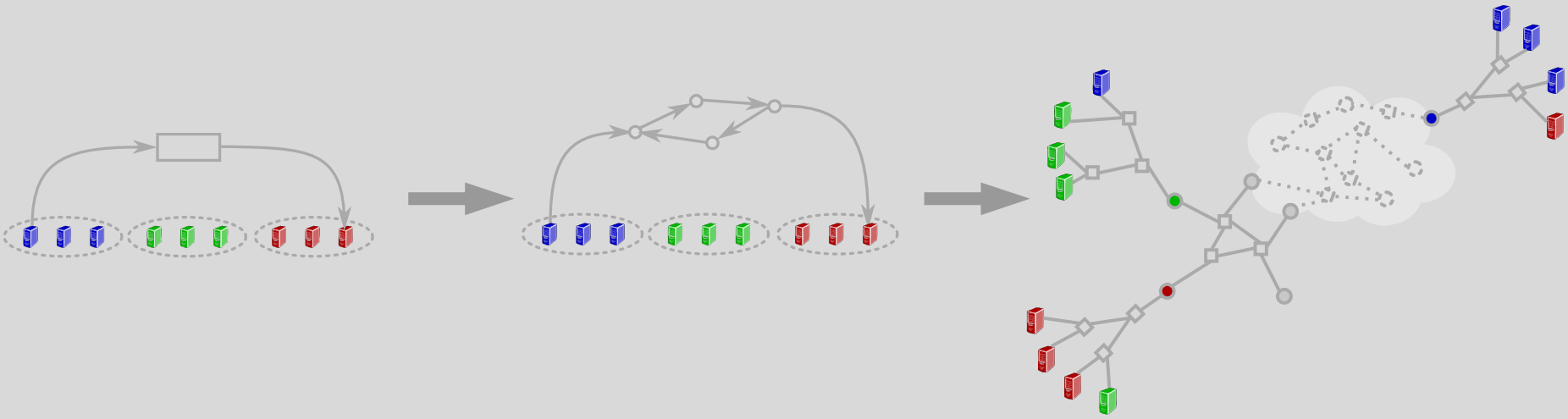
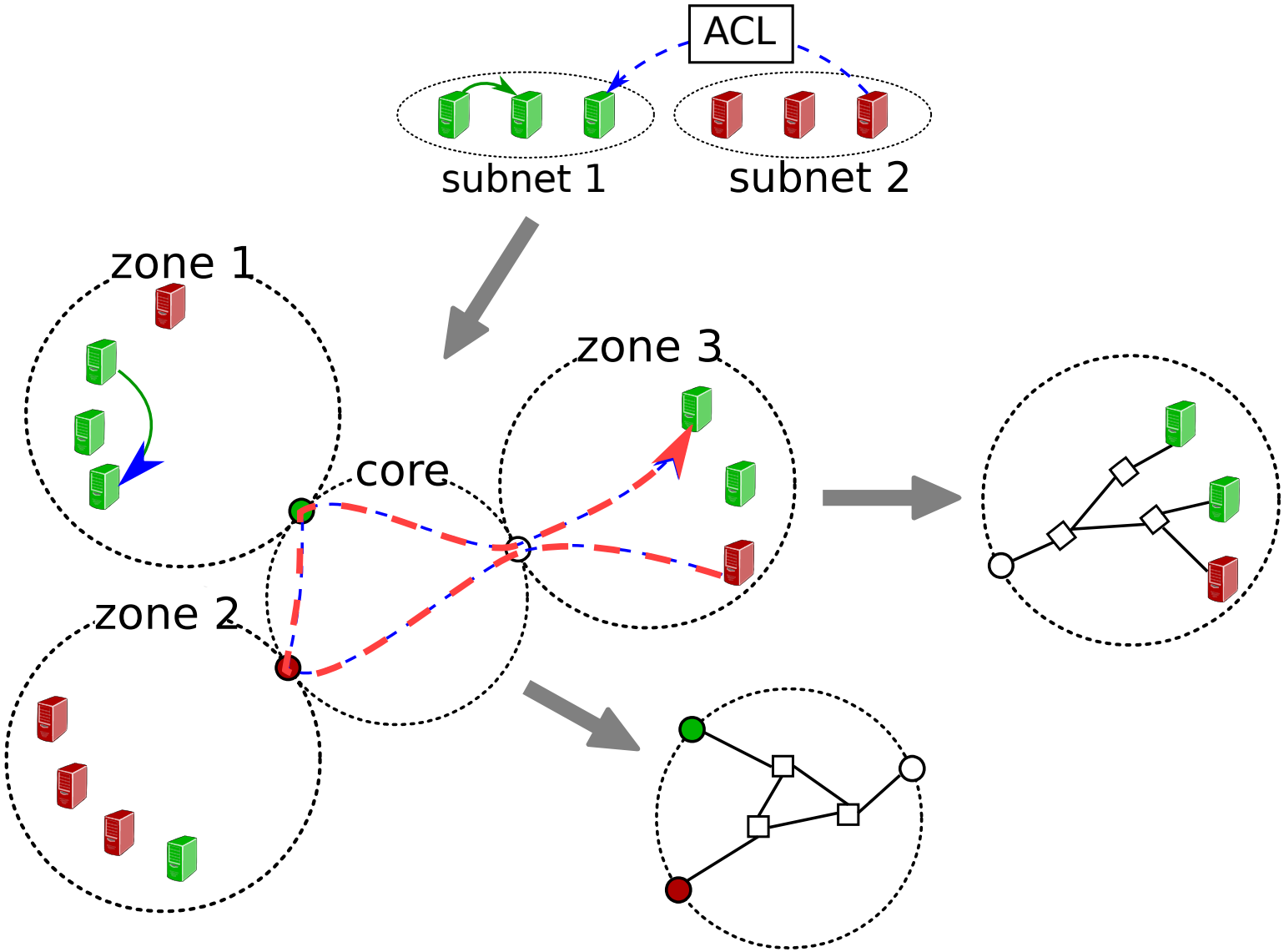
COCOON: CORRECT-BY-CONSTRUCTION NETWORKS USING STEPWISE REFINEMENT

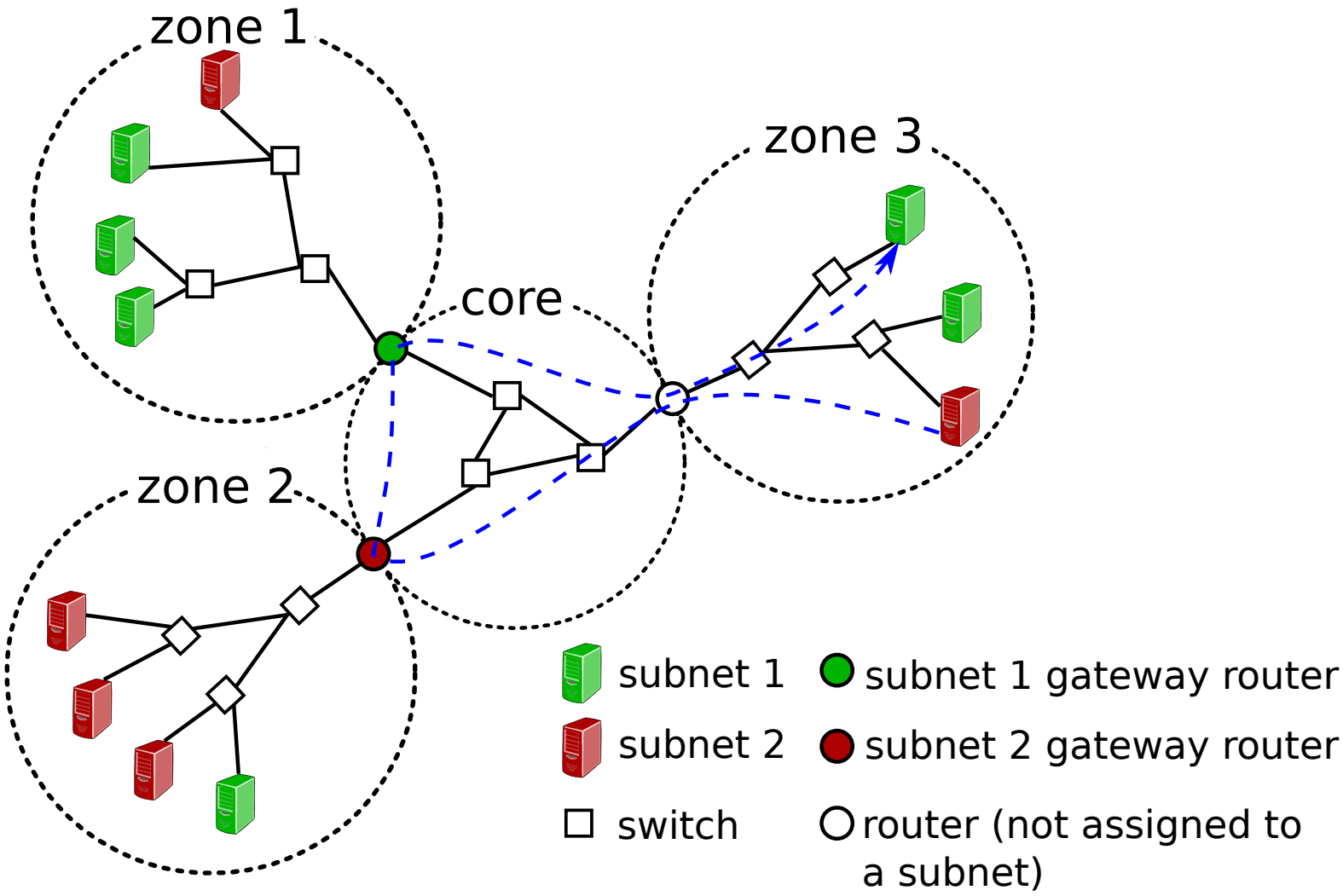
Leonid Ryzhyk **Nikolaj Bjorner** **Marco Canini** **Jean-Baptiste Jeannin**
Cole Schlesinger **Douglas Terry** **George Varghese**



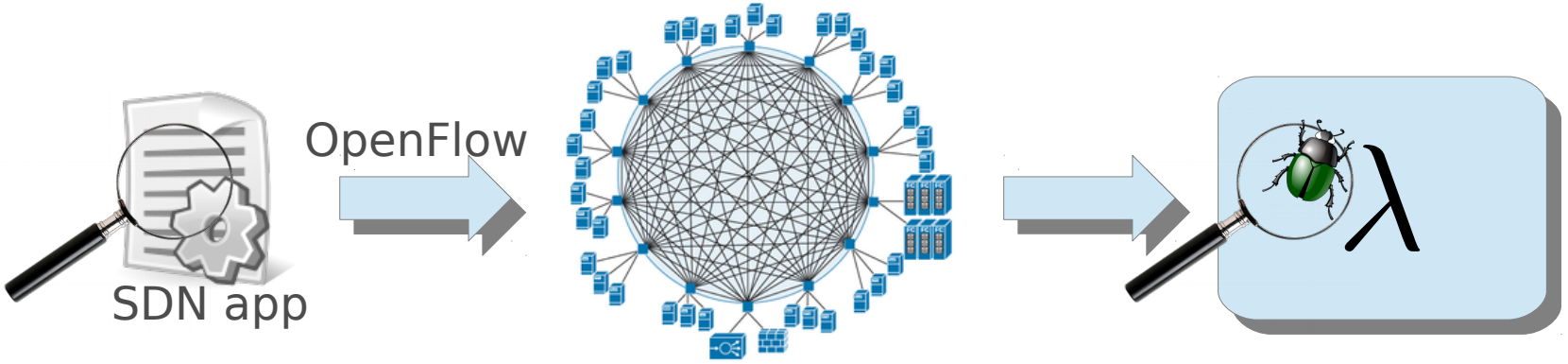
RUNNING EXAMPLE: CAMPUS NETWORK



RUNNING EXAMPLE: CAMPUS NETWORK



NETWORK VERIFICATION: CURRENT PRACTICES



Option 1: Dataplane verification
(NetPlumber, HSA, Veriflow)

- Fixing bugs in a deployed network takes time; may not avoid the downtime

Check for:

- ✔ Loop freedom
- ✔ Black holes
- ✔ Reachability
- ✔ Isolation

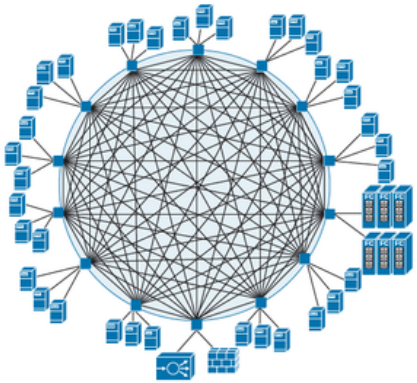
Option 2: Controller verification
(Vericon, FlowLog)

- Limited scalability

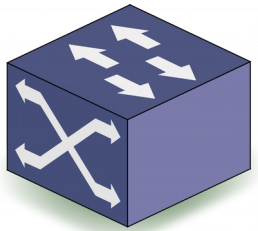
Common to both approaches:

- Property-based verification does not guarantee correctness

NETWORK VERIFICATION IN A NUTSHELL







Big switch abstraction

Option 1: Dataplane verification
(NetKAT)

- Limited scalability

REQUIREMENTS

Ideally, network verification should be:

1. Scalable

(works at DC scale)



2. Static

(verifies all possible configurations)



3. Exhaustive

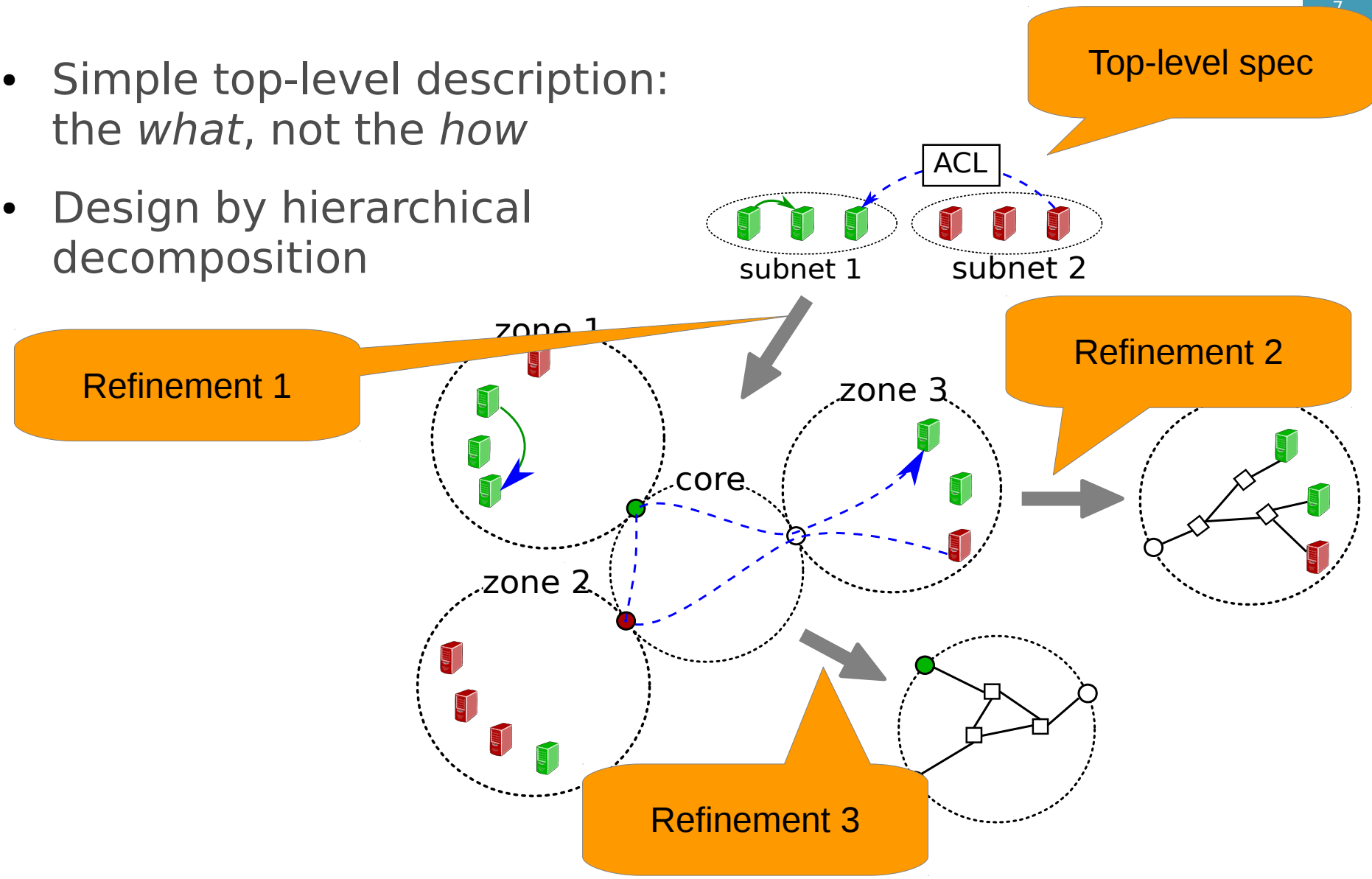
(misses no bugs)



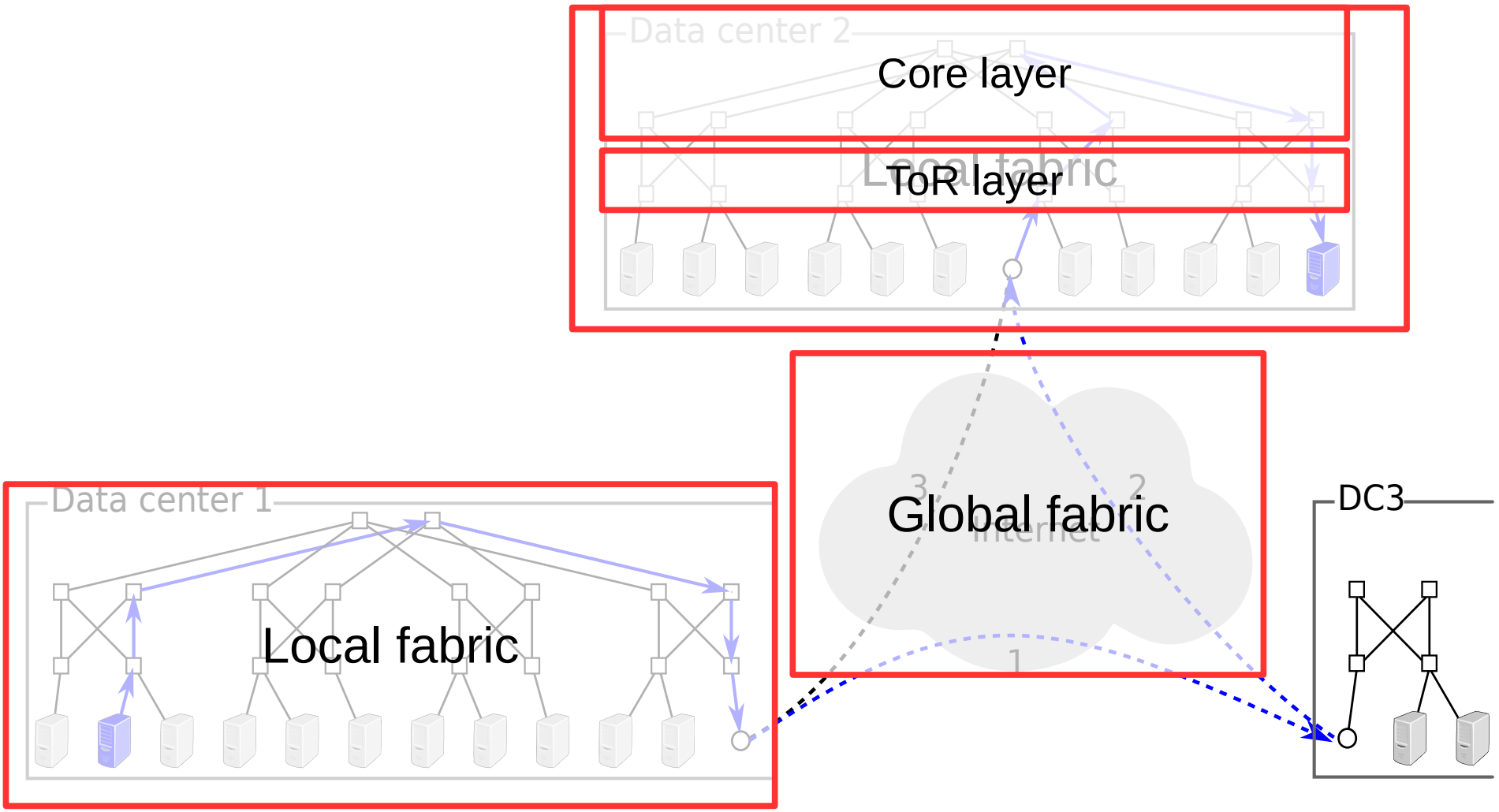
- **State of the art: pick 1 out of 3**

OBSERVATIONS

- Simple top-level description: the *what*, not the *how*
- Design by hierarchical decomposition



DECOMPOSING A WAN



MORE EXAMPLES

- Virtual network is decomposed into
  - Physical fabric
  - Virtual fabric
- Cellular network is decomposed into
  - Edge (base stations)
  - Core
  - Internet gateway

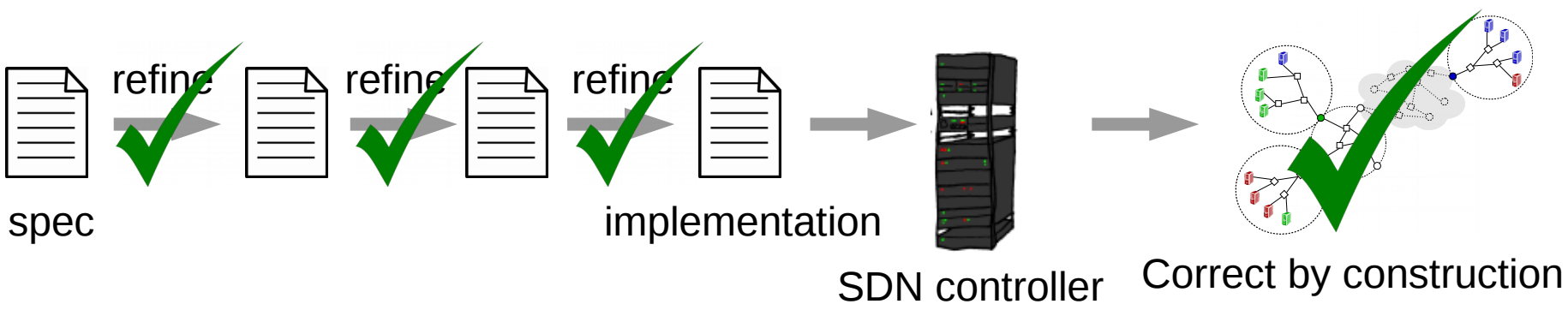
Exposing this structure enables efficient compositional verification

COCOON: **CO**rrect by **CO**nstruction **N**etworking

We propose Cocoon:

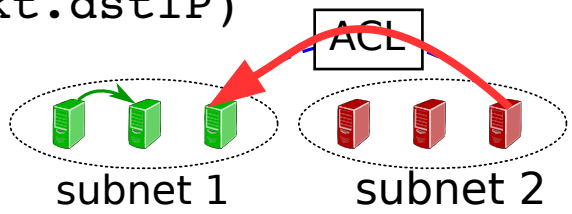
- SDN design method
- Programming language
- Verifier

Cocoon achieves *scalable*, *static*, *exhaustive* verification (3 out of 3!) via a network design process that focuses on correctness.



EXAMPLE COCOON SPECIFICATIONS


```
role HostOut[IP4 addr] | cHost(addr) =
  filter ip2subnet(pkt.srcIP)==ip2subnet(pkt.dstIP)
         or acl(pkt);
  filter cHost(pkt.dstIP);
  send HostIn[pkt.dstIP]
```



Runtime-Defined F

```
function cHost(I
function cSubnet(
function acl(Packet p): bool
function ip2subnet(IP4 ip): vid_t
```

Must return valid subnet ID

```
cHost(addr) =
  addr=={172.30.0.21}||
  addr=={172.30.0.22}||
  ...
```

```
ip2subnet(ip) =
  172.30.*.* → subnet1
  172.20.*.* → subnet2
  ...
```

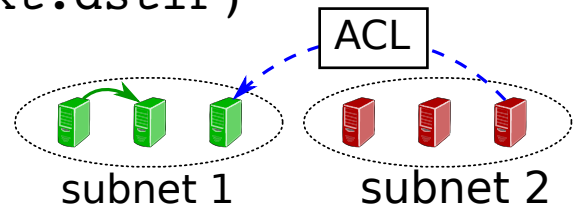
Assumption:

$$\forall addr.cHost(addr) \implies cSubnet(ip2subnet(addr))$$

```
assume(IP4 addr) cHost(addr)=>cSubnet(ip2subnet(addr))
```

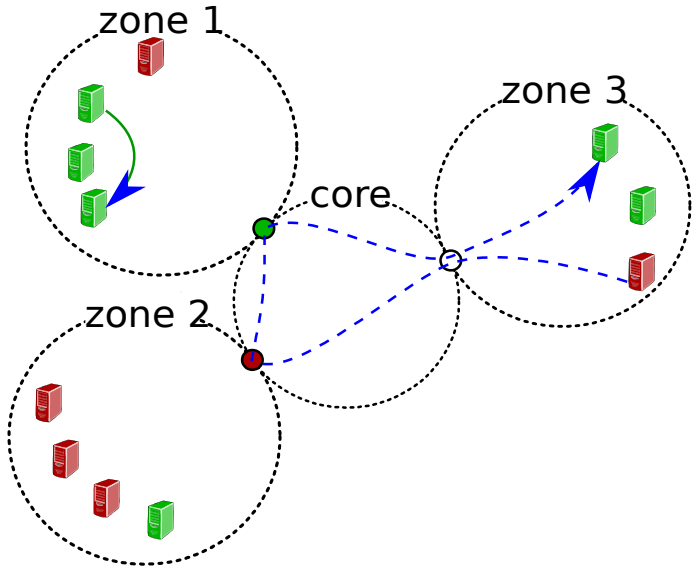
REFINEMENT EXAMPLE


```
role HostOut[IP4 addr] | cHost(addr) =
  filter ip2subnet(pkt.srcIP)==ip2subnet(pkt.dstIP)
         or acl(pkt);
  filter cHost(pkt.dstIP);
  send HostIn[pkt.dstIP]
```




```
refine HostOut {
  role HostOut[IP4 addr] | cHost(addr) =
    ...
    send RouterZoneIn[zone(addr)]

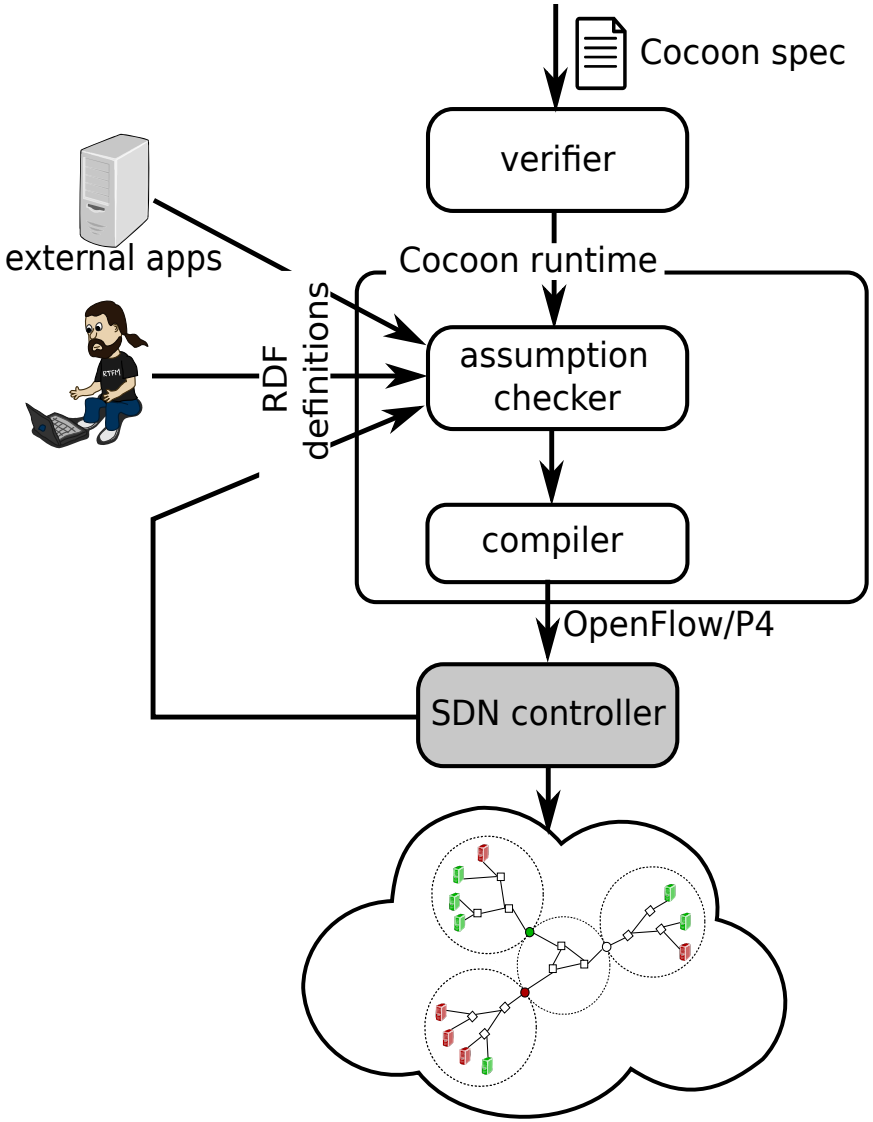
  role RouterZoneIn[zid_t] =
    ...
...
}
```



2-PHASE VERIFICATION

- Refinements + assumptions specify static network design
  - Verified statically
- RDFs encapsulate runtime configuration
  - Checked at runtime against assumptions

COCOON ARCHITECTURE



IMPLEMENTING VERIFICATION

- Role semantics: $R : LPkt \nrightarrow 2^{2^{LPkt}}$
- Role refinement: $\hat{R}(pkt) \subseteq R(pkt)$
- We convert this program to Boogie and use the Corral model checker
  - Enforce static bound on the number of network hops to achieve completeness
- Assumptions are converted to SMT and checked using Z3

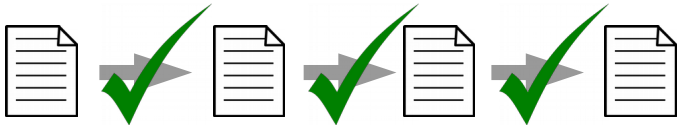
CASE STUDIES

- **B4-style WAN**
  [Jain et al. B4: Experience with a Globally-Deployed Software Defined WAN]
- **NSX-style network virtualization framework**
  [Koponen et al. Network Virtualization in Multi-tenant Datacenters]
- **Enterprise network**
  [Sung et al. Towards Systematic Design of Enterprise Networks]
- **F10**
  [Liu et al. F10: A Fault-Tolerant Engineered Network]
- **Stag**
  [Lopes et al. Automatically verifying reachability and well-formedness in P4 Networks]
- **iSDX**
  [Gupta et al. An Industrial-Scale Software Defined Internet Exchange Point]

PERFORMANCE (static verification)

case study	LOC		#refines	verification time (s)	
	total	high-level		compositional	monolithic
WAN	305	18	6	10	>3600
virtualization	678	97	1	6	6
enterprise	342	50	4	16	>3600
F10	262	52	2	19	57
stag	283	47	1	2	2
iSDX	190	21	2	3	3

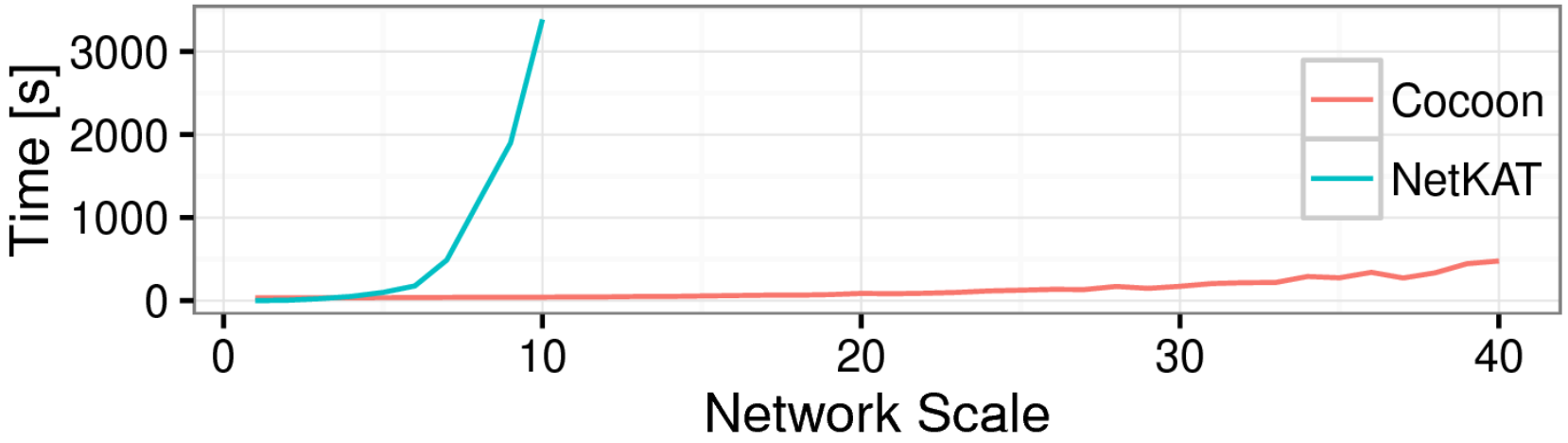
Compositional:



Monolithic:

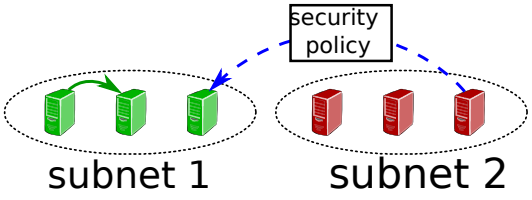


PERFORMANCE (runtime verification)

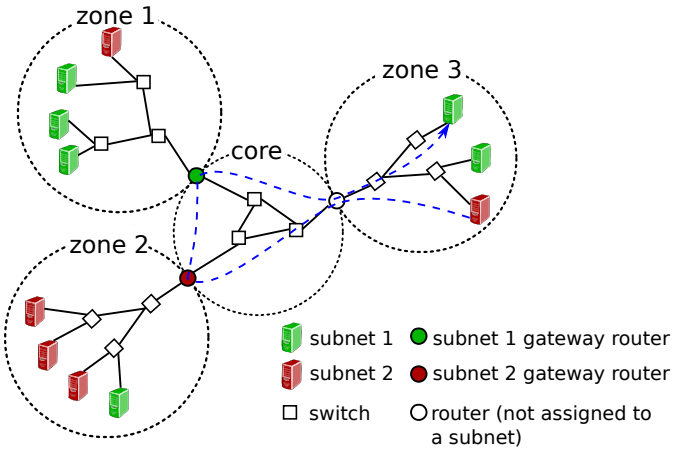


Scale	Hosts	Switches	NetKAT Policy	Flowtable Rules
2	8	11	1,559	830
5	17	23	5,149	3,299
15	47	63	46,094	31,462
25	77	103	151,014	89,216
40	122	163	496,268	212,925

COCOON VS TRADITIONAL NETWORK VERIFICATION



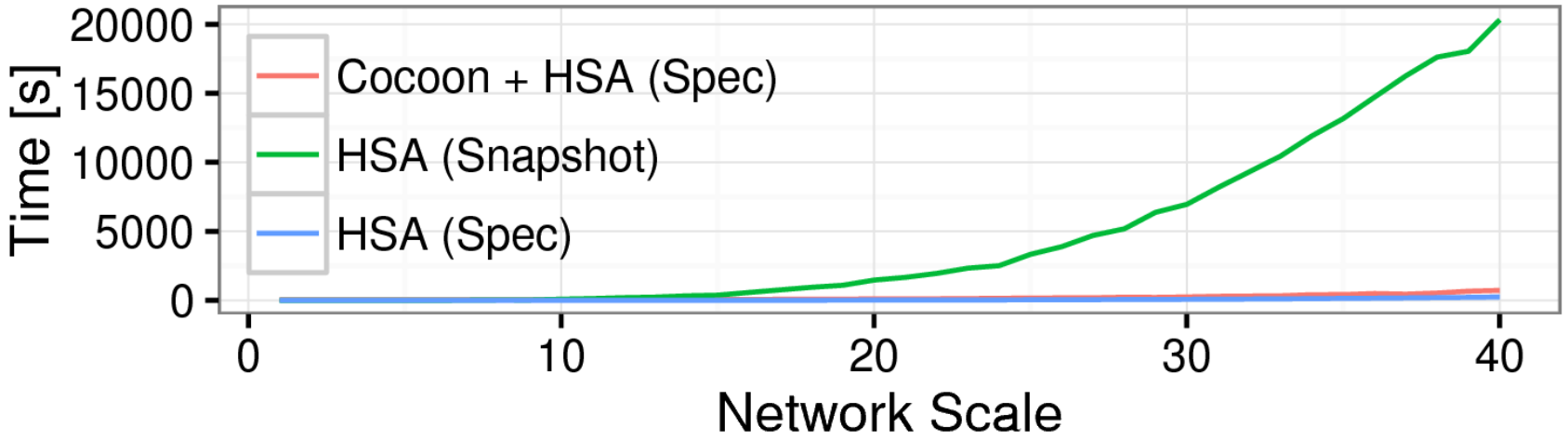




HSA/
Veriflow/
...

Correctness spec

PERFORMANCE (Cocoon + HSA)



Scale	Hosts	Switches	NetKAT Policy	Flowtable Rules
2	8	11	1,559	830
5	17	23	5,149	3,299
15	47	63	46,094	31,462
25	77	103	151,014	89,216
40	122	163	496,268	212,925

CONCLUSION

- Design-by-refinement works well for networks:
  - Allow concise high-level specifications
  - Well-defined module boundaries
  - Verification is feasible for a single refinement: no pointers, concurrency, dynamic memory allocation, etc.

Source code, case studies:

https://github.com/ryzhyk/cocoon