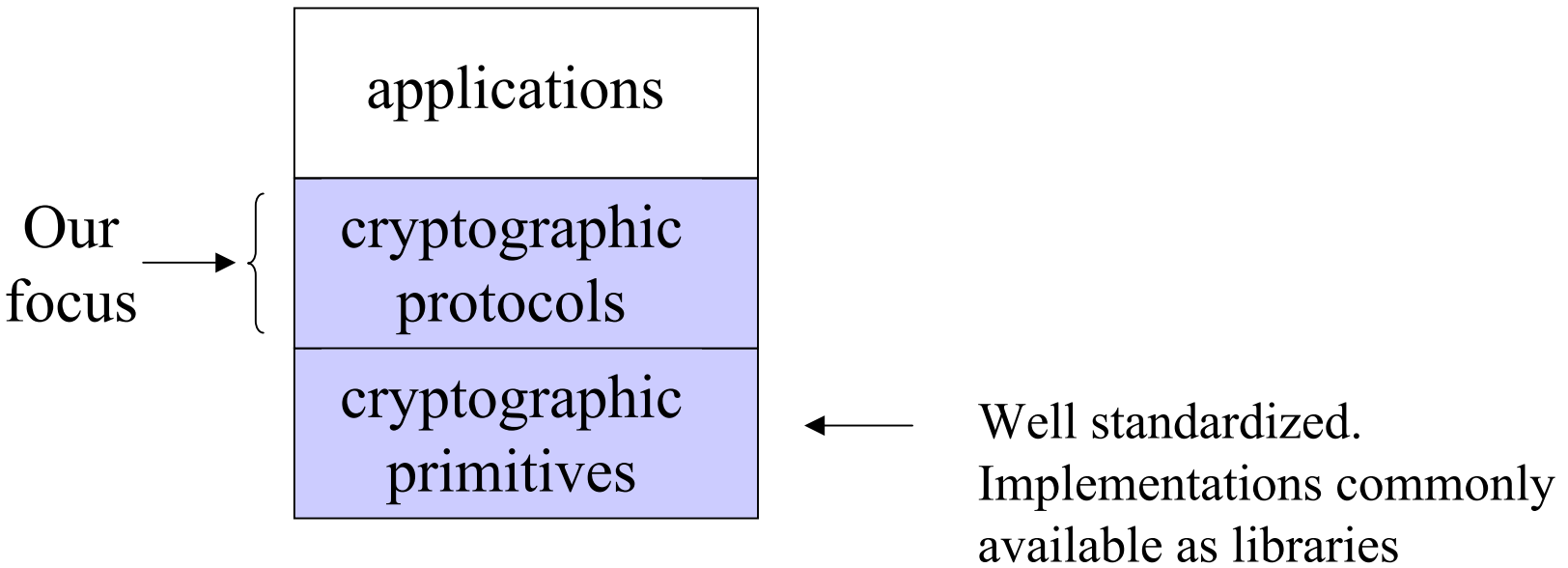
- Key issue to get the goal is constrain SPPL
  - Give no more expressivity than needed (in order not to lose decidability)
- How is it possible?
  - Some starting ideas:
    - Finite state (+ finite lookup tables)
    - Finite control variables + data streams (packets)
- Methodology
  - Start with simple core and try coding typical applications
  - Add new features as far as needed

Contributions are welcome…

- Typical applications?
- Safety properties to be considered?
- Most common faults?

Cryptographic Protocols

- software components that implement security-related services in distributed environments



Cryptographic protocols: Challenges

- despite their apparent simplicity, they may exhibit subtle flaws, difficult to find by hand:
  - difficult foreseeing all the possible behaviors of a potential attacker
  - especially when concurrent sessions are possible

The aim of our work

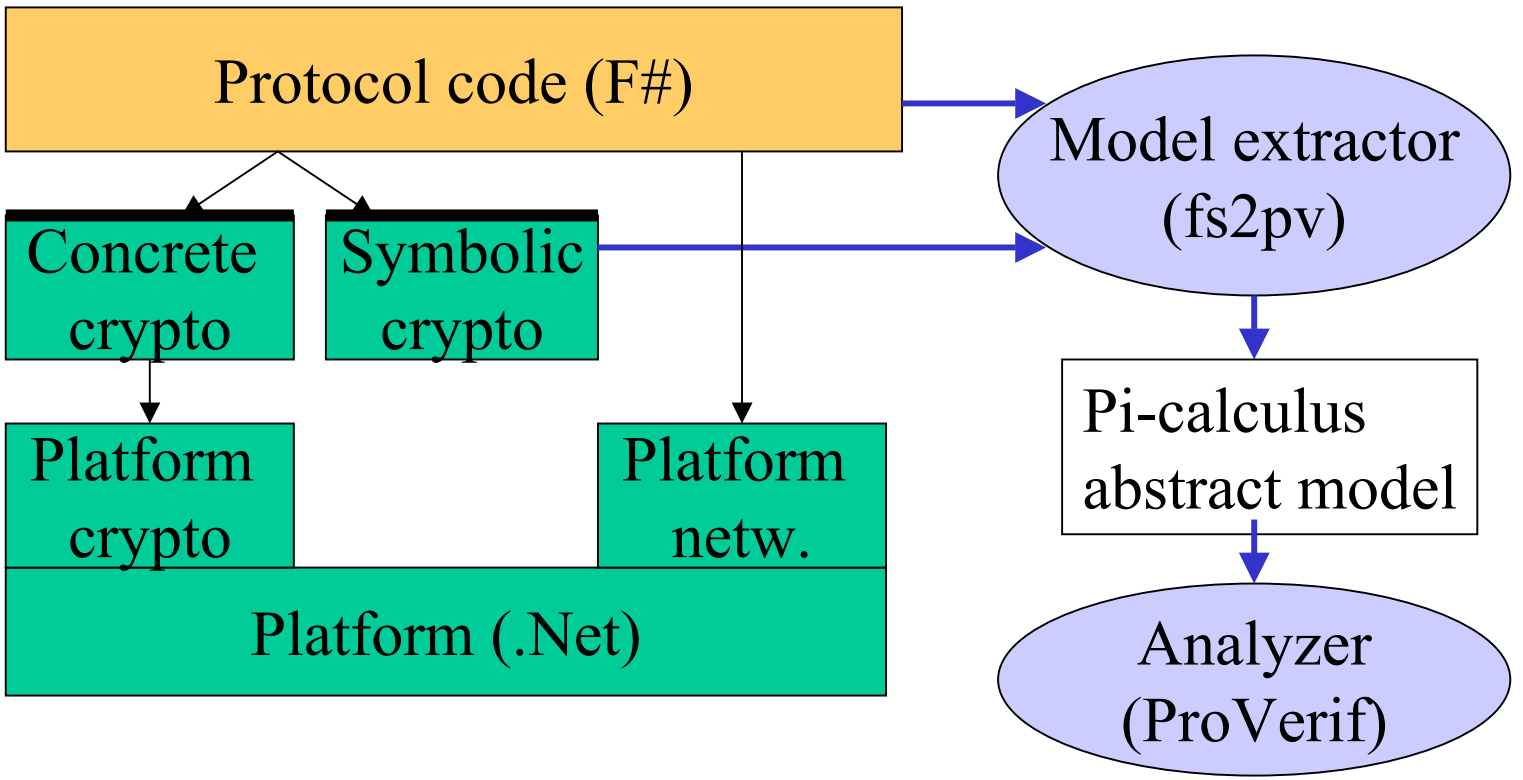
- Much research work now available on (automated) formal verification of abstract protocol models
  - Dolev-Yao formal model (perfect cryptography, abstract data representations)
  - detection of logical protocol errors
  - in abstract models of real protocols
- Yes, but …

…how secure is my protocol implementation?

=> Need to ensure abstract models are sound abstractions of protocol implementations

A Model Extraction Approach

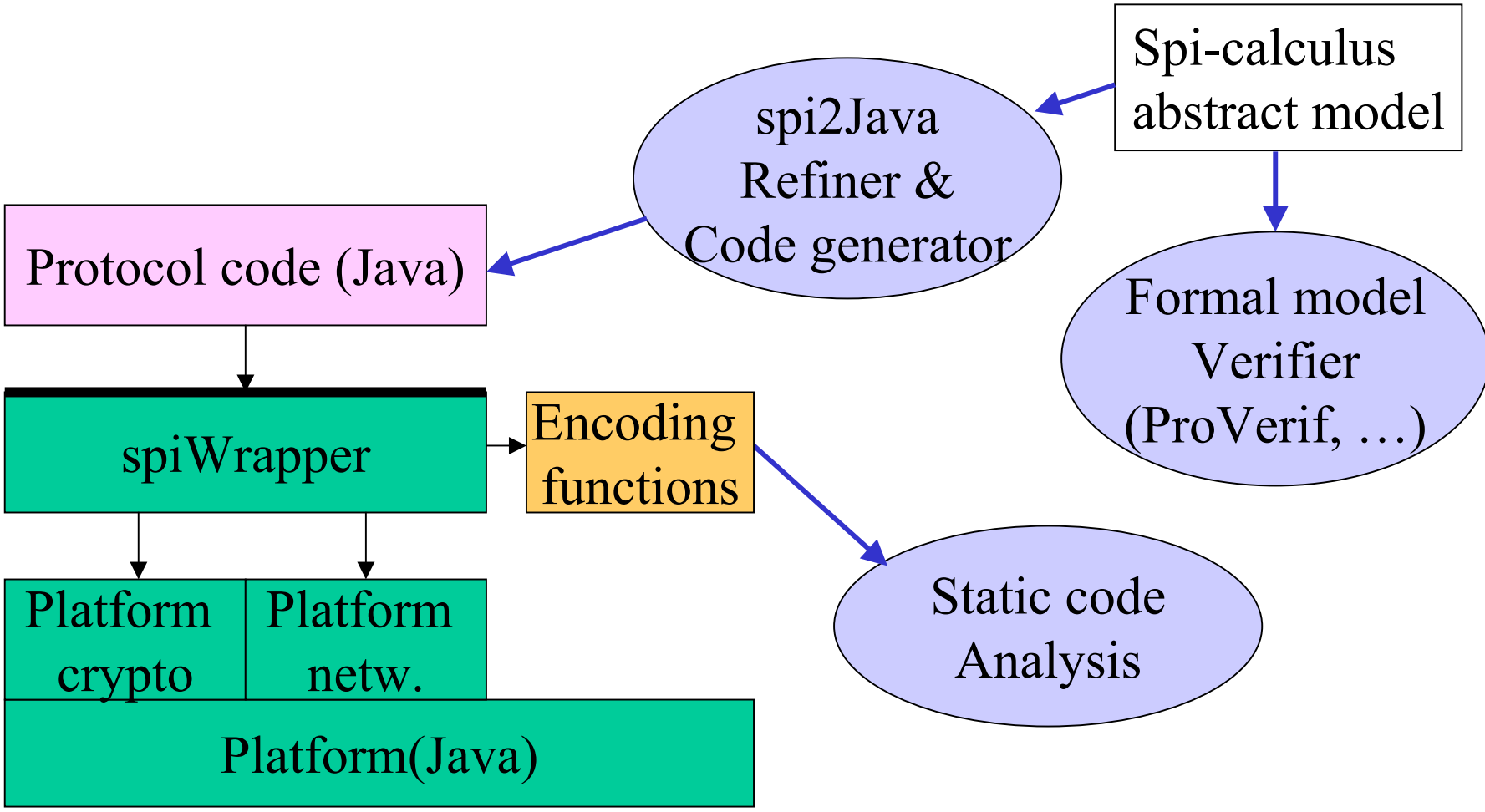
- Bhargavan, Fournet, Gordon, Tse, CSFW 2006



Downsides

- F# is quite unusual language
- Many constraints on protocol code
  - Actually, no legacy code can be analyzed
- The extracted formal model may be too large to be analyzed
  - Concrete data handling may be quite complex

Our (Model-based) Approach



Theoretical Result

- Encoding functions for marshalling/unmarshalling can be safely left out of the formal model provided
  - The intruder knows marshal parameters/algorithms
  - Result of marshalling depends only on data to be marshalled and marshalling parameters/algorithms

Case Study Experiment

- Write SSH Transport protocol in Spi calculus
- Verify Security properties by Proverif
  - Secrecy of shared key
  - Server authentication
    - agreement on *final hash*
- Generate interoperable client implementation

Some Measures on Code

- Reliability
  - SSHredder suite
- Interoperability
  - Tests with third party servers

Server name	Test result
OpenSSH_4.2p1	
PragmaFortress 4.0	
cryptlib (KpyM 1.13)	
lshd-2.0.2	
dropbear_0.48	
3.2.9.1 SSH Secure Shell	

Some Measures on Code II

- Metrics
  - A small amount of code is manually written
    - quick development
  - SpiWrapper offers encapsulation
    - protocol logic is rather small

Package	TLOC	MLOC
spiWrapper	2064	1267
spiWrapperSSH	1557	733
sshClient	218	169

Conclusions and Future Work

- Case study has shown the approach can work on real protocols
- Model-implementation gap only partially filled:
  - Our reasoning is not about cryptography
  - There are ways of attacking a protocol that are not modelled (e.g. exploiting code vulnerabilities, covert channels, etc)
- Usability could be improved (turn Spi-calculus into user-friendly language with pre-defined properties)
- Same approach could be extended to other target languages (e.g. C)
- Code generation could be extended to encoding functions

Questions?

More information available on the web

- FSA packet filters
  - Report submitted for publication available at
    http://staff.polito.it/riccardo.sisto/cisco/report.pdf
- Spi2java
  http://spi2java.polito.it