Reverse Engineering for Security

Pramod Subramanyan

spramod@cse.iitk.ac.in

Indian Institute of Technology, Kanpur

6th Workshop on Design Automation for Understanding Hardware Designs

March 29 2019

This work was supported in part by the Semiconductor Research Corporation (SRC).

acknowledgements

- Srinivas Devadas
- Bruno Dutertre
- Jason M. Fung
- Dejan Jovanovic
- Hareesh Khattri
- Ilia Lebedev
- Abhranil Maiti
- Aarti Gupta
- Adria Gascon
- Wenchao Li
- Sharad Malik
- Kanika Pasricha
- Dillon Reisman
- Sanjit Seshia
- Rohit Sinha
- Adriana Susnea
- Ashish Tiwari
- Nestan Tsiskaradze
- Weikun Yang
- Yakir Vizel

outline

why do we care about HW security?

view from the bottom (reconstructing trust)

view from the top: security property verification (constructing trust)

is there a middle-ground where the ‘twain shall meet?

in the news: security breaches galore

1B records

40m cards

80m records

100TB of data

500M records

$400B in losses worldwide due to cybercrime [McAfee, 2014]

not even the police are safe

Georgia county paid ransom of $400,000 to cyber-criminals!

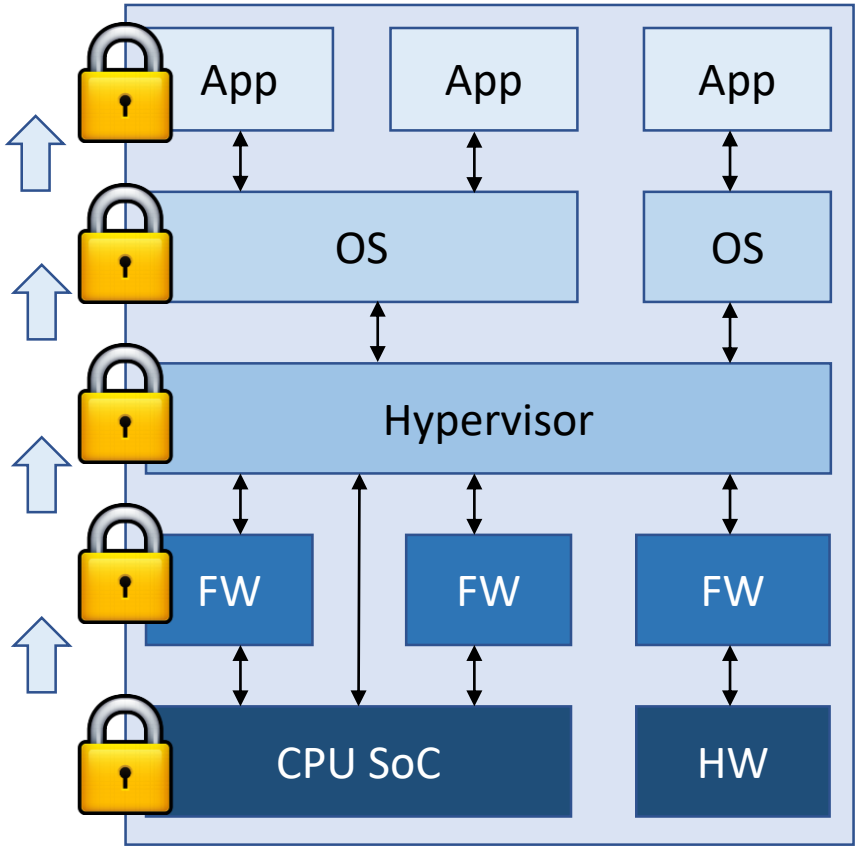
“everything we have is down … we're operating by paper in terms of reports and arrest bookings.”

https://www.zdnet.com/article/georgia-county-pays-a-whopping-400000-to-get-rid-of-a-ransomware-infection/

industry response: move all the thingz to HW

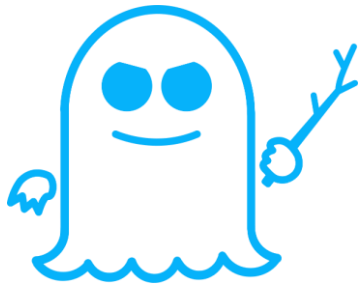
examples of HW security primitives

- HW-supported SW isolation: SGX, Keystone, TrustZone
- SW authentication/measurement: SecureBoot, TXT
- HW support for code protection: e.g, MPX, CFI mechanisms



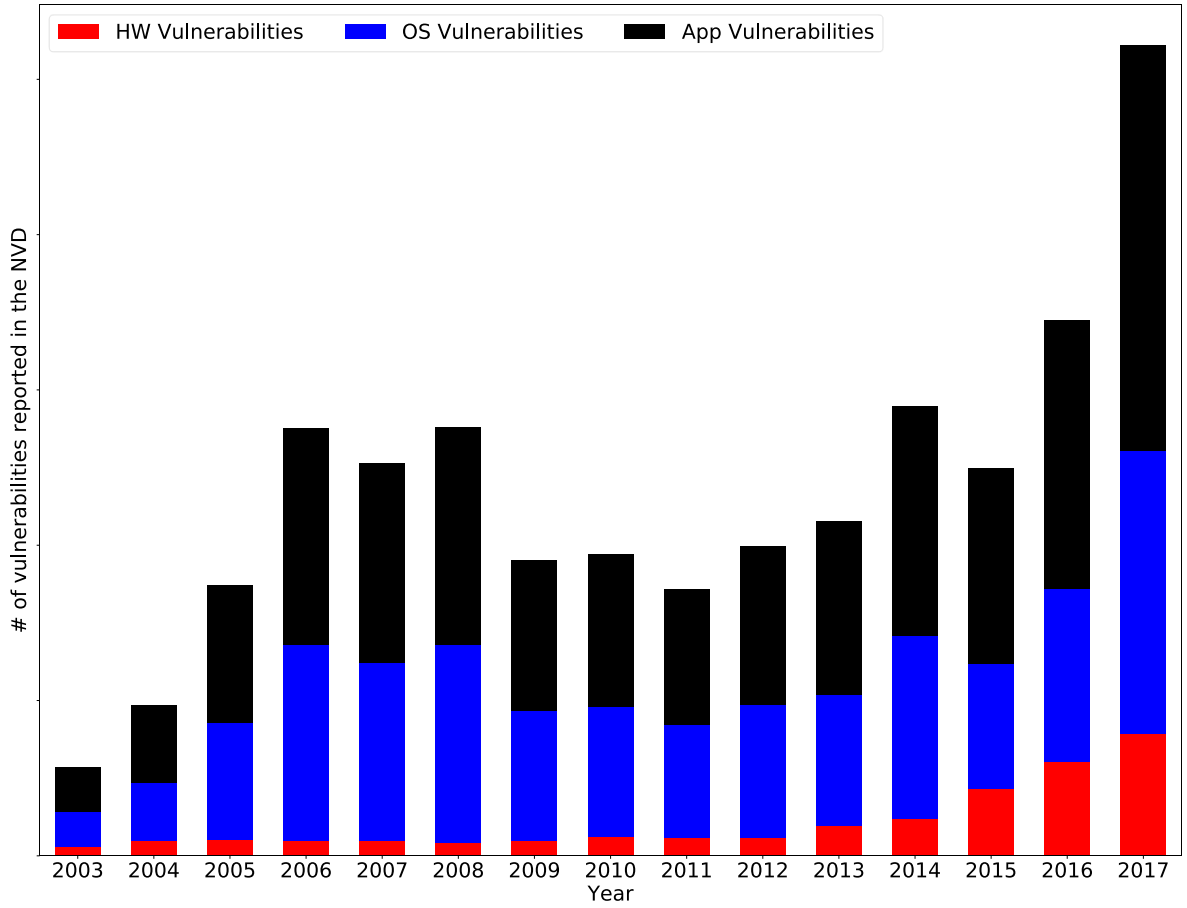
all driven by the perception that hardware is “harder” to “hack” than software

but if we learned one thing in 2018 …



hardware can have security vulnerabilities too

data supports growing HW vulns claim



outline: revisited

we should care about HW security!

can we reconstruct trust in an untrusted component? (bottom-up)

can we prove a trusted component is trustworthy? (top-down)

what's the connection between these two?

two views of hardware vulnerabilities

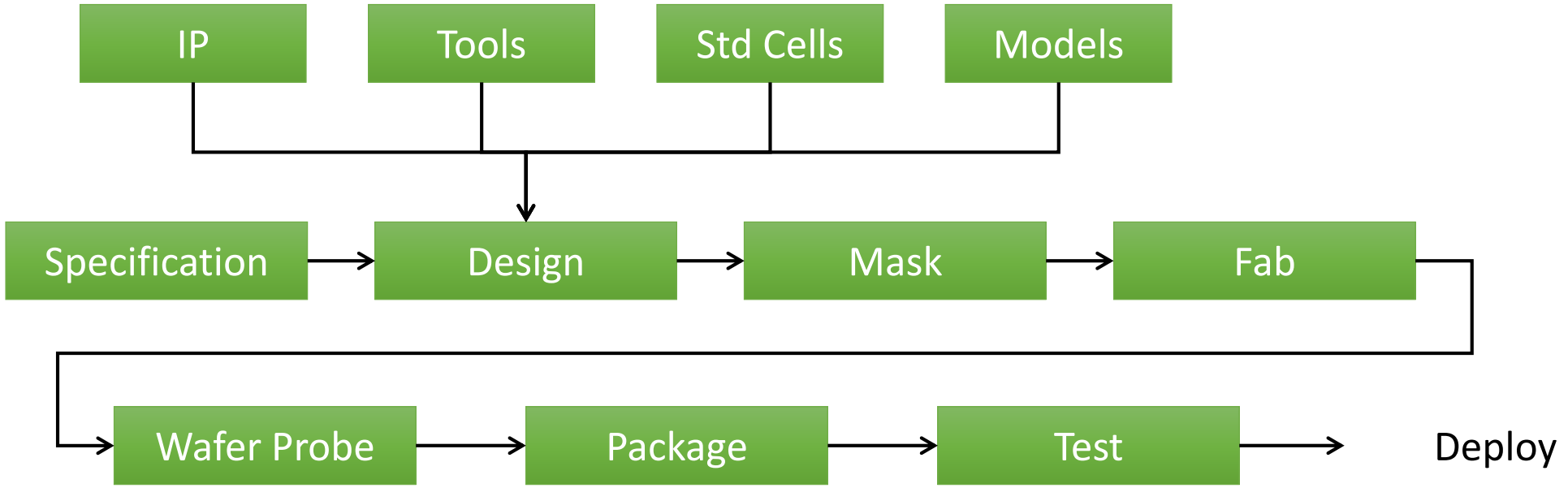
1. someone untrusted controls part of my design

   (can I somehow analyze and then trust this design?)

2. the design is trusted, but is it trustworthy?

   (what can I do to convince myself that the design's security behavior is what I intended)

what could be untrusted in my IC?

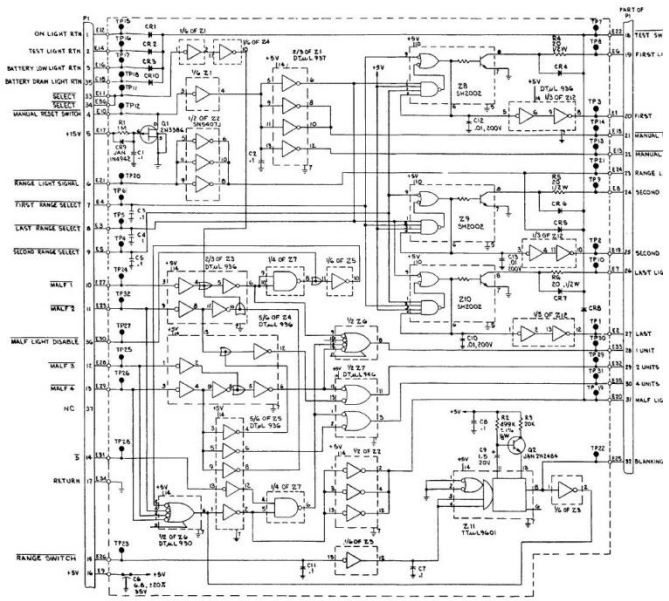


[Brian Sharkey, *TRUST in Integrated Circuits Program: Briefing to Industry*, DARPA MTO, 26 March 2007]

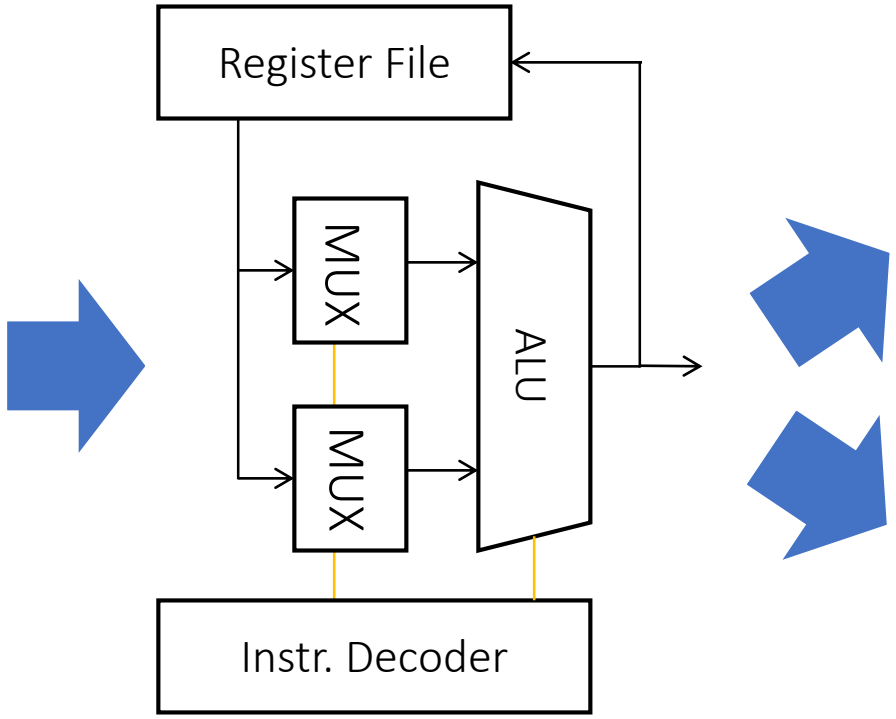
Trusted

Untrusted

one way of reconstructing trust



Source: http://miscpartsmanuals2.tpub.com/TM-9-1240-369-34/TM-9-1240-369-340115.htm

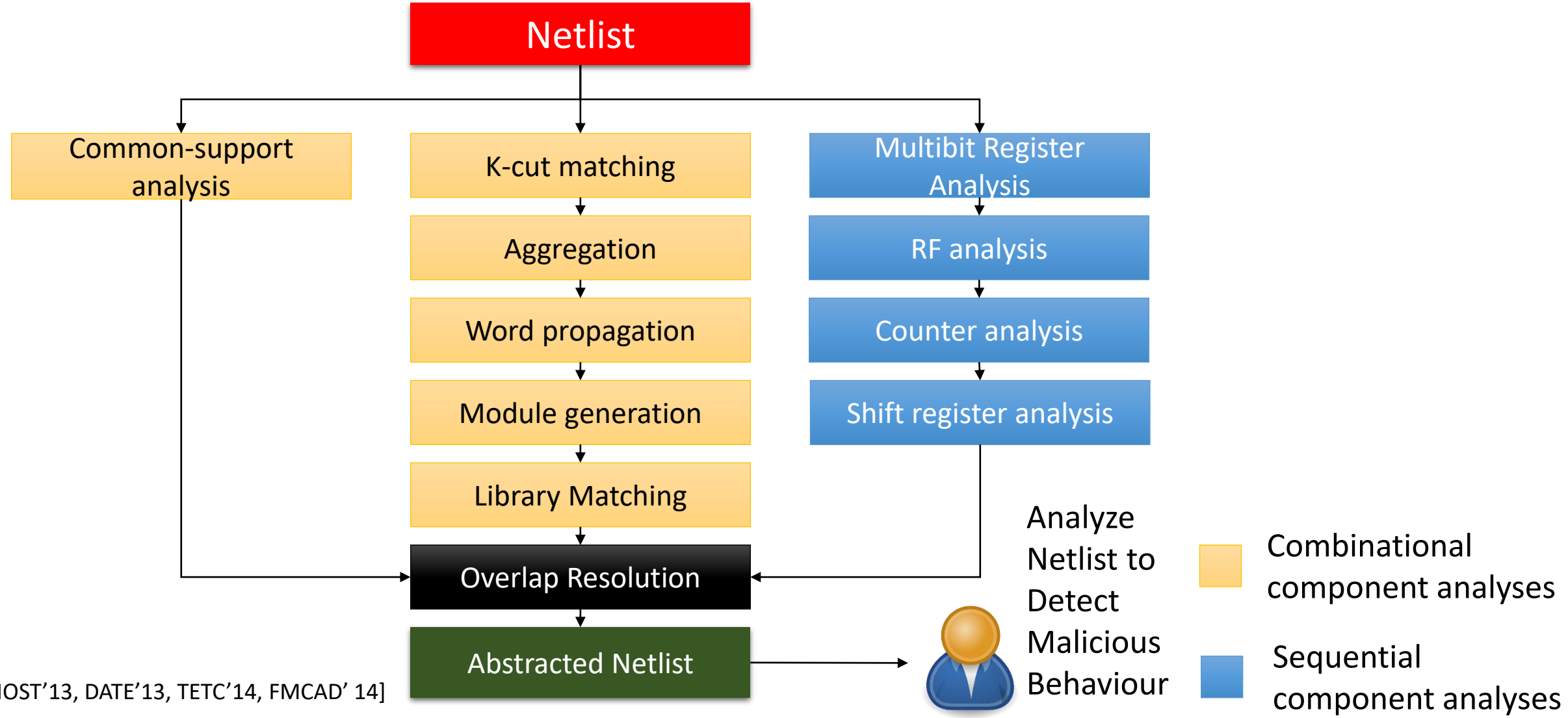


Algorithmic analysis

Manual analysis

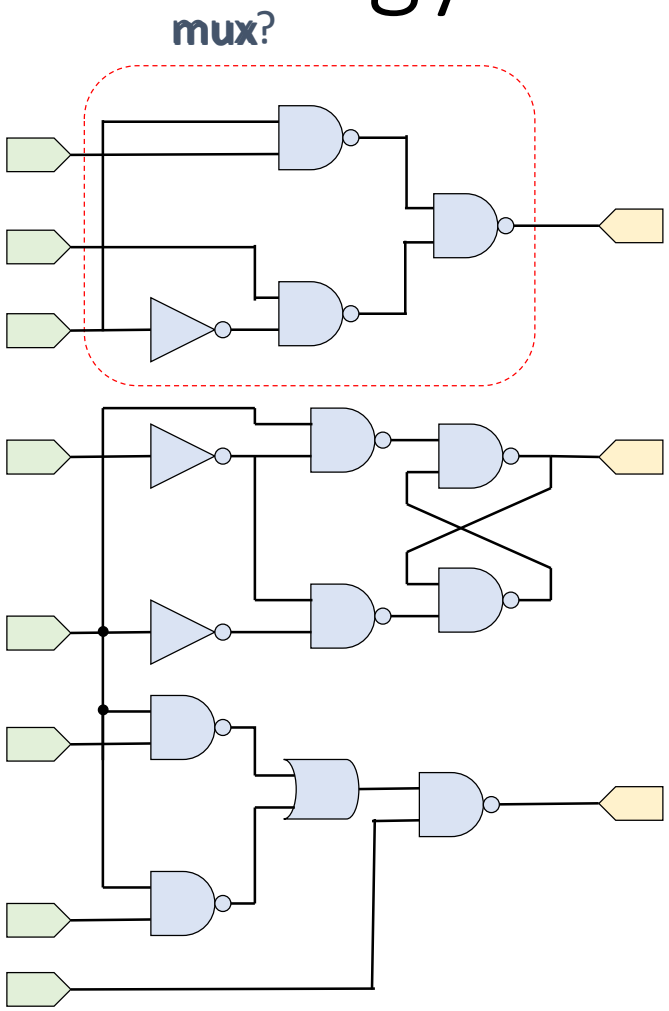
reverse engineering to extract high-level components from an unstructured and flat netlist

reverse engineering algorithm portfolio



[HOST'13, DATE'13, TETC'14, FMCAD' 14]

general strategy



Netlist is a sea of gates! No information about boundaries of modules inside it.

Identify Potential Module Boundaries

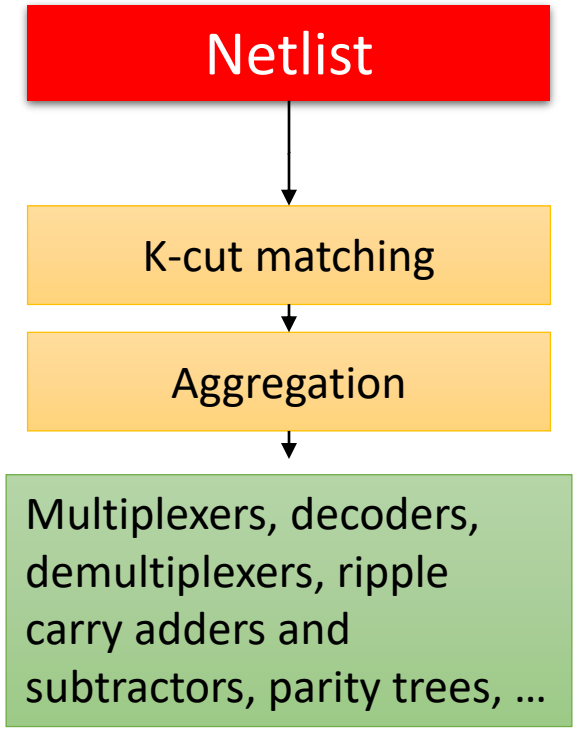
↓

BDD/SAT-Based Analyses to Verify Functionality

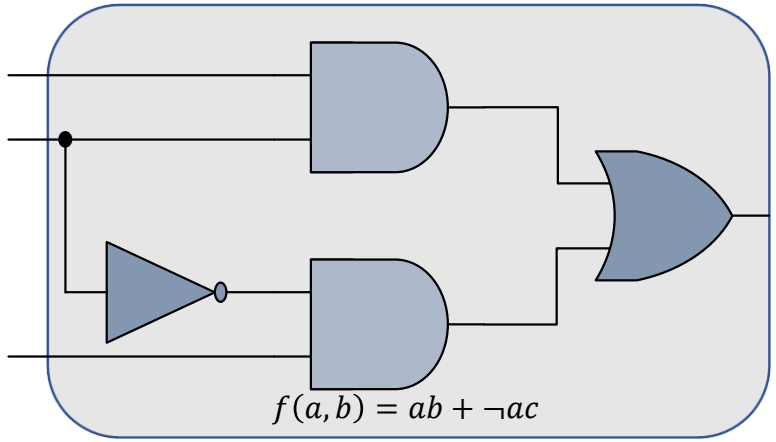
↓

Output Inferred Modules

bitslice identification and aggregation



bitslice Identification using cut matching



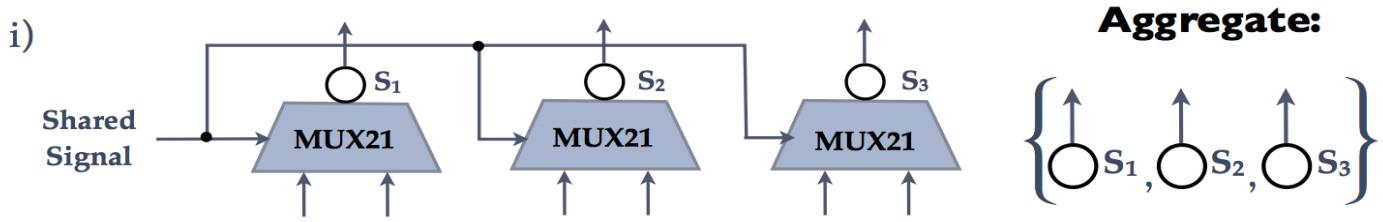
- cuts are computed recursively
- made tractable by enumerating cuts with k ≤ 6 inputs
- group cuts into equivalence classes using permutation independent comparison
- BDDs used to represent Boolean functions during matching

Cong and Ding, *FlowMap*, [TCAD'94]
Chatterjee et al., *Reducing Structural Bias in Technology Mapping*, [ICCAD'05]

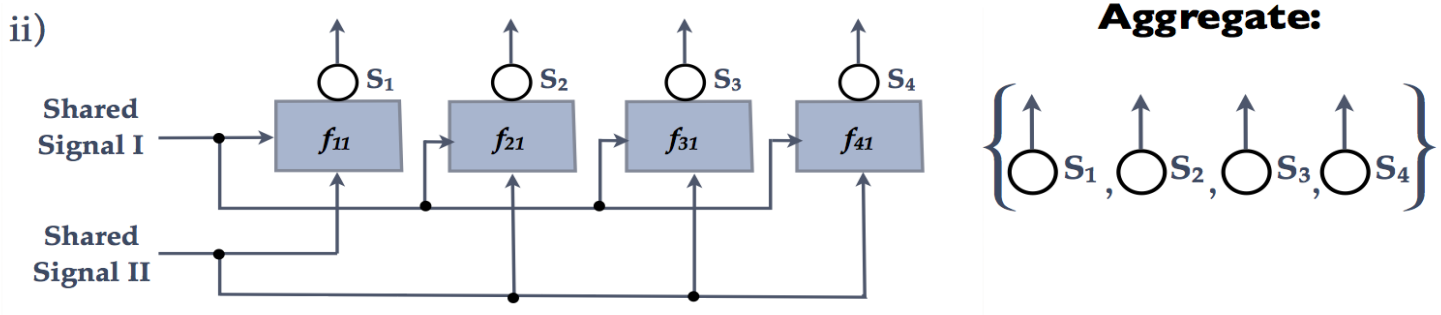
bitslice aggregation

Group Bitslices With Shared Signals

Multi-bit Multiplexer 2-to-1

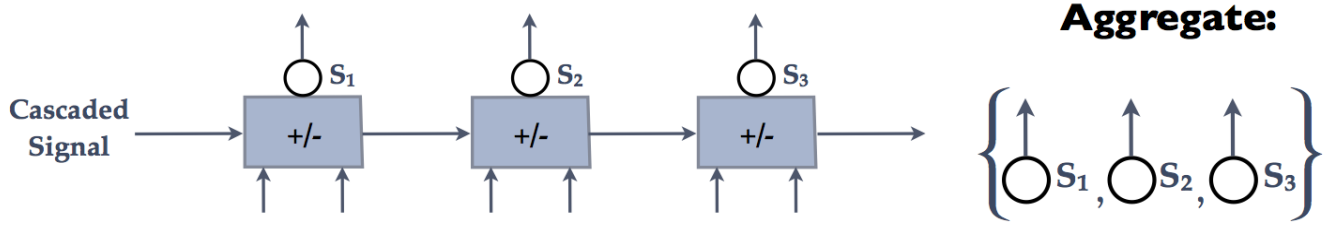


Decoder 2-to-4

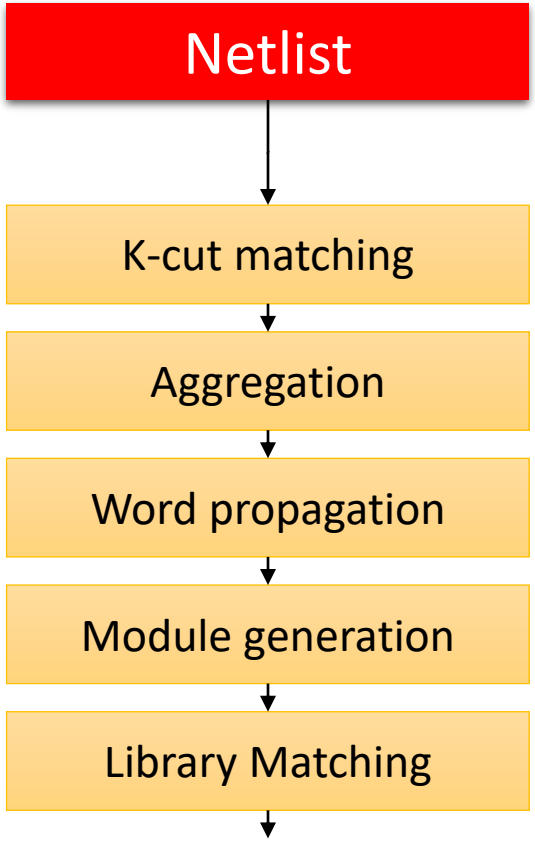


Group Bitslices With Cascading Signals

Ripple Carry Adder/Subtractor

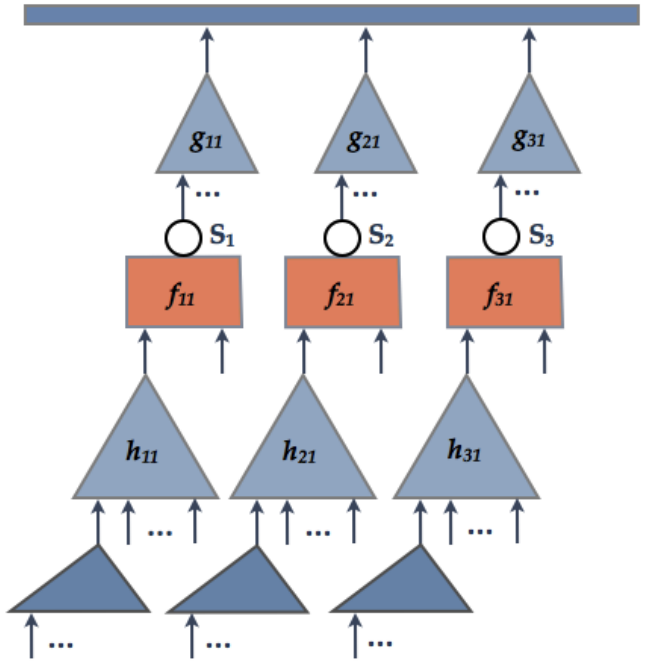


word propagation and module matching

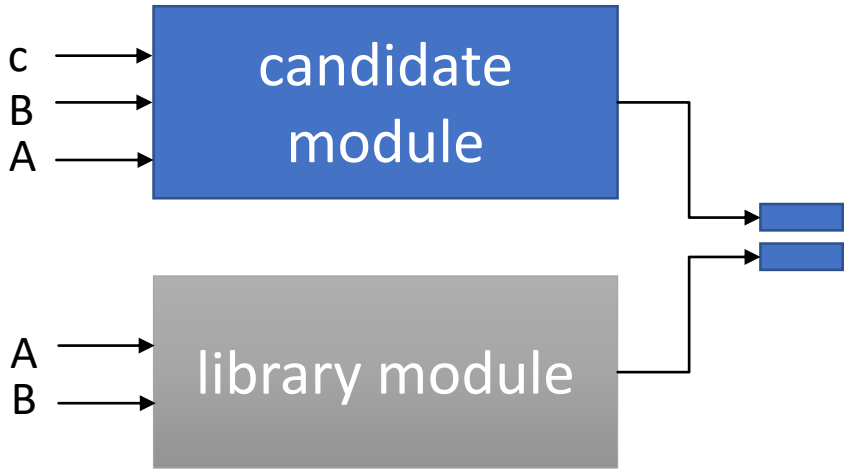


word propagation and module generation

given an “output” word, we can traverse backwards/forwards to closely-related words to find candidate modules



library matching



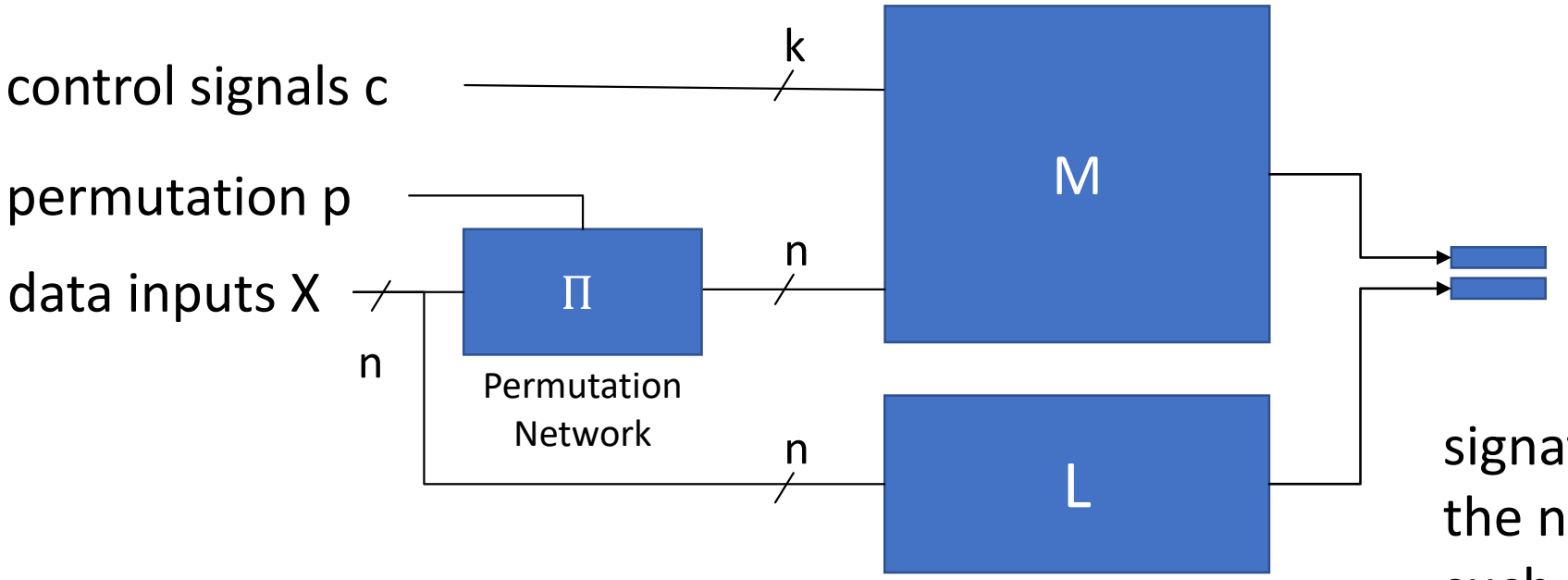
Want to match candidate modules against a library of common modules such as adders, ALUs, …

Challenges

- Permutation and polarity of inputs
- Setting of control inputs

- formulate as QBF problem
- use *signatures* to restrict space of permutations searched [FMCAD'14]

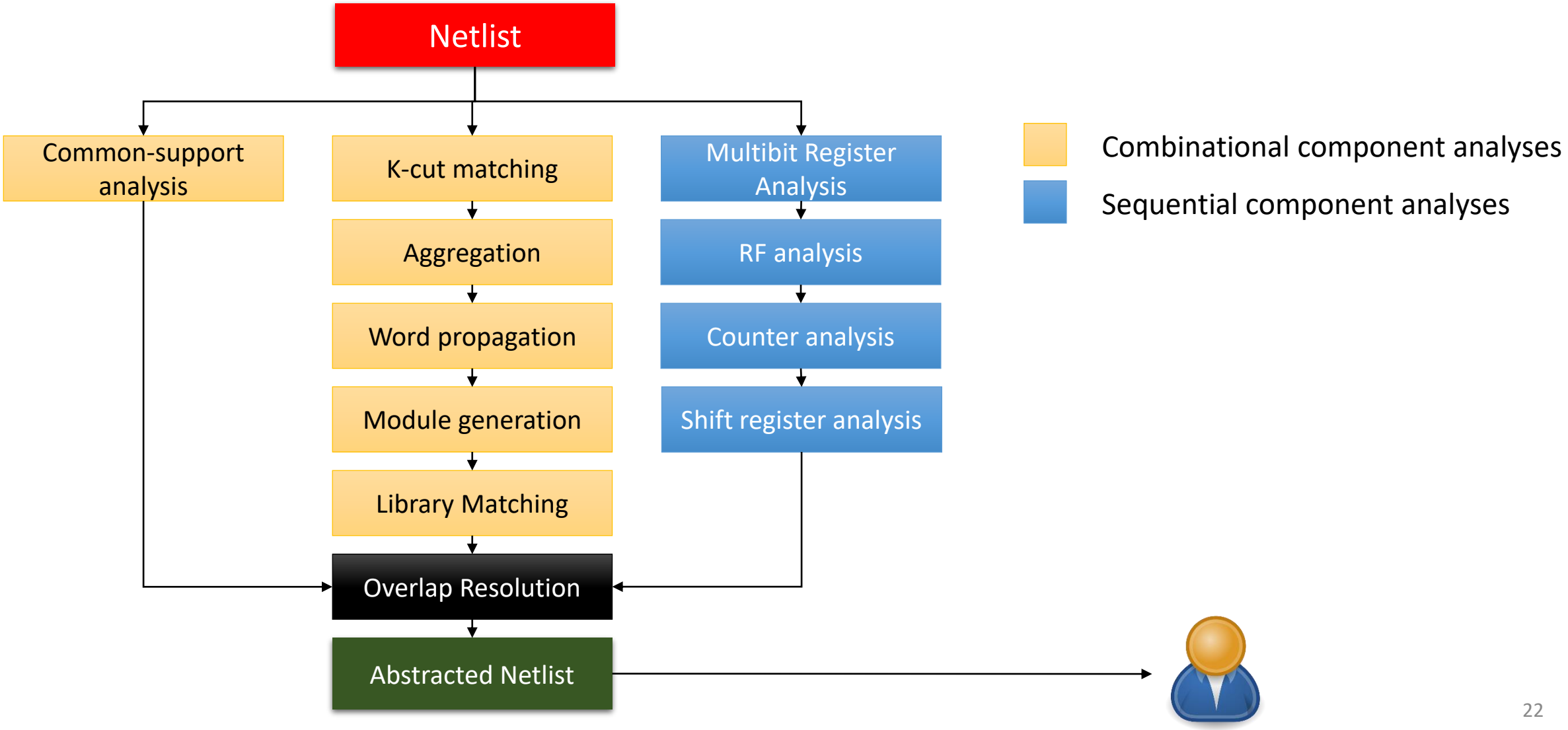
library matching as QBF



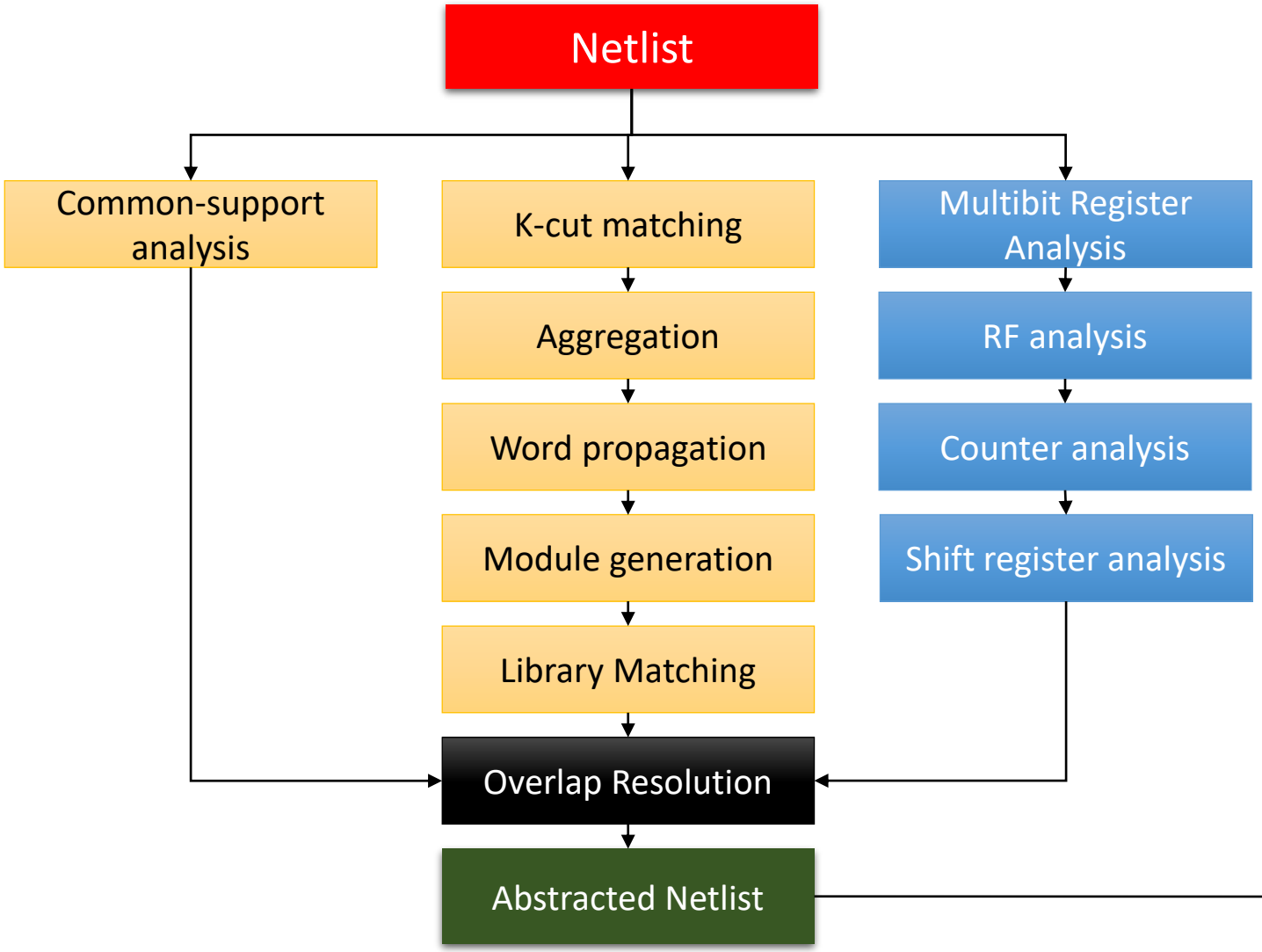
signature of an input *i* is the number of outputs *o* such that $o_i \neq o_{\neg i}$.

$$\exists c, p\ \forall X\colon\ M(\Pi(p, \mathrm{X}), \mathrm{c}) = \mathrm{L}(\mathrm{X})$$

there are many more algorithms here …

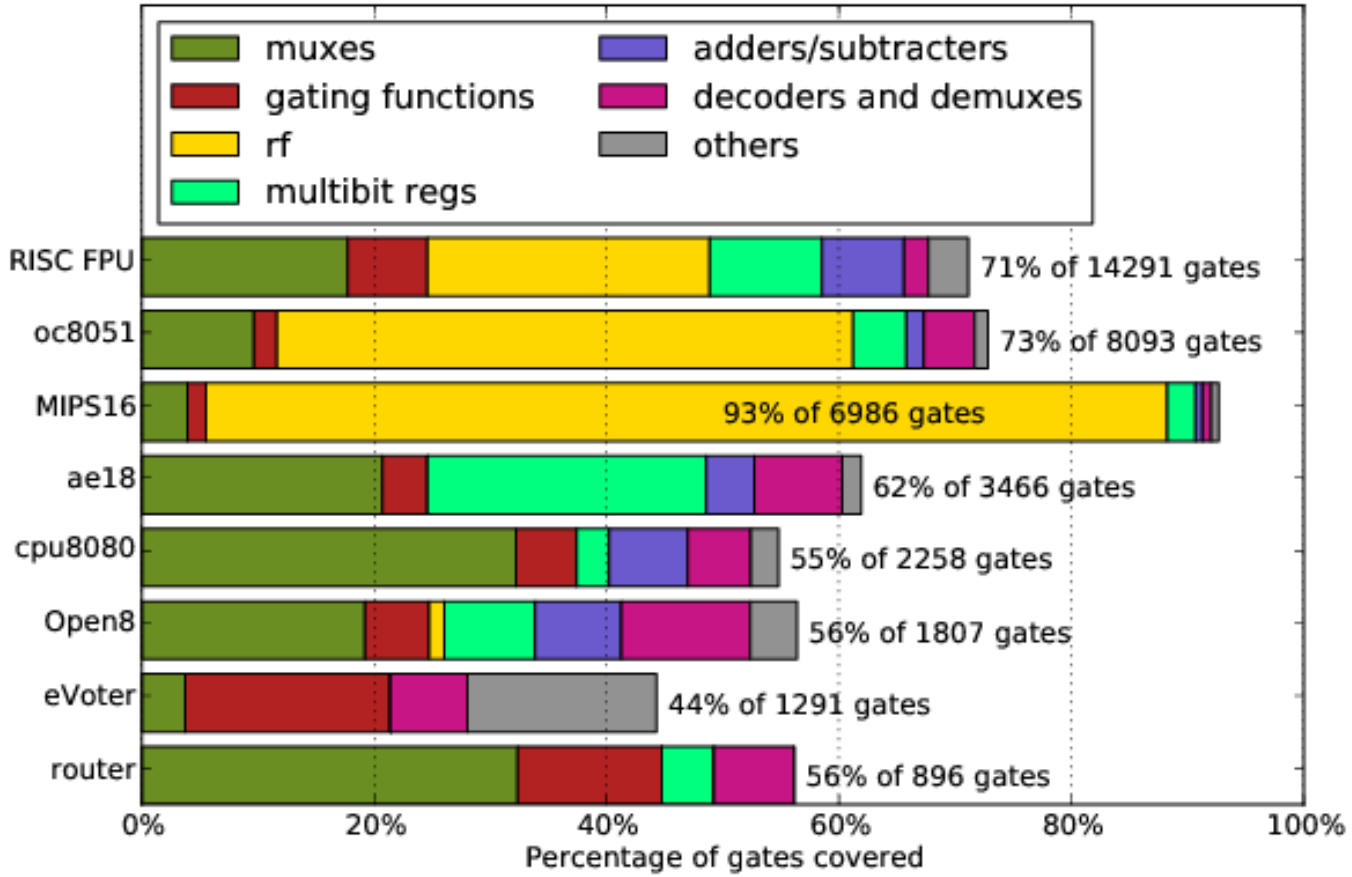


but the takeaway is:



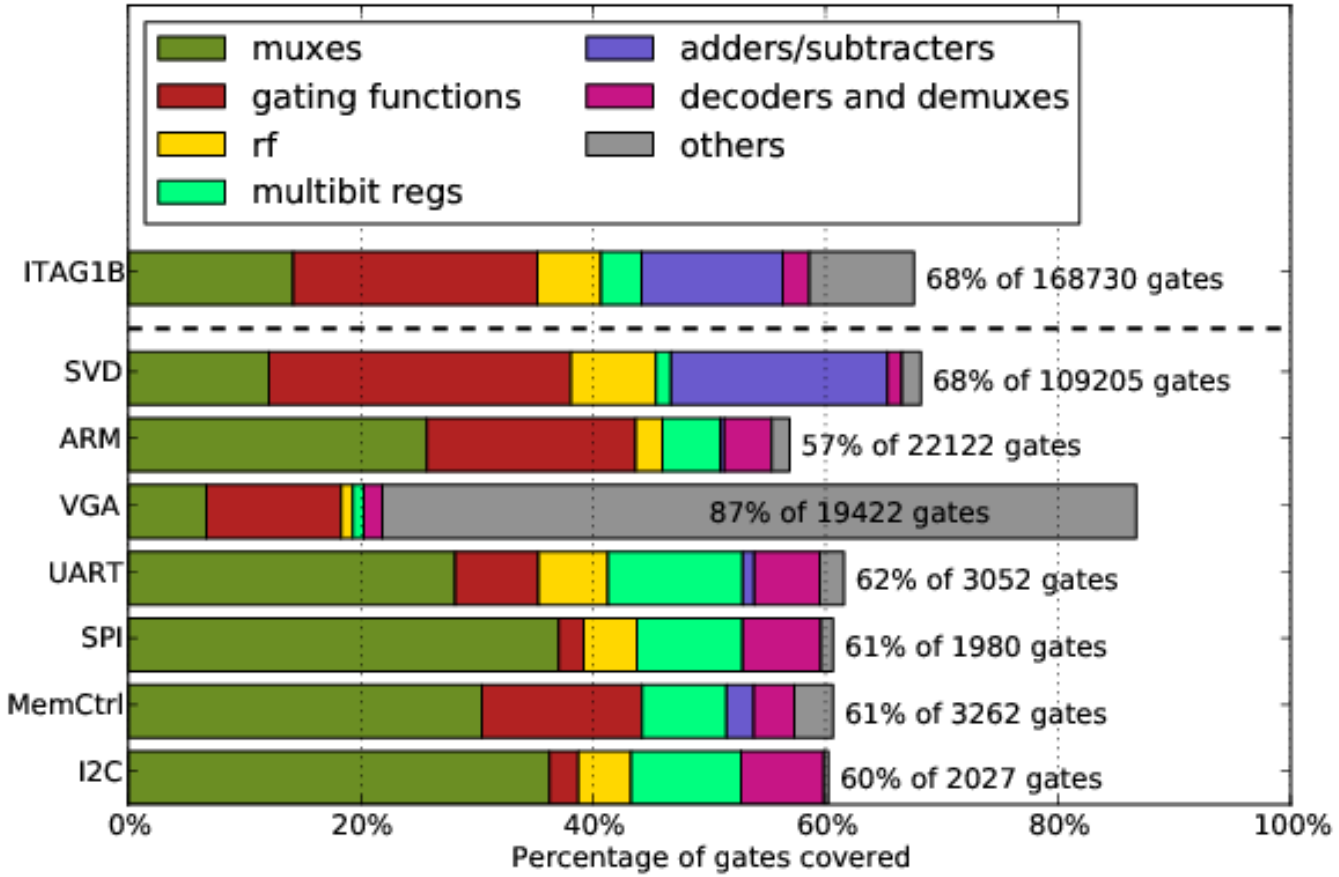
we have a reasonably effective portfolio of inference algorithms to identify word-level modules from a unstructured netlists

summarizing inference results (1/2)



- 50-90% of the gates in these are covered
- Runtime is a maximum of a several minutes

summarizing inference results (2/2)

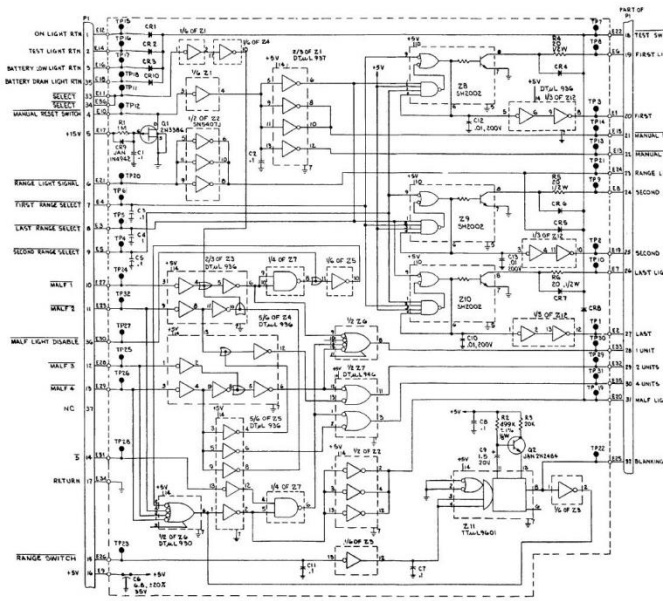


- Covered ~70% of the large test article (375k gates)
- Split the up big design into 7 subcomponents using reset tree; Covered 60-87%

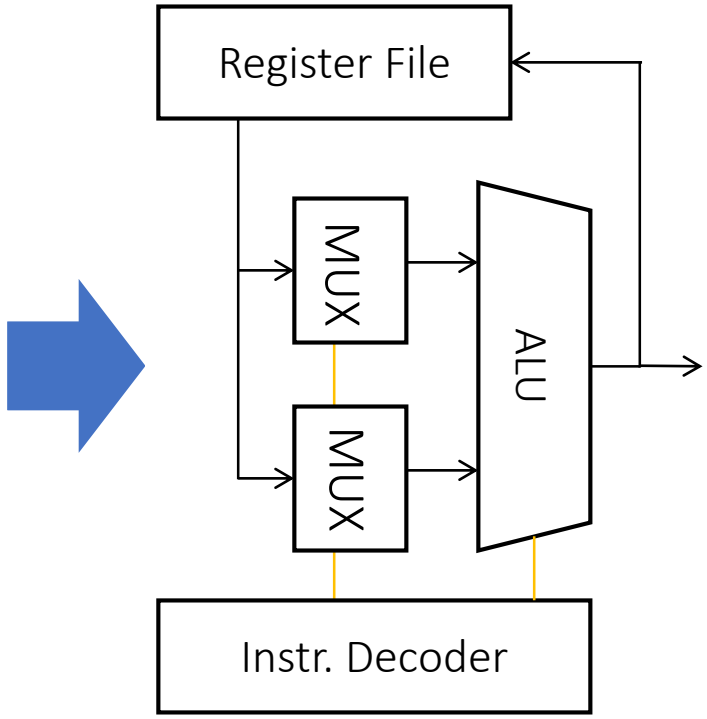
retrospective on reverse engineering

- **possible** to reconstruct many high-level structures from flat netlists
- easier if we have information about what we **expect to find**
  recall we were working blind – had no RTL info at all
- challenges are mostly due to **aggressive synthesis and optimization**
  - tools merge equivalent functionality across sub-modules
  - tools aggressively optimize out constants
- these challenges may be solvable with CEGIS

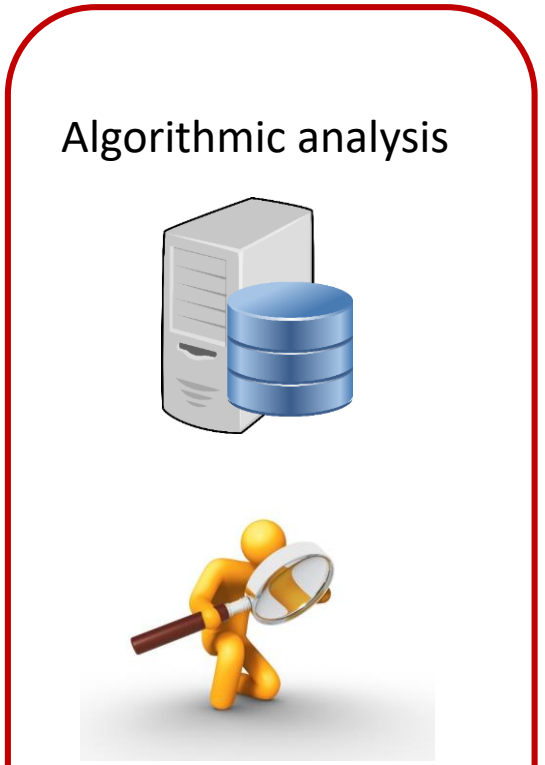
so we can do this …



Source: http://miscpartsmanuals2.tpub.com/TM-9-1240-369-34/TM-9-1240-369-340115.htm



but the elephant in the room is …



How does one analyze a design to find security vulnerabilities?

two views of hardware vulnerabilities

1. someone untrusted controls part of my design
   (can I somehow analyze and then trust this design?)
2. the design is trusted, but is it trustworthy?
   (what can I do to convince myself that the design's security behavior is what I intended)

outline: revisited

we should care about HW security!

can we reconstruct trust in an untrusted component? (bottom-up)

can we prove a trusted component is trustworthy? (top-down)

- techniques for security property specification
- case study of enclave platform verification

what's connection between the two?

need more than safety for security (1/4)

```
int secret[N];
int public[N];
int foo(int index) {
  int r = 0;
  if (index >= 0 && index < N)
    if (priv_level == sup_user)
      r = secret[index];
    else
      r = public[index];
  return r;
}
```

Specification: only super user must access secret array.

```
mov ebx, secret
mov edi, index
mov eax, [ebx+4*edi]
mov r, eax
```

Idea: instrument each load to ensure that secret array is not accessed

need more than safety for security (2/4)

```
define valid (addr) =
  (addr >= secret &&
   addr <  secret + N)
  ⇒ priv_level == sup_user)
```

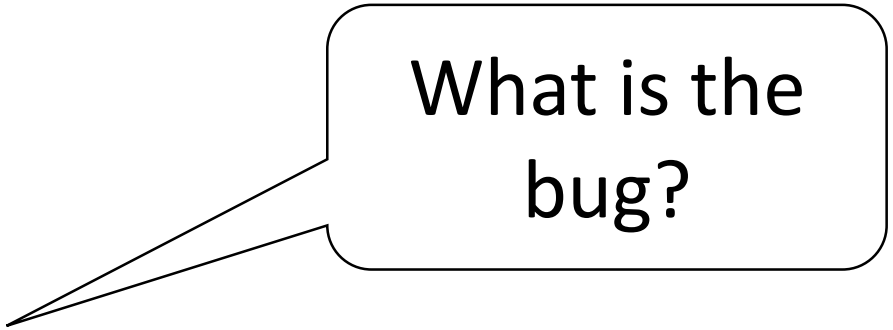
Specification: only super user must access secret array.

```
mov ebx, secret
mov edi, index
assert valid(ebx+4*edi);
mov eax, [ebx+4*edi]
mov r, eax
```

Idea: instrument each load to ensure that secret array is not accessed

need more than safety for security (3/4)

```
int secret[N]; // &secret[0]=100
int public[N]; // &public[0]=104
int t;         // &t=108; N=4
int foo(int index) {
  int r = 0;
  if (index >= 0 && index <= N)
    if (priv_level == sup_user)
      r=t=secret[index];
    else
      r = public[index];
  return r;
}
```



Specification: only super user must access secret array.

need more than safety for security (4/4)

```
int secret[N]; // &secret[0]=100
int public[N]; // &public[0]=104
int t;         // &t=108; N=4
int foo(int index) {
  int r = 0;
  if (index >= 0 && index <= N)
    if (priv_level == sup_user)
      r=t=secret[index];
    else
      r = public[index];
  return r;
}
```

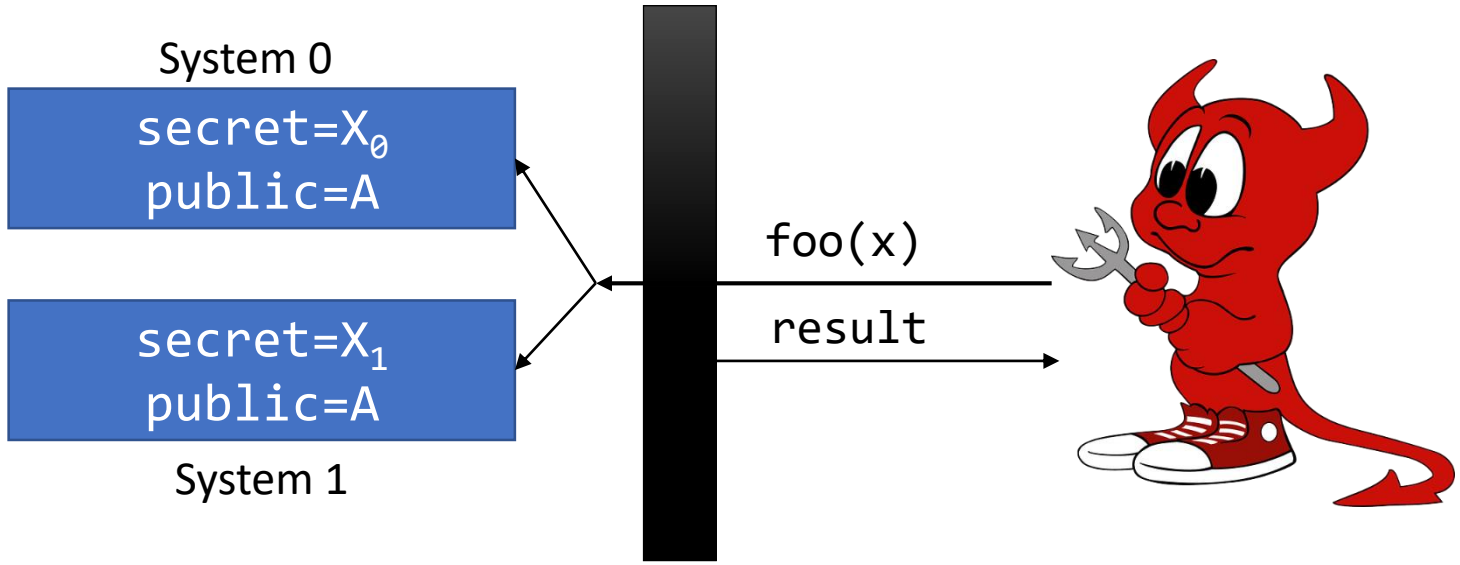
1. superuser calls foo(0)
2. t contains secret[0] (bug #1)
3. attacker calls foo(4)
4. this should have returned 0
5. due to bug #2 we return t
   (which contained secret[0])

Note our property is still satisfied!

moral of the story

- we wanted to verify the **flow of information**
- but what we were actually checking was **access** to the information
- trace (or safety) properties can capture the latter, not the former

cryptographers solution: distinguishability games



- Pose game between an attacker and defender
- Attacker wins game: system is **not secure**
- Defender wins the game: system **is secure**

distinguishability game for function foo

Defender

Choose $b \in \{0,1\}$

$\perp$

$public_b[1]$

...

$secret_0$, $secret_1$, public

foo(0), SU

foo(1), U

...

b = 1

Attacker

Game Initialization

- Defender chooses $b \in \{0,1\}$
- Attacker chooses $secret_0, secret_1, public$
- System initial state is defined by $secret_b, public$

Game Execution

- Attacker makes calls to foo(x), and observes result r
- Result defined to be $\perp$ when called in SU mode
- Result is the return value of foo in usr mode

Finalization: Attacker wins if she can determine b

can attacker win the game?

Defender

Choose
$b = 1$

⊥

7

8

…

$secret_{0}$={1,2,3,4},
$secret_{1}$={5,6,7,8},
public={9,7,8,6}

foo(0), SU

foo(1), U

foo(2), U

…

b = ??

Attacker

```
int secret[N];
int public[N];
int foo(int index) {
  int r = 0;
  if (index >= 0 && index < N)
    if (priv_level == sup_user)
      r = secret[index];
    else
      r = public[index];
  return r;
}
```

The game can go on forever, nothing will reveal secret to adversary

can attacker win game with buggy foo?

Defender

Choose $b = 1$

⊥ (t = 5)

7

5 (val in t)

…

$secret_0$={1,2,3,4},
$secret_1$={5,6,7,8},
public={9,7,8,6}

foo(0), SU

foo(1), U

foo(4), U

…

b = 1

Attacker

```
int secret[N]; //&secret[0]=100
int public[N]; //&public[0]=104
int t = 0; //&t=108; N=4
int foo(int index) {
  int r = 0;
  if (index >= 0 && index <= N)
    if (priv_level == sup_user)
      r=t=secret[index];
    else
      r=public[index];
  return r;
}
```

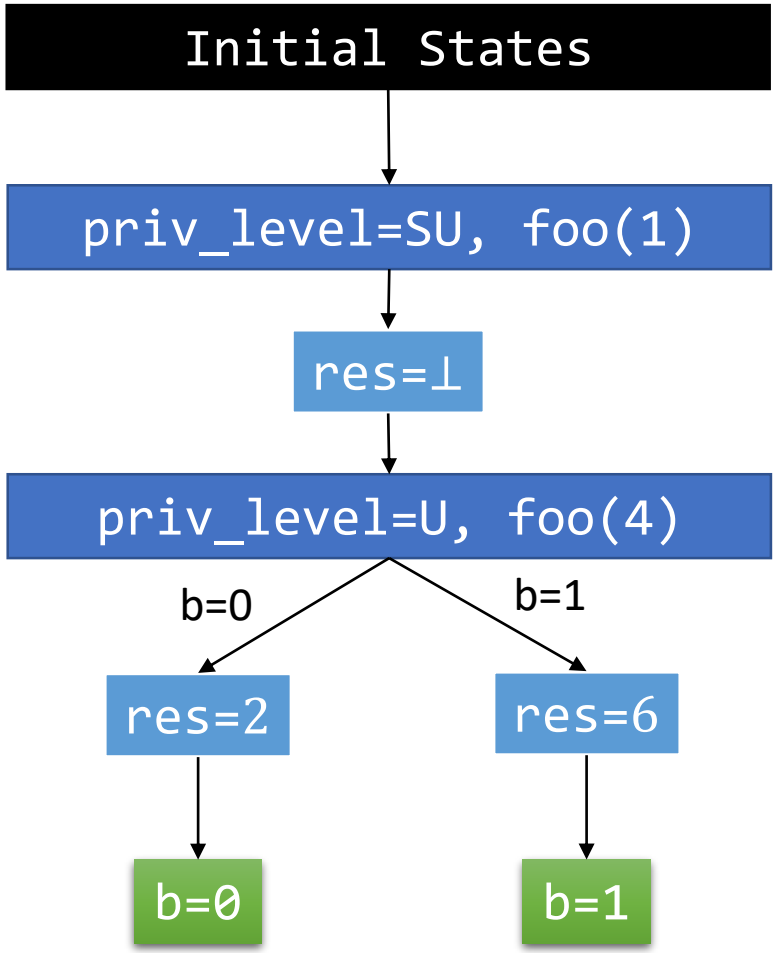
- Now, attacker always wins!
- Secrets are **not indistinguishable**

game viewed as a sequence of states

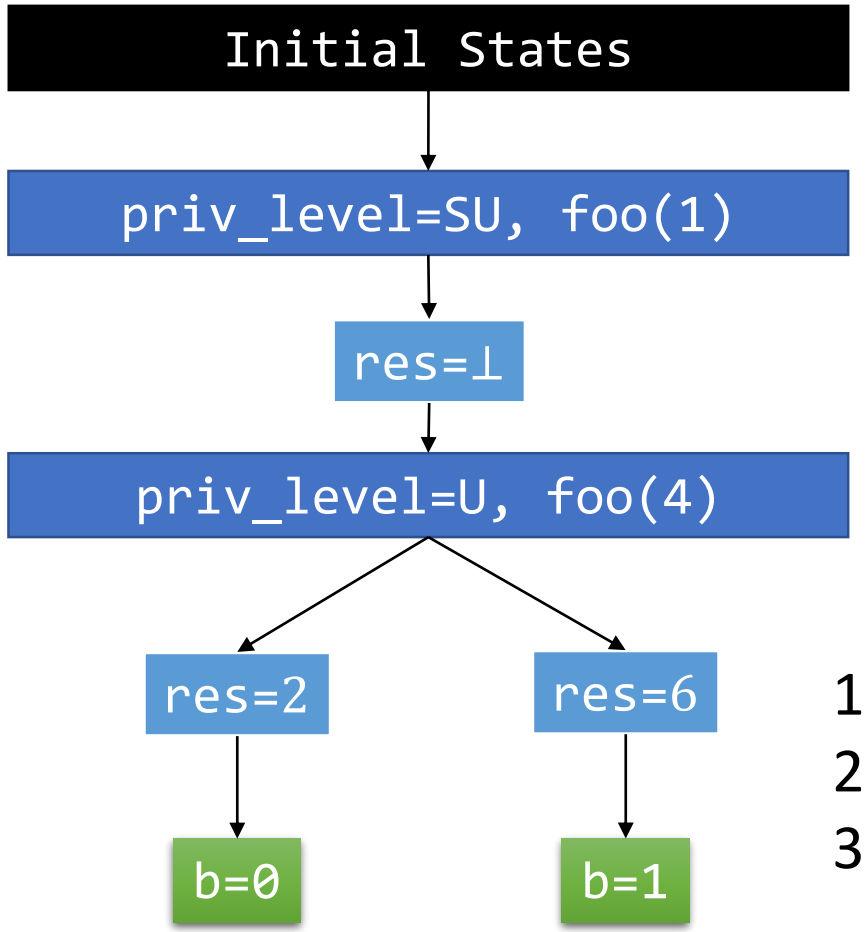
```
int secret[N];
int public[N];
int t = 0;
int foo(int index) {
  int r = 0;
  if (index >= 0 && index <= N)
    if (priv_level == sup_user)
      r=t=secret[index];
    else
      r=public[index];
  return r;
}
```

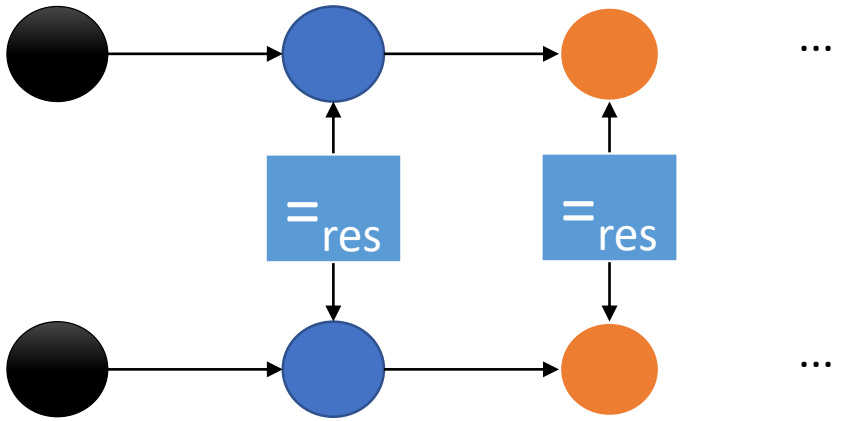
Initial states

- $secret_0$ = {1,2,3,4}
- $secret_1$ = {5,6,7,8}
- public = {10,11,12,13}



we can view this as a 2-trace property (1/2)

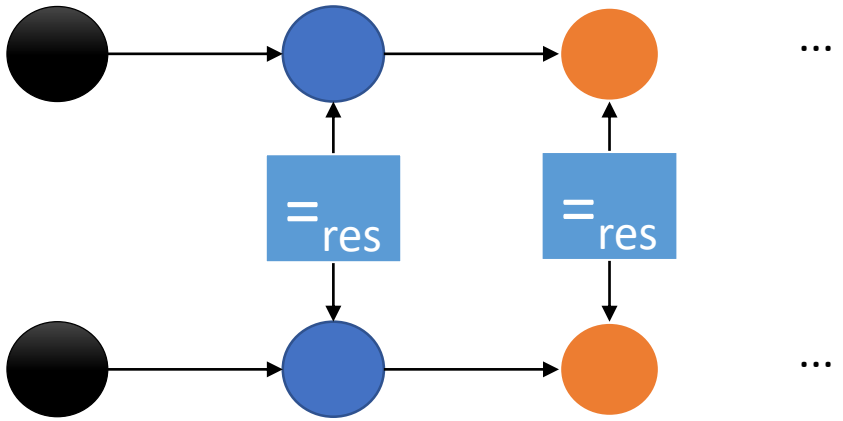




1. init states have different secrets, same public values
2. attacker takes arbitrary actions at each step
3. must prove results of invocation are same at each step

we can view this as a 2-trace property (2/2)

1. Init states have different secrets, same public values
2. Attacker takes arbitrary actions at each step
3. Must prove results of invocation are same at each step



$$\forall \pi_1 \pi_2 \in Traces:$$
$$\pi_1[0].public = \pi_2[0].public \Rightarrow$$
$$\forall i. res(\pi_1[i]) = res(\pi_2[i])$$

there is a well-developed theory here

- This was an example of a hyperproperty [Clarkson and Schneider, 2009]
- Was a relation over traces, not a set of traces
- Studied since at least the early eighties
  - Non-interference [Goguen and Meseguer, 1982]
  - Observational Determinism [Zdancewic and Myers, 2003]
  - Many more classes, with many subtle (but important) differences

Hyperproperties enable succinct system-level security specification

outline: revisited

we should care about HW security!

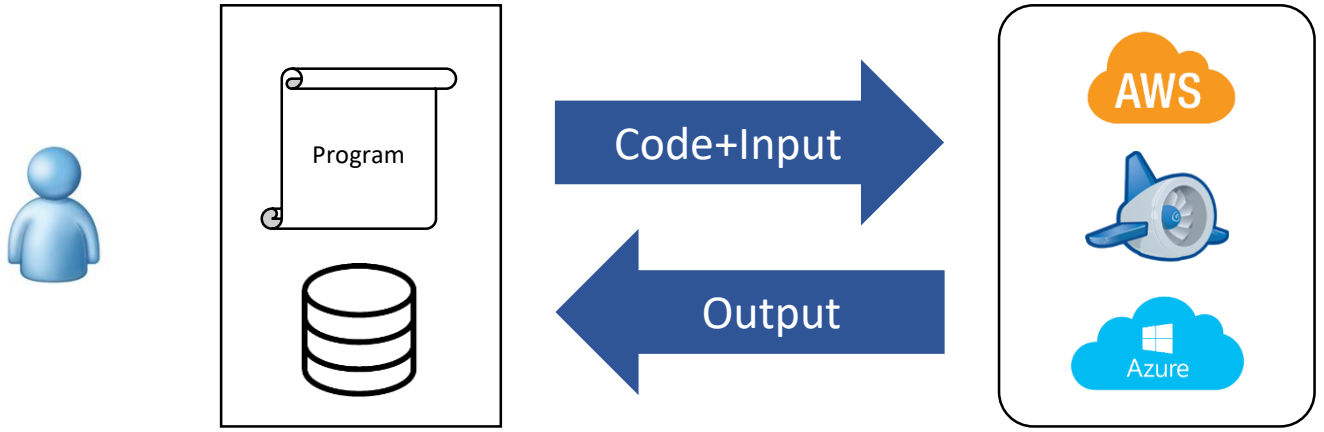
can we reconstruct trust in an untrusted component? (bottom-up)

can we prove a trusted component is trustworthy? (top-down)

- techniques for security property specification
- case study of enclave platform verification

what's connection between the two?

enclave platforms 101 (1/3)

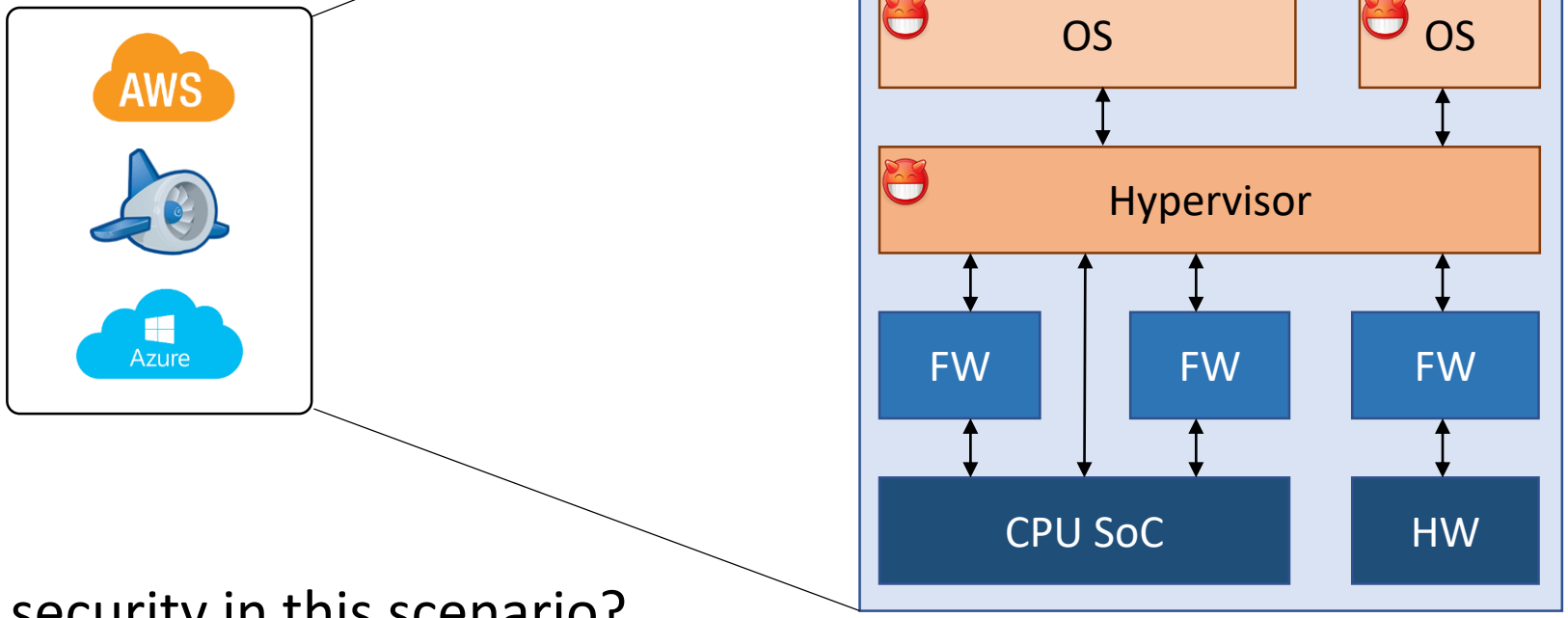


What guarantees do we want for **secure remote execution**?

[CCS'17]

enclave platforms 101 (2/3)

Typical public cloud implies untrusted OS, hypervisor and co-located apps



How to ensure security in this scenario?

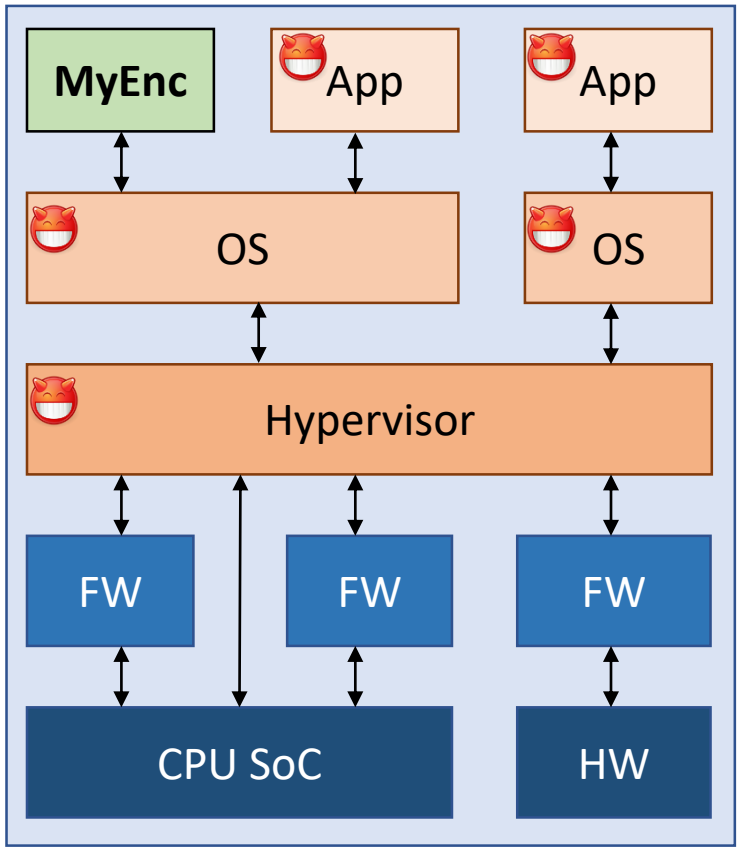
enclave platforms 101 (3/3)

Enclave Platforms like Intel SGX and MIT Sanctum provide ISA-level primitives for “secure” remote execution

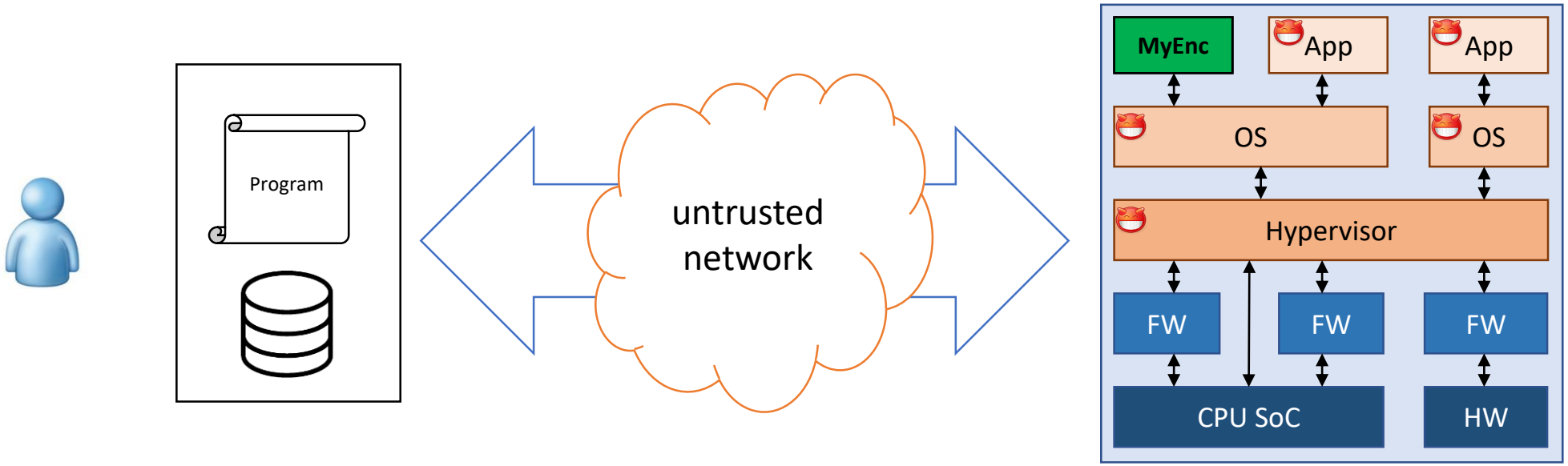
Enclave consists of:

- Protected mem region for code/data
- Even OS/VM can’t access enclave mem
- HW “measurement” operator

Research Question: How do enclave primitives translate to secure remote execution?



using enclaves



1. User sends program
2. Untrusted OS creates enclave
3. User authenticates w/ measurement
4. If measurement matches, encrypted input is set to enclave
5. Enclave computes returne encrypted output

how could platforms betray us?

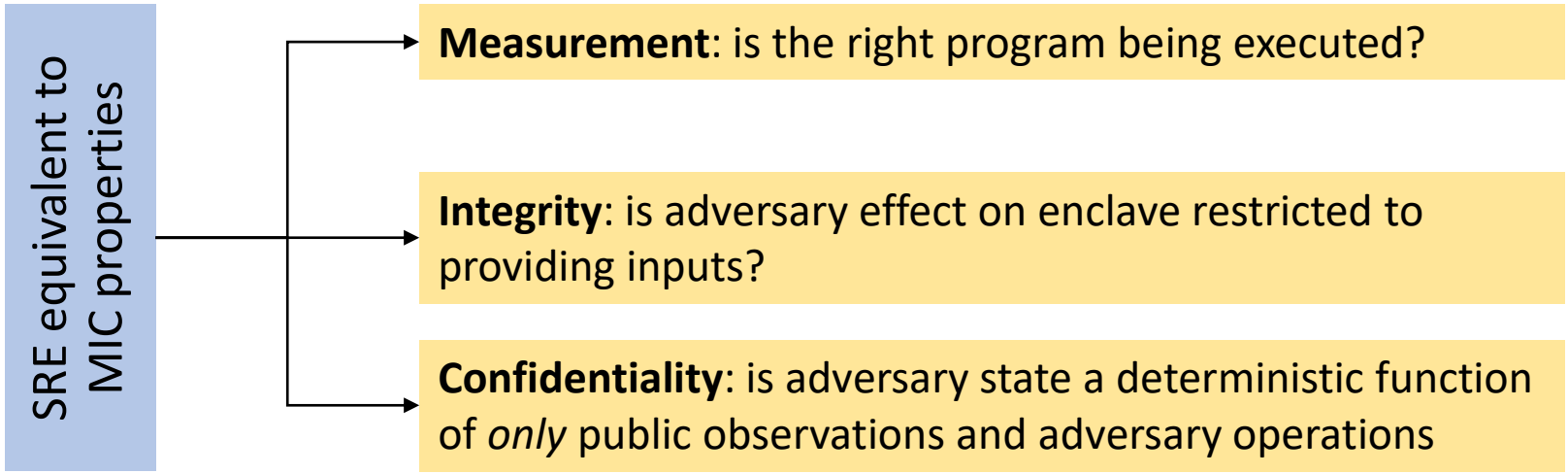
- Access control: untrusted code is able to read/write enclave memory
- Side channels: caches, page tables, speculation, etc.
- Measurements: different enclaves with the same measurement
- Bad operations like cloning of enclaves

Secure Remote Execution of Enclaves

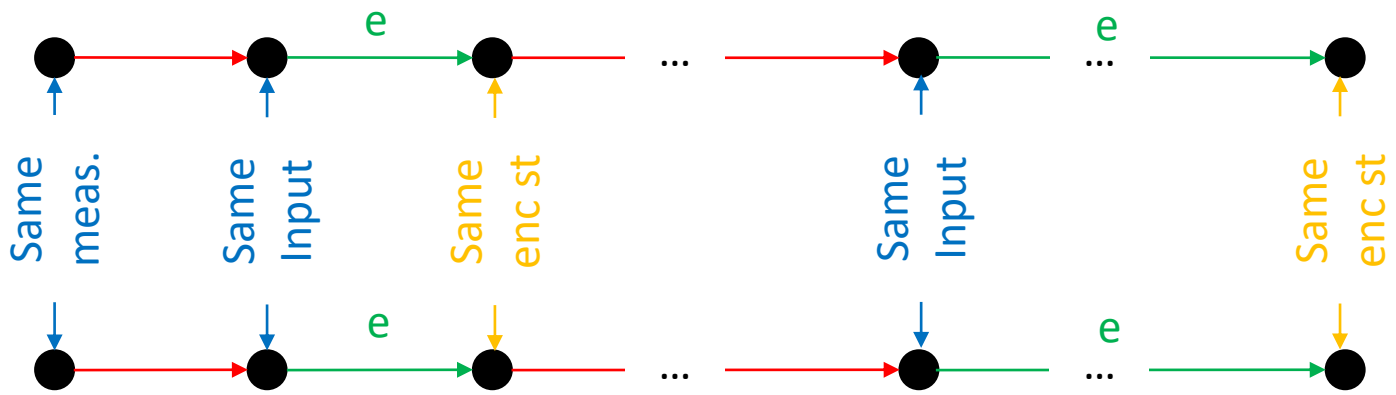
(Definition) Secure Remote Execution (SRE)

Remote server's execution of enclave program *e* must be identical to trusted local execution modulo inputs provided to the enclave.

(Theorem) SRE Decomposition

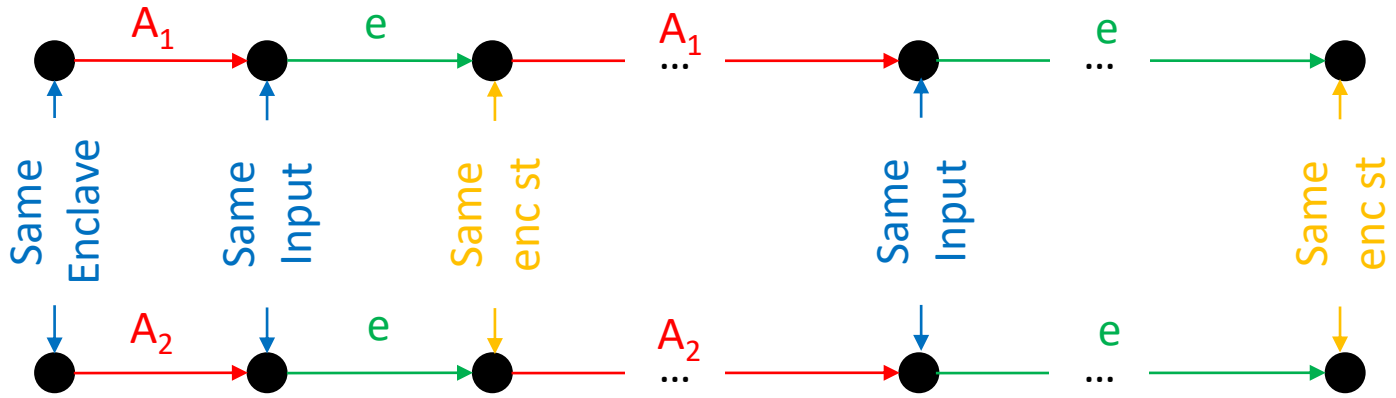


Enclave Measurement Property



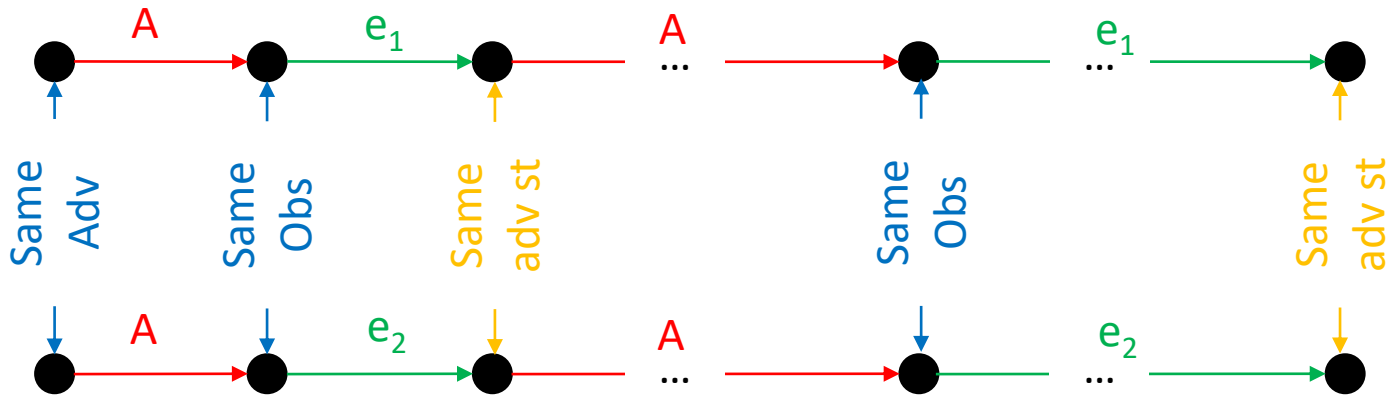
If we start two enclaves with the same measurement and, and if at each step the enclave executes, the inputs to the enclave are identical, then these two enclaves must have the same state and outputs at every step

Enclave Integrity Property



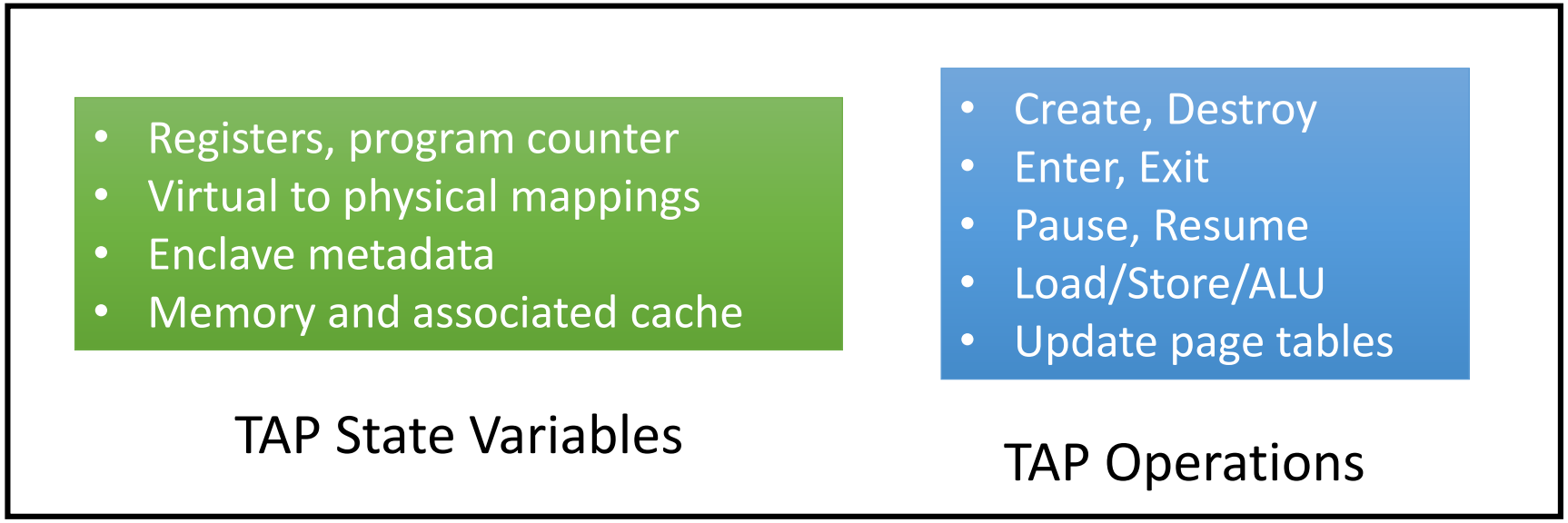
If we start two identical enclaves, and if at each step the enclave executes, the inputs to the enclave are identical, then regardless of adversary actions, the enclave computation must also be identical

Enclave Confidentiality Property



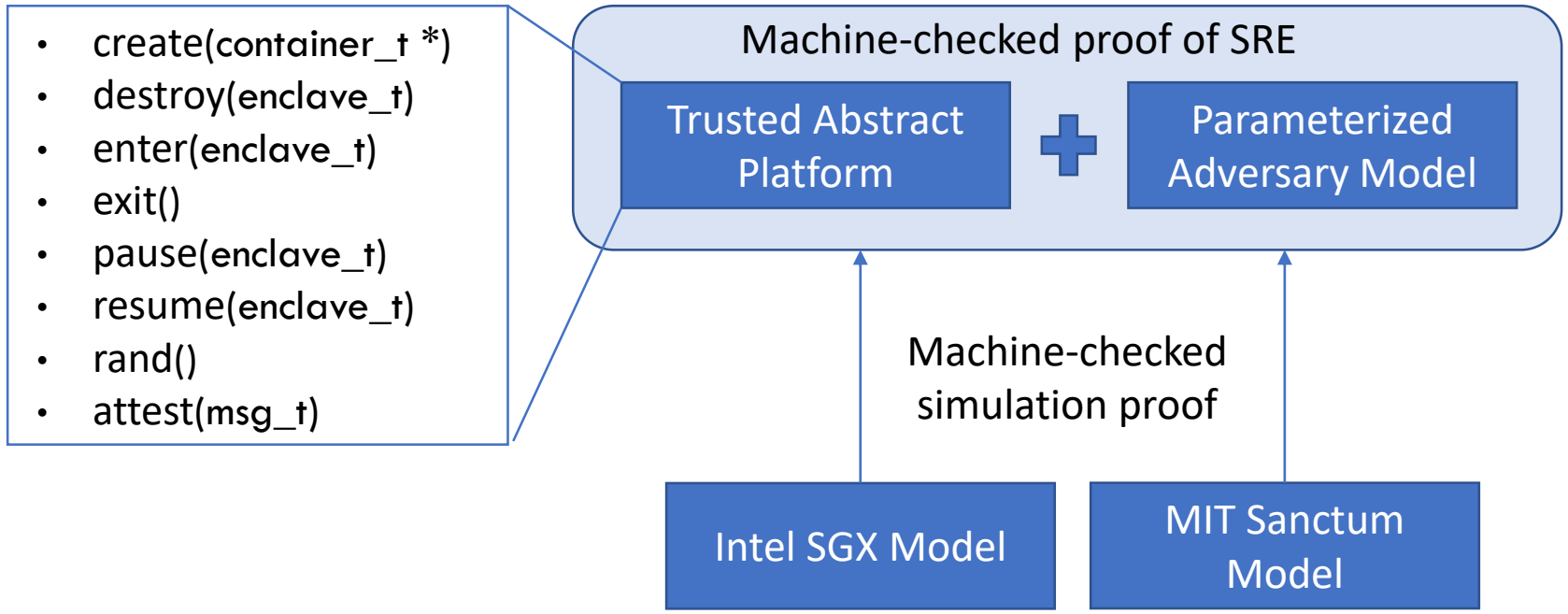
If we start two different enclaves, and if at each step the enclave executes, the outputs of the enclave are identical, and the adversary actions are the same, then regardless of adversary actions, the adversary state must also be identical

The Trusted Abstract Platform



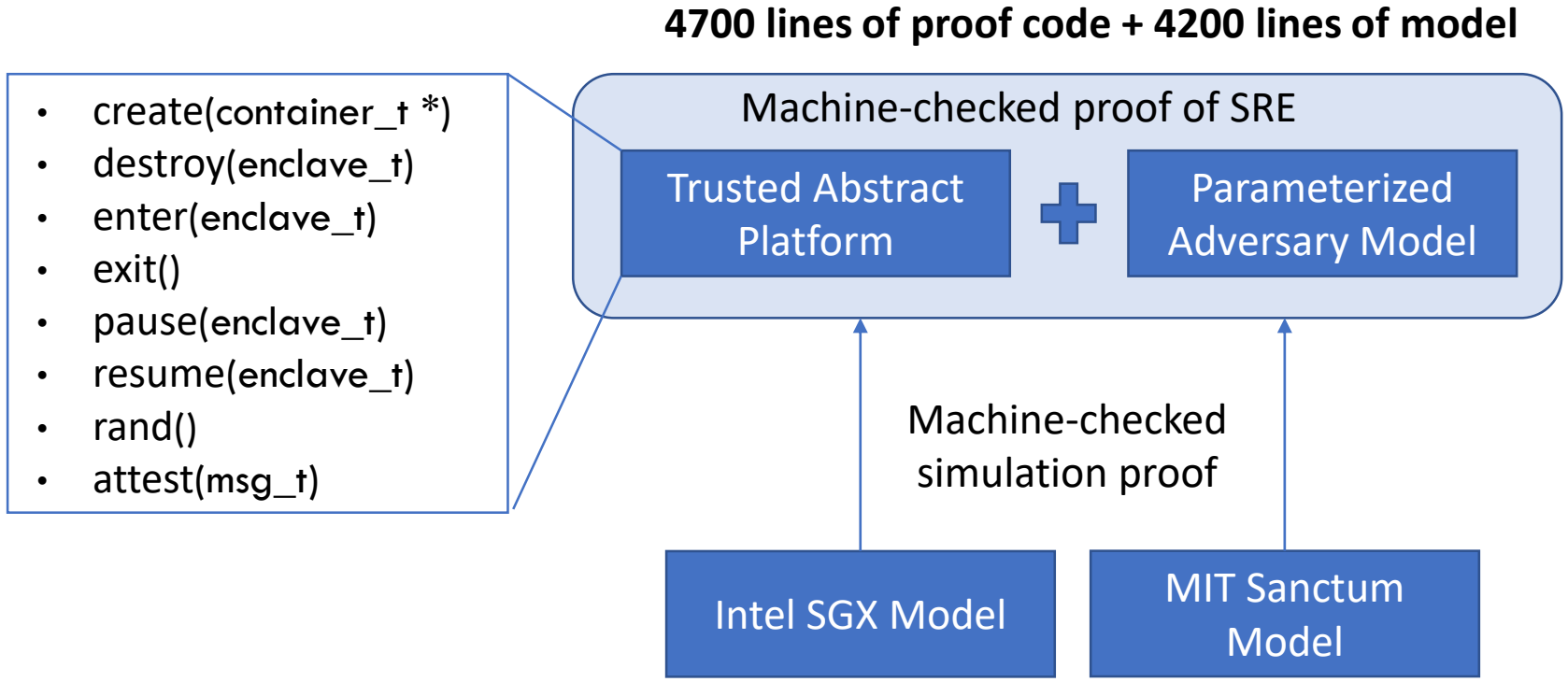
A model of an enclave platform that is not specific to a particular implementation like SGX, or Sanctum

Verification of Enclave Platforms



The TAP provides a common framework for reasoning about security of enclave platform primitives and adversary models.

Verification of Enclave Platforms



- Proofs were done by induction, just like this tutorial
- Two safety properties proven using self-composition

what is the point of all this?

- enclave platforms are quite complex
  SGX developer guide is 381 pages
  and guide already assumes reader understands enclave security guarantees
- you might think their security specification would be complex too
- but no! only 3 hyperproperties capture enclave security guarantees

moral of the story: security specification can be a lot more succinct (and easier to verify) than a specification of full functional correctness

outline: revisited

we should care about HW security!

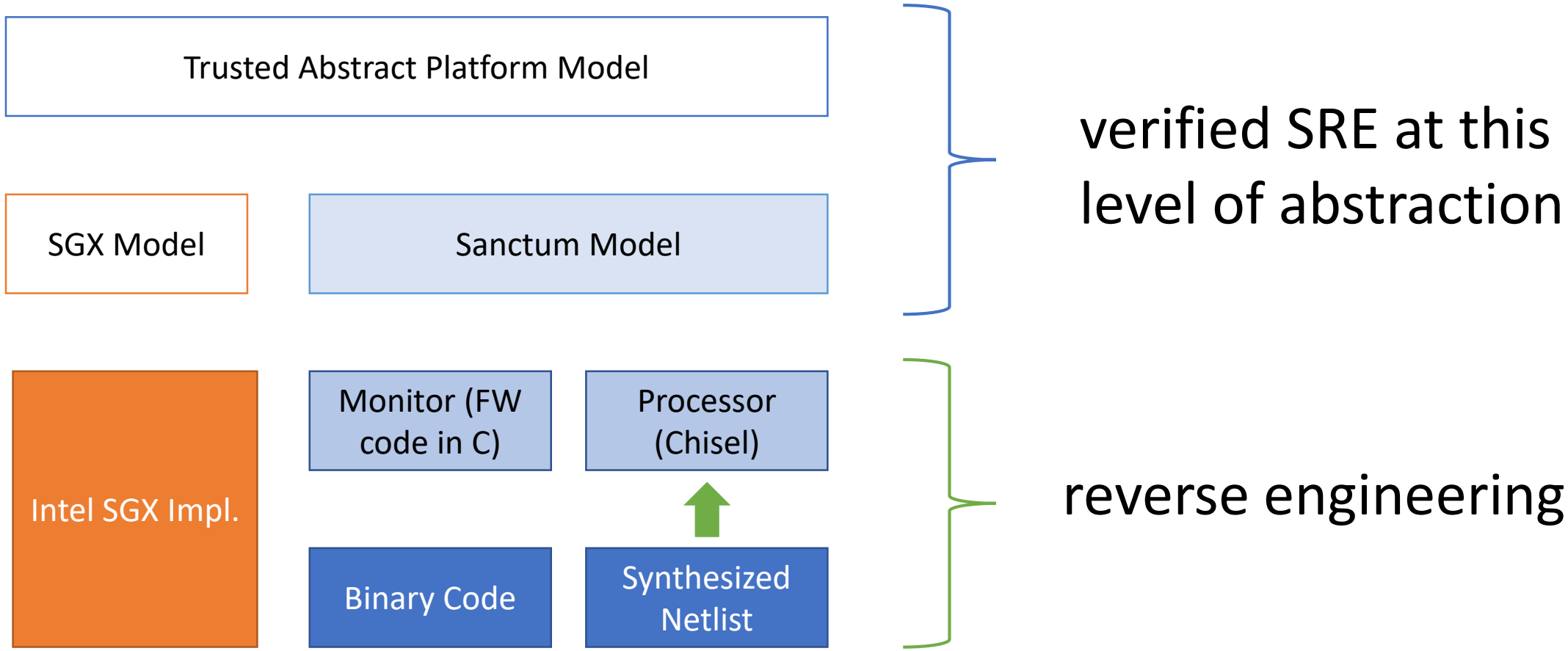
can we reconstruct trust in an untrusted component? (bottom-up)

can we prove a trusted component is trustworthy? (top-down)

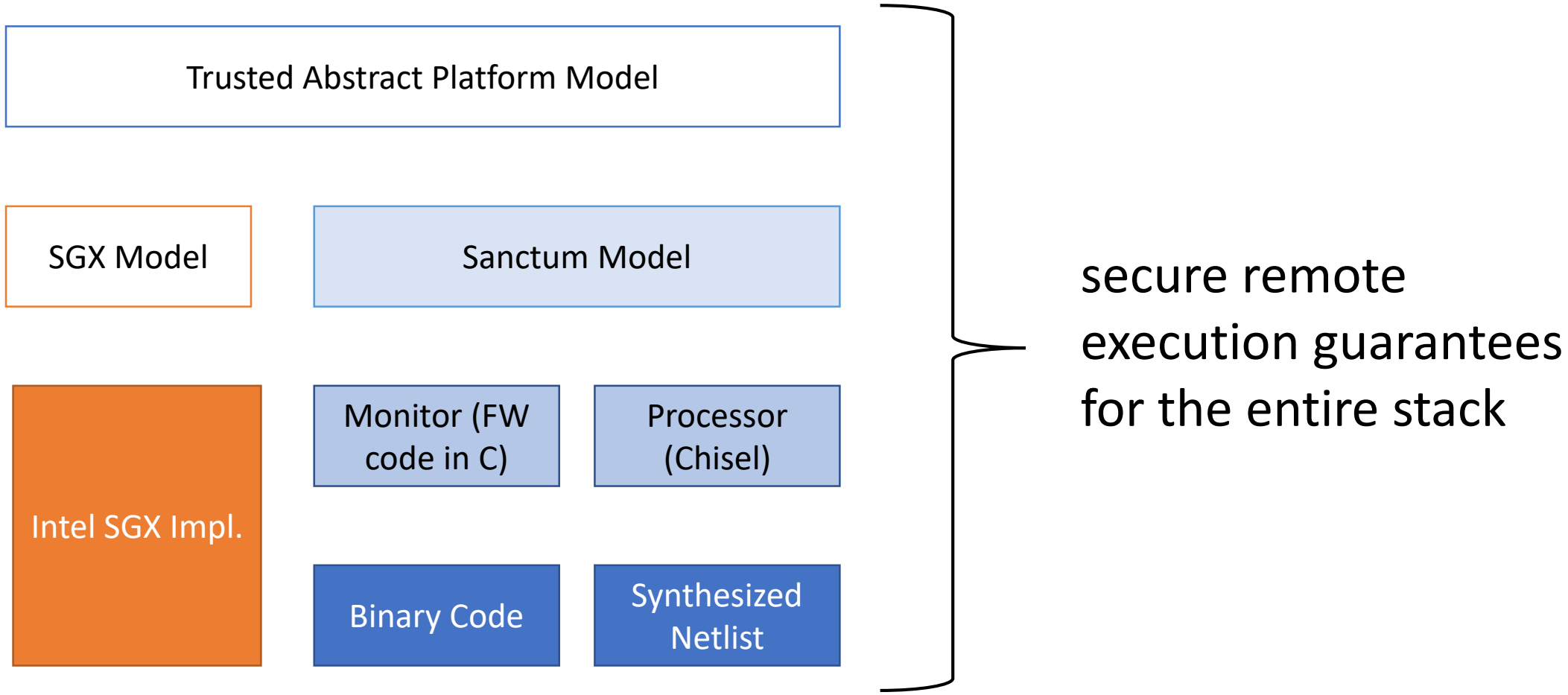
- techniques for security property specification
- case study of enclave platform verification

what's connection between the two?

where are we now?



where we would like to be



what makes this difficult?

microarchitectural impl and RTL synthesis pose new verif. challenges

- new attacks become possible at the lower layers (e.g. spectre)
- system invariants may be messed up due to retiming etc.

therefore, proofs cannot be straightforwardly translated to lower levels

this is where reverse engineering could help bridge the gap

lot of important research problems here

- can we do hyperproperty mining for finding security properties?
- software compilation is known to introduce security bugs
  - does synthesis also do the same?
  - if so, what are they and how do we fix them?
  - if not, why not and can we use those tricks in software?
- the µarch/HW/FW/SW interfaces seem to be a rich source of bugs
  - e.g., spectre, sysret, rowhammer, etc.
  - what can we do to eliminate these bugs?

more broadly, hyperproperties are under-explored

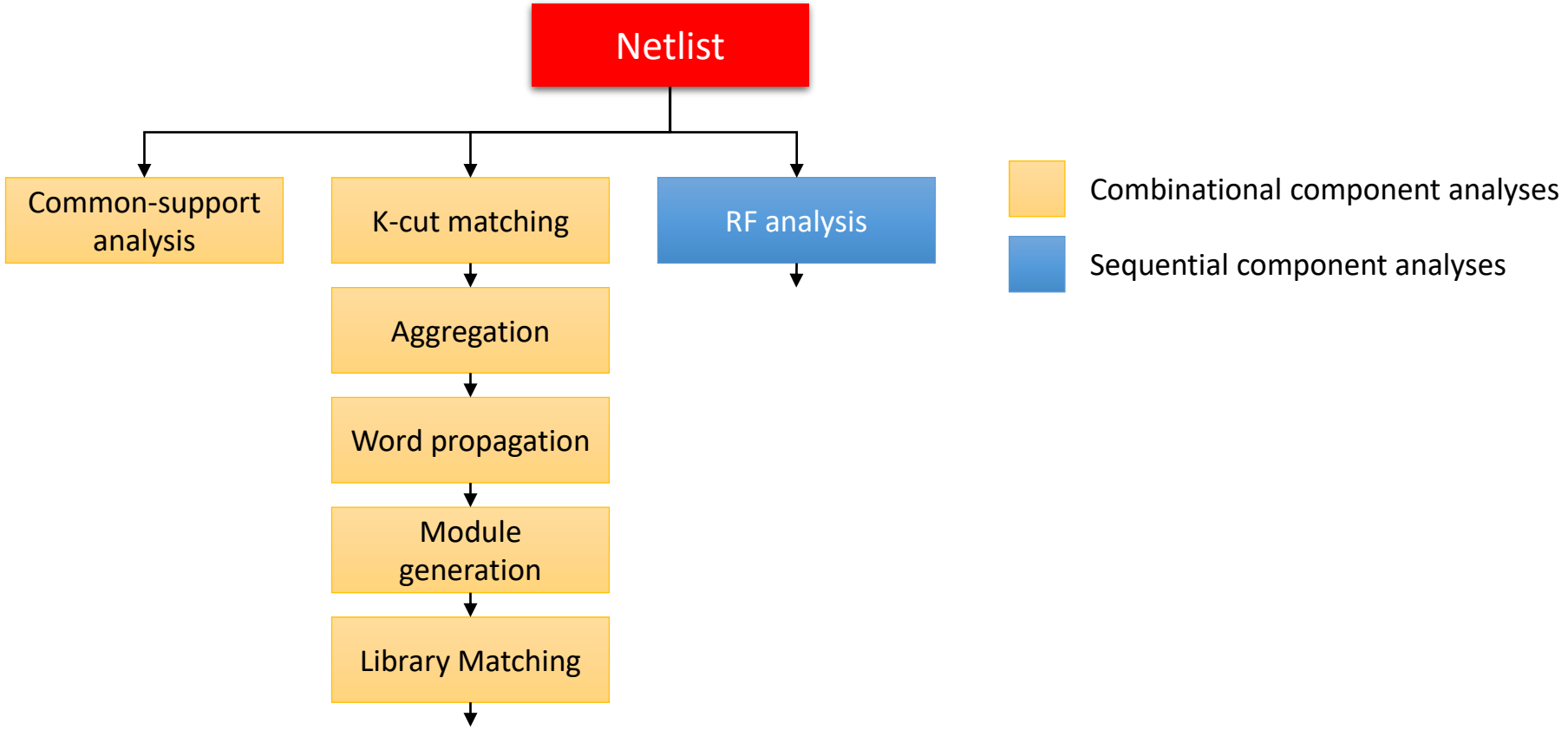
- they capture properties like determinism, commutativity, associativity
  - could be important in specification of system-level behavior
- in security, a key challenge is formulating the right property
  - hyperproperties can help here
- verification techniques are also relatively under-studied
  - relational invariants seem to be harder for IC3 etc.

references

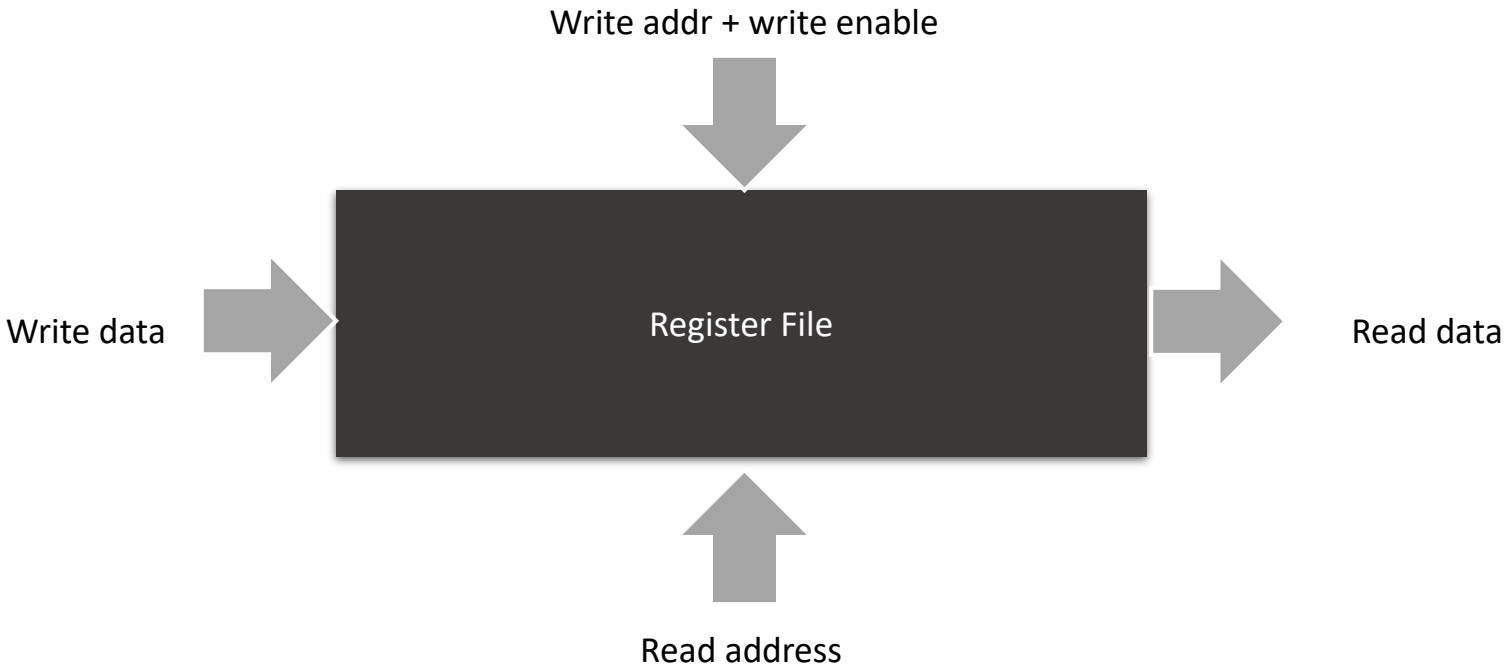
- Goguen and Meseguer, Security Policies and Security Models, SP ‘82
- Cong and Ding, FlowMap, TCAD ’94
- Zdancewic and Myers, Observational Determinism for Concurrent Program Security, CSFW ’04
- Terauchi and Aiken, Secure Information Flow as a Safety Problem, SAS ‘05
- Chatterjee, Mischenko, Brayton, Reducing Structural Bias in Technology Mapping, TCAD ’06
- Li et al., WordRev: Finding word-level structures in a sea of bit-level gates, HOST ‘13
- Subramanyan et al., Reverse Engineering Digitial Circuits using Functional Analyses, DATE ‘13
- Subramanyan et al., Reverse Engineering Digital Circuits using Structural and Functional Analyses, TETC ’14
- Gascon et al., Template-based circuit understanding, FMCAD ’14
- Subramanyan et al., A Formal Foundation for Secure Remote Execution of Enclaves, CCS ’17
- Yang et al., Lazy Self-Composition for Security Verification, CAV ’18

Thank You

Identifying Register Files



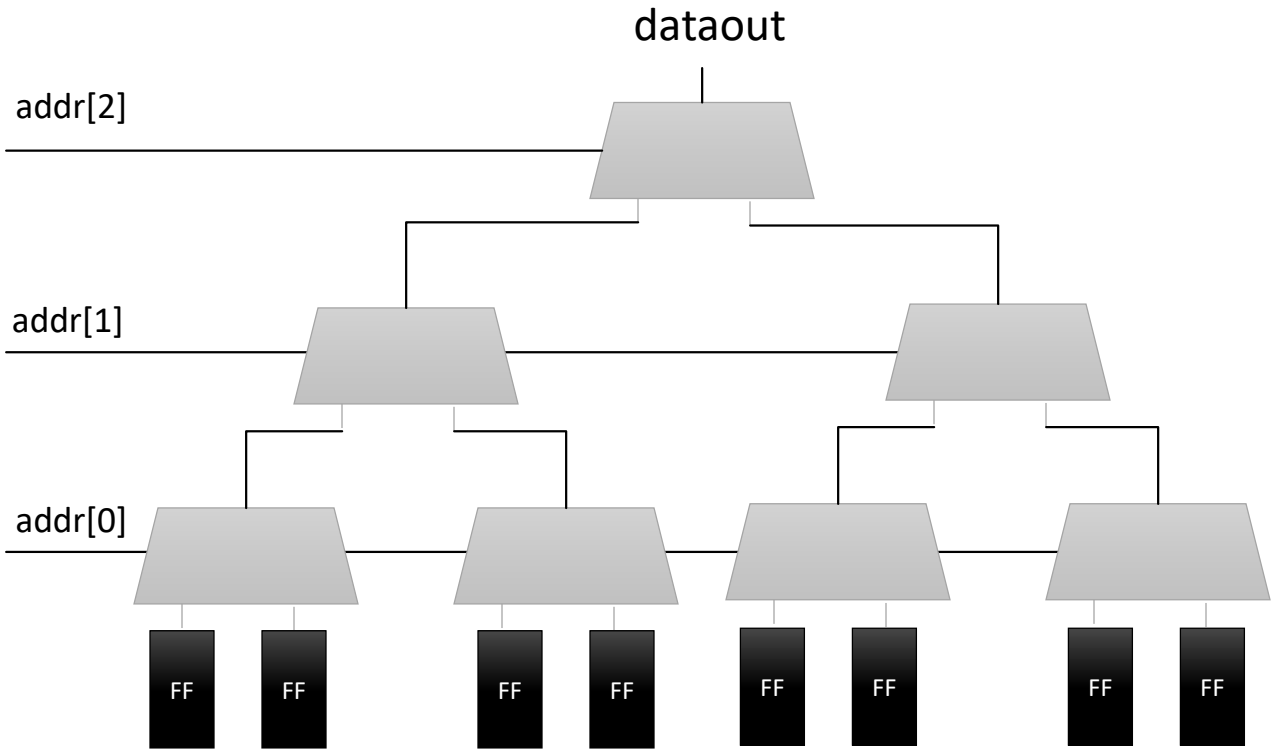
The Structure of a Register File



Register file consists of:

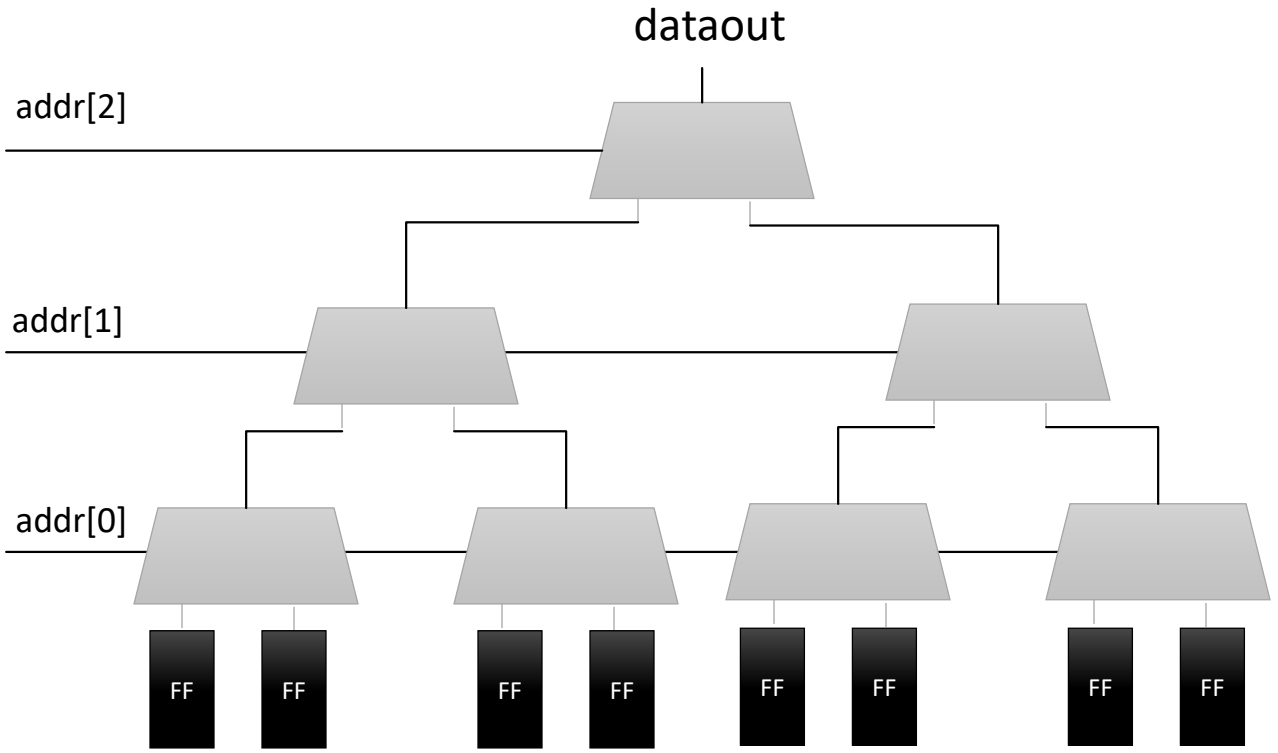
- Flip-flops that store information
- Read logic: takes a read address and outputs stored data
- Write logic: *stores* data in the register file

Identifying Read Logic



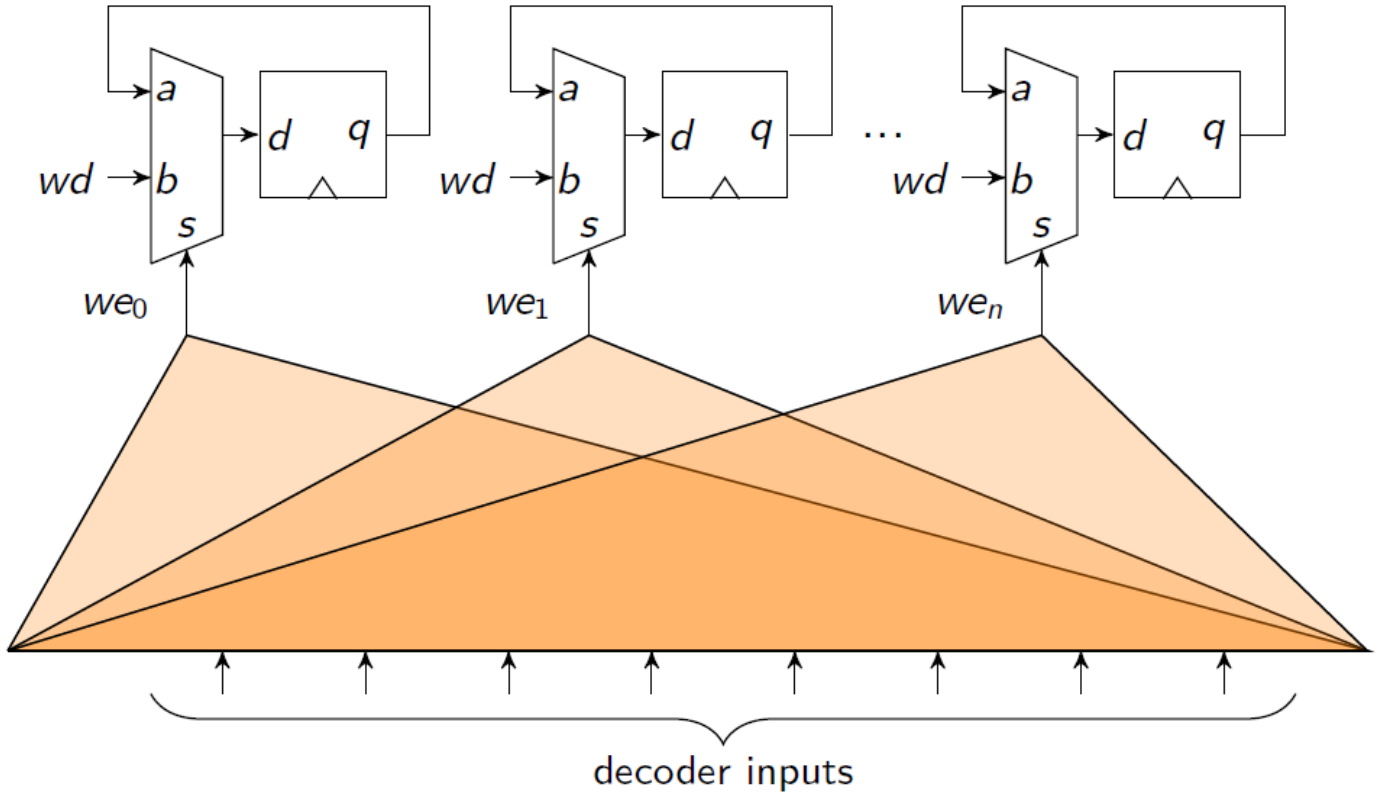
Insight: look for trees of logic where the leaves of the tree are flip-flops

Verifying Identified Read Logic



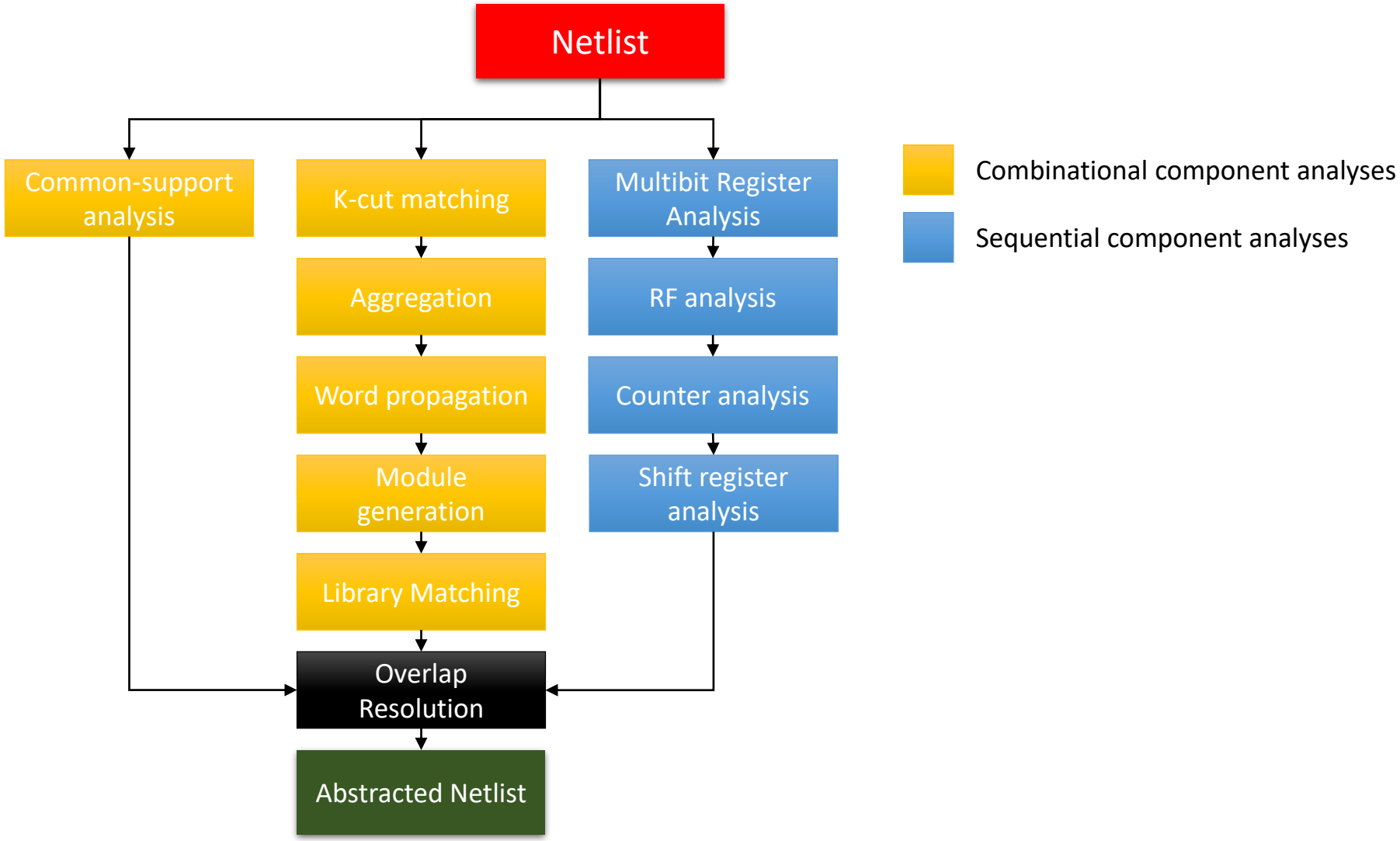
- Verify there exists some address which propagates each flip-flop output to the data output
- This is done using a BDD-based analysis

Identifying Write Logic

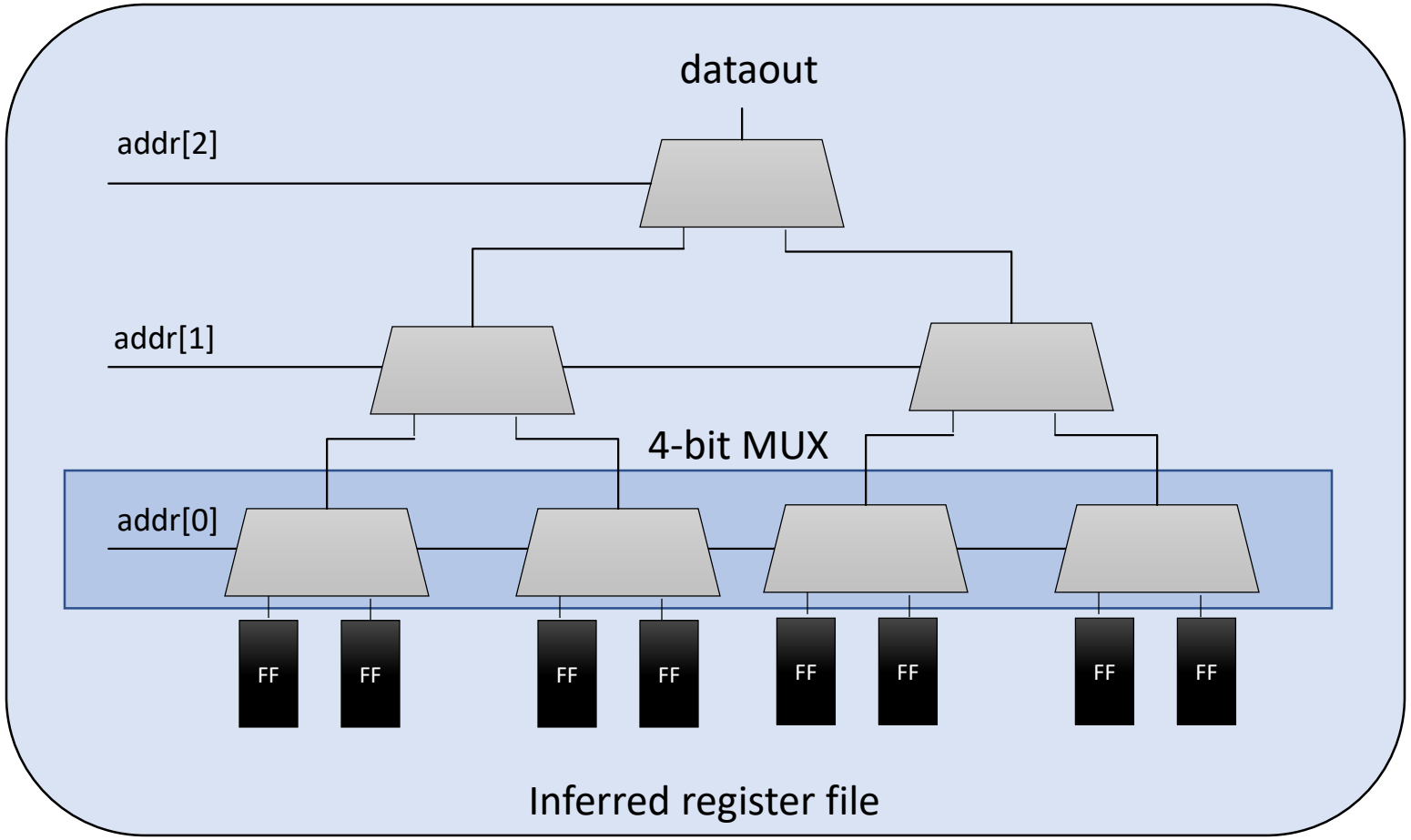


- Muxes select between current value and write data
- Decoders select the location that is being written to
- Easy to find muxes and decoders after we find the flip-flops

Overlap Resolution



Problem: Inferred Modules Overlap



Resolving Overlaps

Formulate an **Integer-Linear Program**

1. Constraints specify that modules must not overlap
2. Objective is one of the following
   - **Maximize** the number of **covered** gates OR
   - **Minimize** the number of modules given a coverage target

Experimental Setup

Toolchain

- Implemented in C++
- MiniSAT 2.2
- CUDD 2.4
- CPLEX 12.5

Designs

- Many from OpenCores.org
- Size ranges from few hundred to several thousand gates
- Also got a test case with 375k gates from DARPA

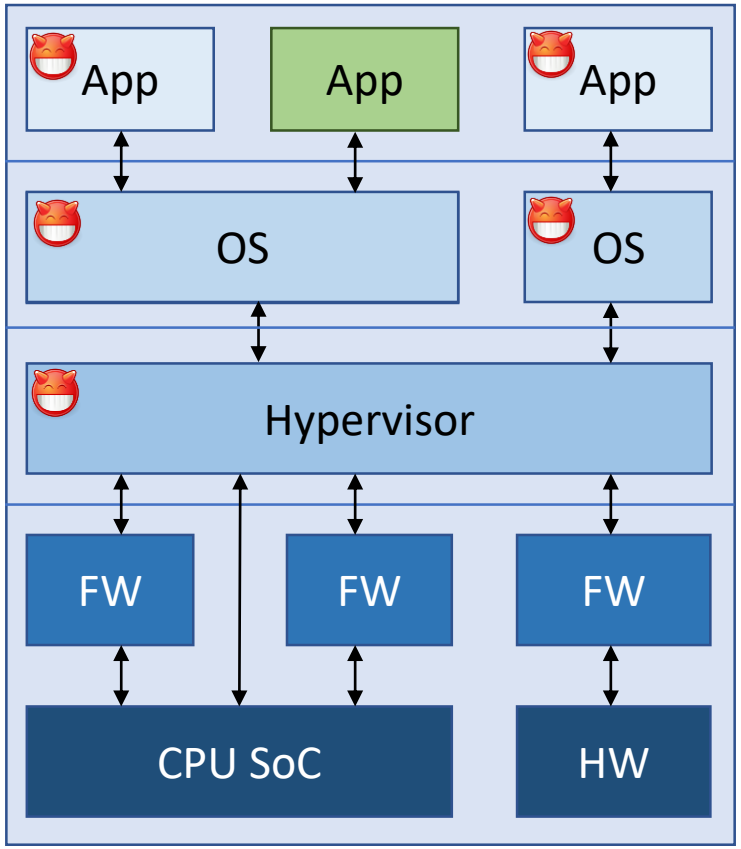
why the deluge of security breaches?

my theory of the case

1. attacks possible from above, below and peers
   (so everything ends up in the TCB)

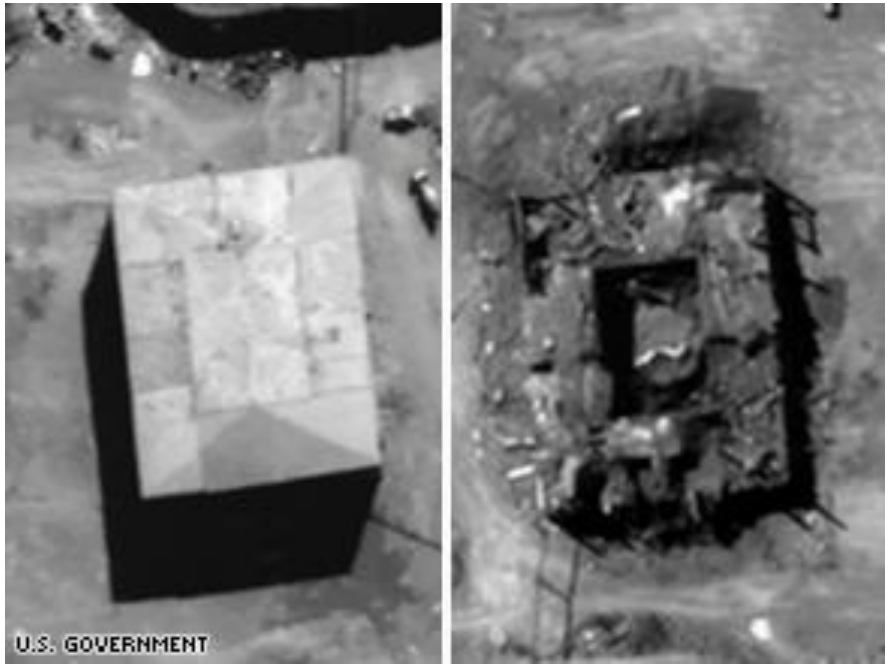
2. layer boundaries are ill-specified
   for ex: AMD vs Intel `sysret`

3. security requirements not formally specified
   for ex: meltdown, foreshadow



what is the impact of hardware bugs?

Before/after pictures of a suspected nuclear reactor site



Suspicion that a hardware backdoor was exploited to disable the radar system

[Sally Adee, *The Hunt for the Kill Switch*, IEEE Spectrum May 2008]
[John Markoff, *Old Trick Threatens the Newest Weapons*, NY Times, 26 October 2009]