UCLID5: Integrating Modeling, Verification, Synthesis, and Learning

Sanjit A. Seshia

Professor

EECS, UC Berkeley

Joint work with: Pramod Subramanyan, Kevin Cheang, Cameron Rasmussen, Rohit Sinha, Ilia Lebedev, Susmit Jha, Randal Bryant, Srinivas Devadas

UCLID5: http://github.com/uclid-org/uclid/

MEMOCODE 2018
October 15, 2018

A Quote from a Classic Paper

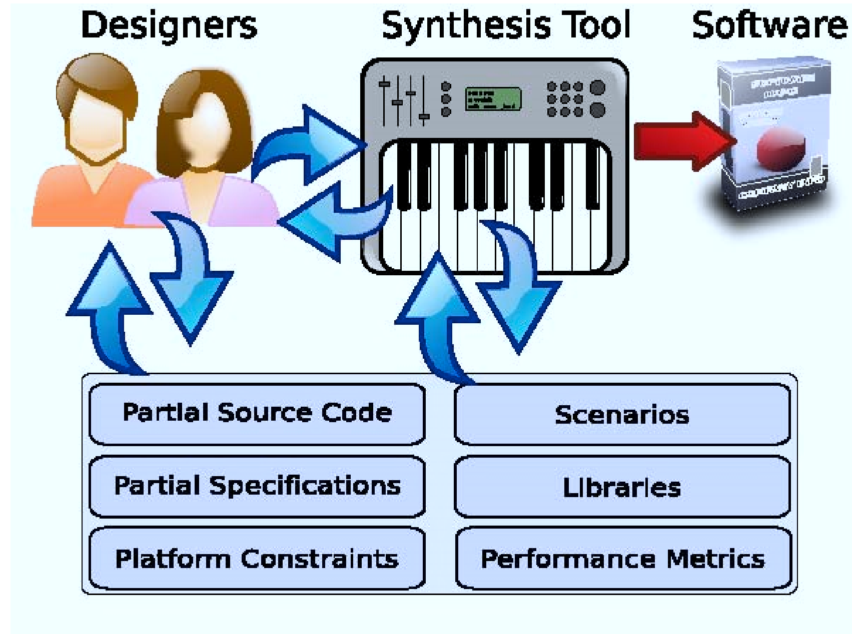
"We propose a method of constructing concurrent programs in which the synchronization skeleton of the program is automatically synthesized from a high-level (branching time) Temporal Logic specification."

E. M. Clarke and E. A. Emerson, 1981
(1st sentence of their original paper on model checking)

Connections: Verification & Synthesis

Counterexample-Guided Inductive Synthesis of Programs (CEGIS)
[ASPLOS 2006,…]

Syntax-Guided Synthesis (SyGuS)
[FMCAD 2013]



NSF ExCAPE Project (2012-2017)

Synthesis ⟷ **Verification**

Learning, Verification, Synthesis: Major Trends

- Specification Mining – Learning Properties from Data
  - an enabler for formal verification in practice

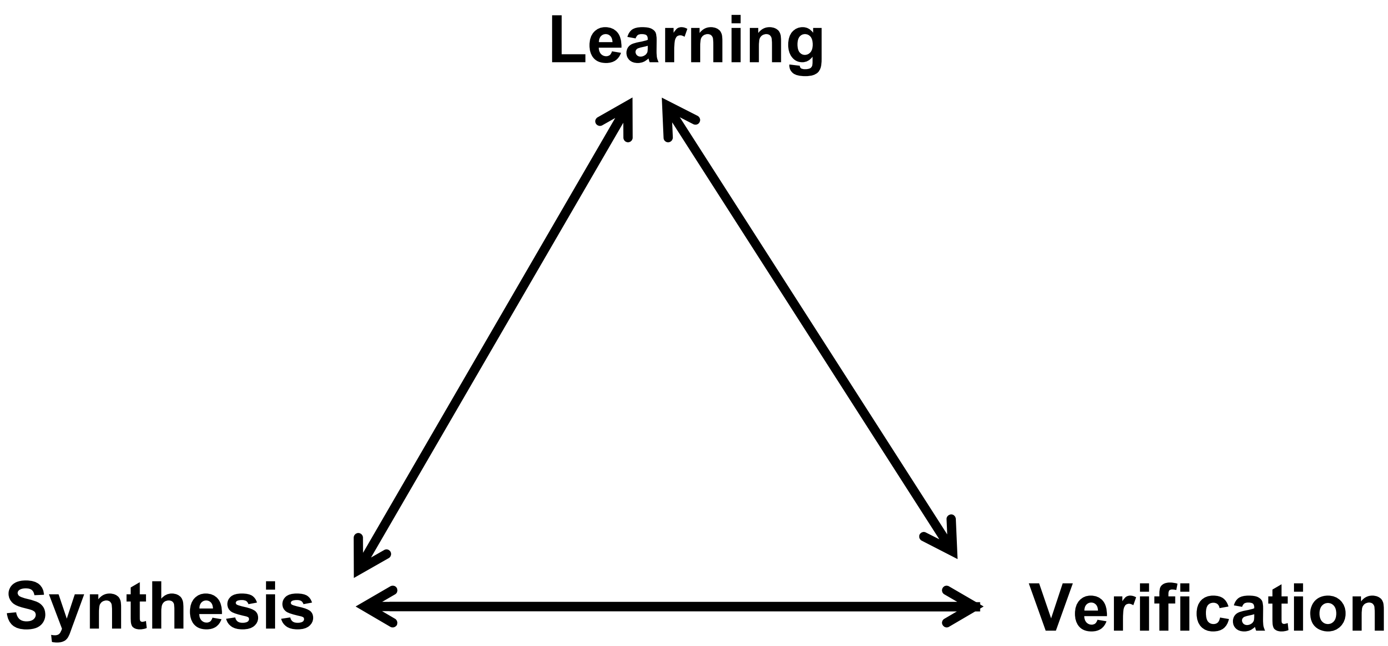
- Inductive Synthesis – Synthesis from Examples
  - a dominant approach to program synthesis

- Data-Driven Design
  - integration of learned components into systems

S. A. Seshia, "Combining Induction, Deduction, and Structure for Verification and Synthesis", Proceedings of the IEEE, November 2015.

More Connections

Observation circa 2016:
No single formal system makes all these connections!



UCLID5: A new formal tool that blends verification, synthesis, and learning

Outline

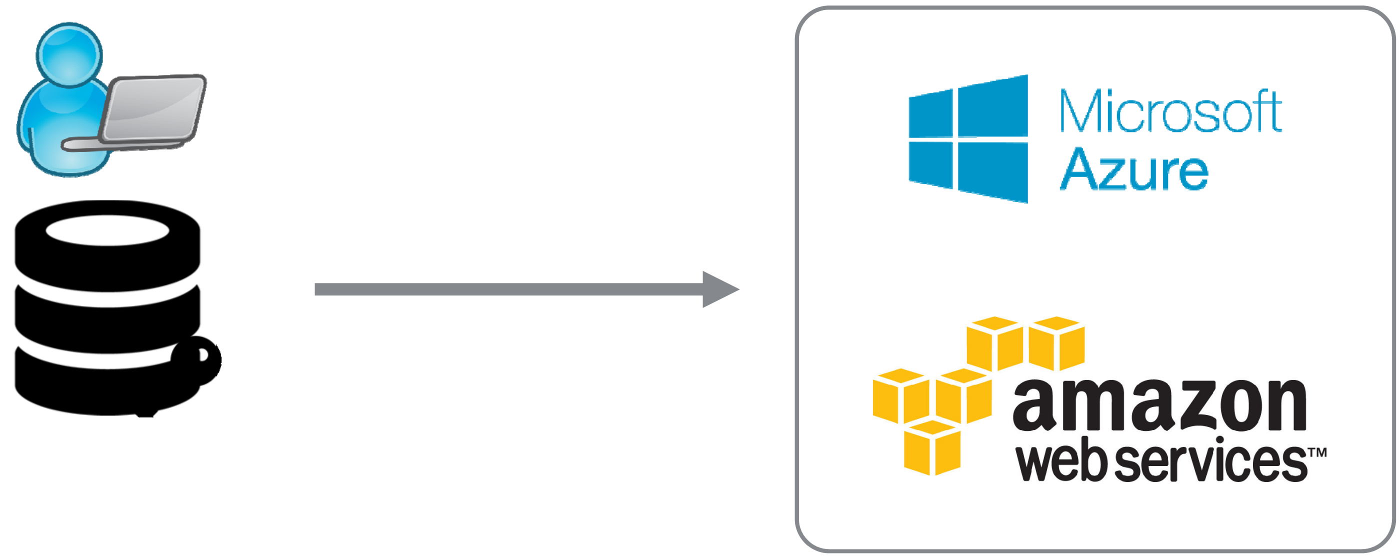
- Motivating Problem: Verification of Trusted Platforms

- Formal Inductive Synthesis and Oracle-Guided Inductive Synthesis

- UCLID5 Modeling, Verification, & Synthesis System

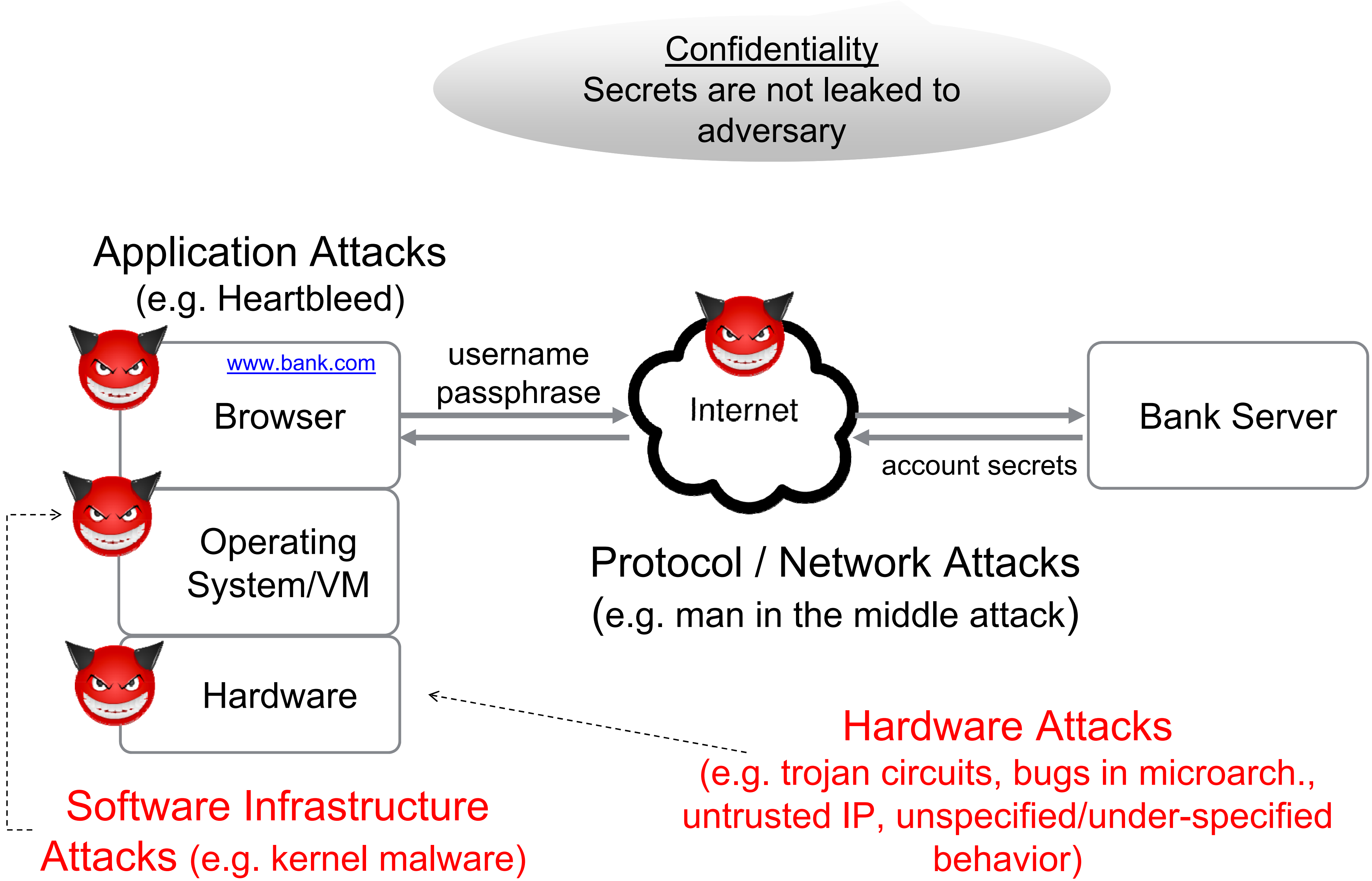
- Conclusion & Future Work

Secure Remote Computation



- Does my secret data remain secret?
- Does the program execute as it is supposed to?
- Is the right program executed?

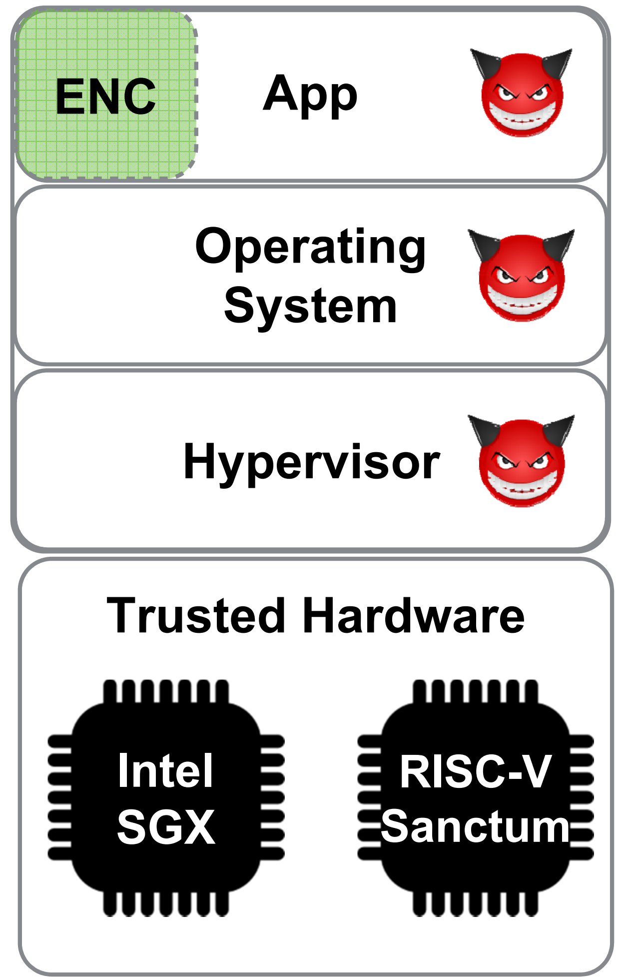
What Classes of Attacks are Possible?



Enclaves and Trusted Hardware

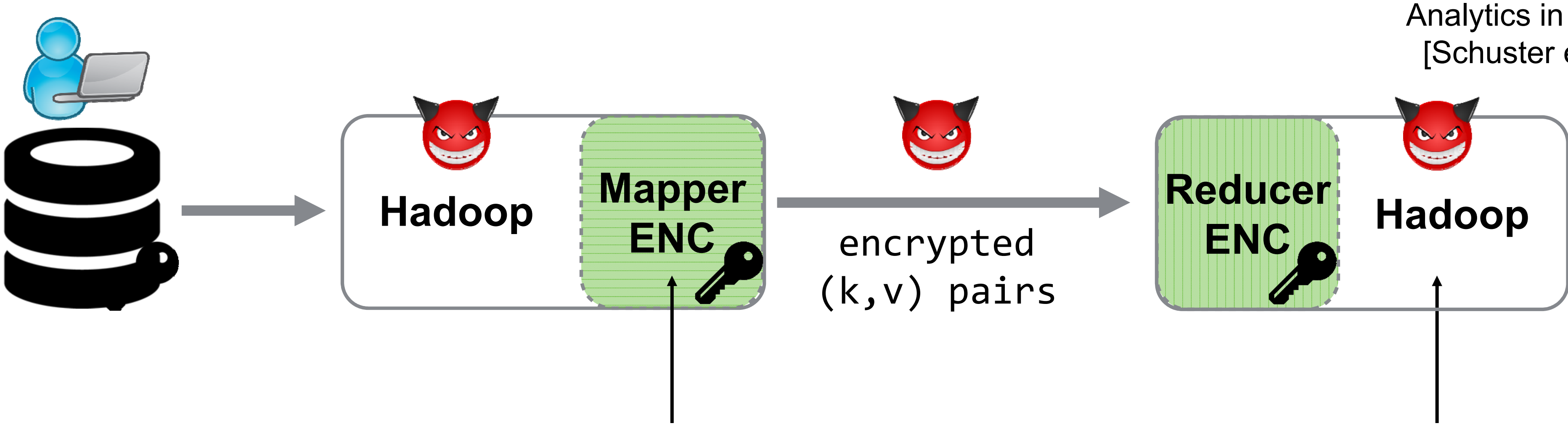
Enclave memory is protected: only enclave code can access it

All trusted computation happens within enclaves



World View with Enclaves

VC3: Trustworthy Data Analytics in the Cloud [Schuster et. al.'15]

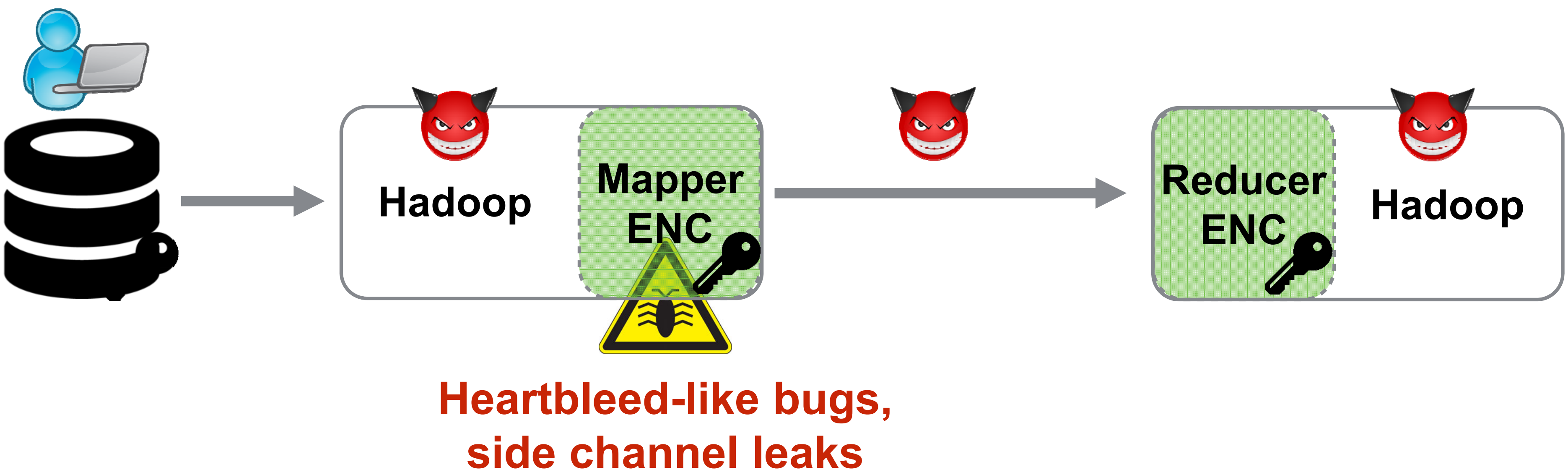



```
/* decrypt input ciphertext
 * compute on sensitive data
 * encrypt and output (k,v)
 */
```

```
/* transmit ciphertexts
 * schedule jobs
 */
```

Software Trusted Computing Base (TCB) contains only enclaves

Bugs in Enclaves can be Exploited



Desiderata:

Confidentiality

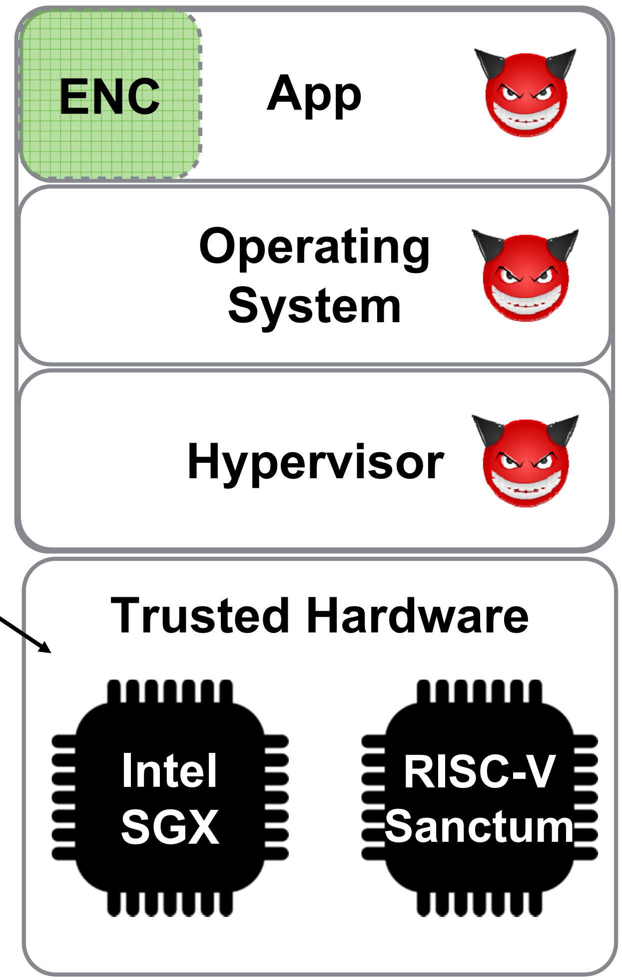
- ✓ Outputs from enclave are always encrypted
- ✓ Side channels do not leak secrets
- ✓ Guarantees on machine code execution

Hardware can be Exploited

"Bugs" in Hardware (e.g. at the Microarchitectural Level) can be Exploited

Meltdown

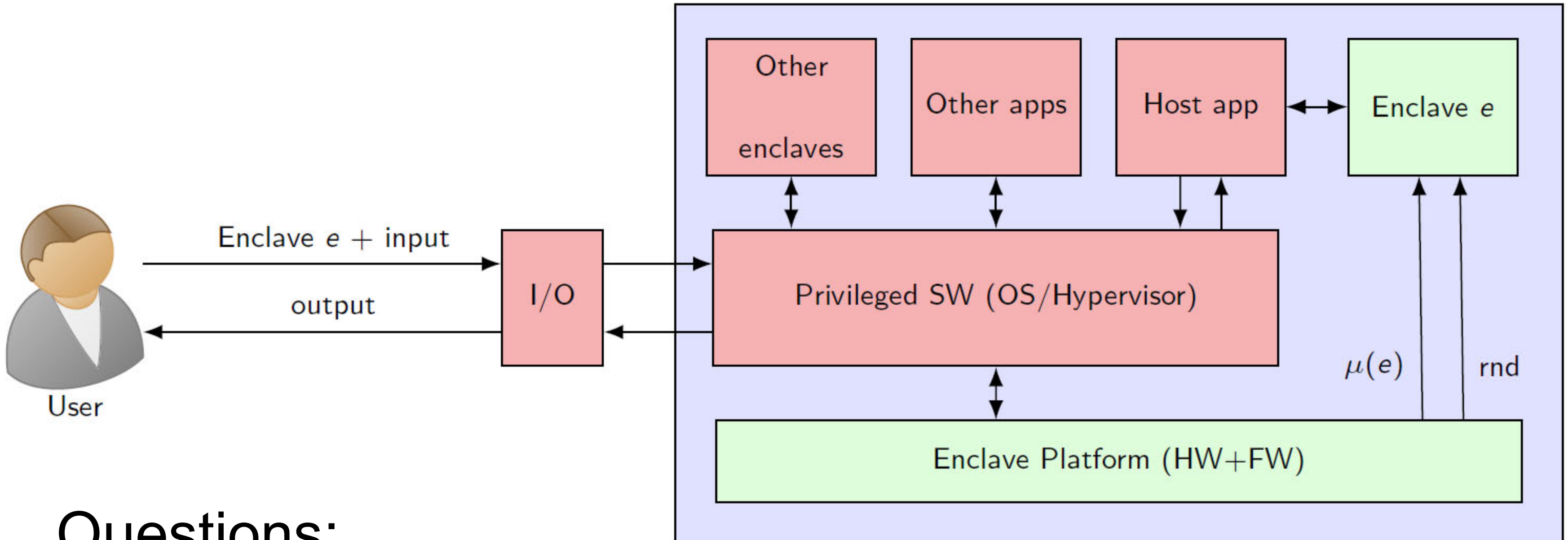
Spectre



How can we formally verify that trusted “enclave” platforms provide secure remote execution?

Secure Remote Execution using Trusted Platforms

[Subramanyan et al., ACM CCS'17]



<u>Questions:</u>

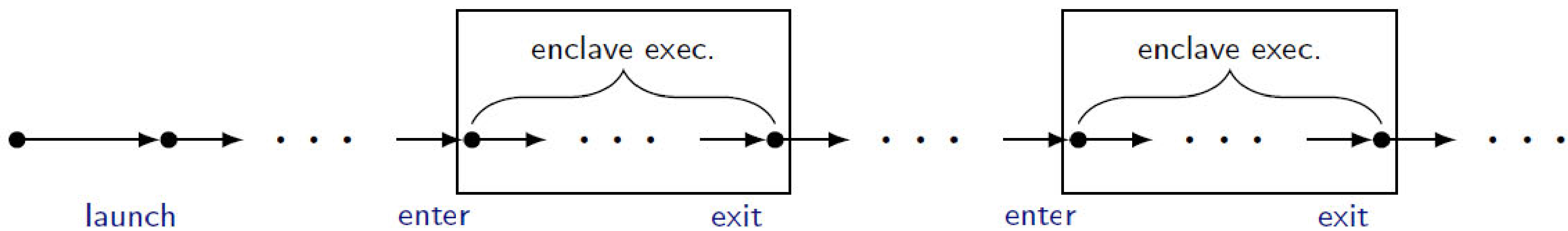
- What does "secure remote execution" mean precisely?
- What primitives must a platform provide for secure remote execution?
- How do we verify that a platform guarantees secure remote execution?

Key Contributions

[Subramanyan et al., ACM CCS'17]

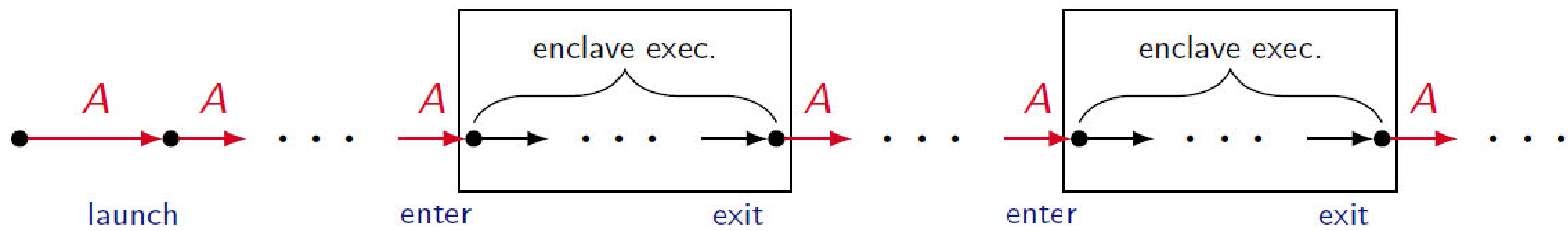
- A formal definition of secure remote execution (SRE)
- Decomposition of SRE into three properties
- Formal model of idealized enclave platform: Trusted Abstract Platform (TAP)
- TAP, Sanctum, SGX models; machined-checked proofs of SRE

Modeling Enclave and Adversary



1. Let $[\![e]\!]$ be the set of all traces for enclave e

Modeling Enclave and Adversary



1. Let $[\![e]\!]$ be the set of all traces for enclave e
2. Assume privileged software adversary who can:
   - tamper with enclave by executing arbitrary platform operations
   - observe public information – defined by an observation function

Secure Remote Execution (SRE): Definition

Remote platform securely executes enclave program e if:

- Any execution trace of e on the platform is contained in $[\![e]\!]$
- Adversary knowledge is restricted to the observation function

Decomposing Secure Remote Execution (SRE)

Remote platform securely executes enclave program e if:

- Any execution trace of e on the platform is contained in $[\![e]\!]$
- Adversary knowledge is restricted to the observation function

Decomposition Theorem

Secure remote execution is implied by three properties:
measurement, integrity and confidentiality.

Proof Sketch:

- Measurement: we are executing the right enclave
- Integrity: adversary influences enclave execution only through inputs
- Confidentiality: adversary knowledge limited to observation function

[all 3 are hyperproperties]

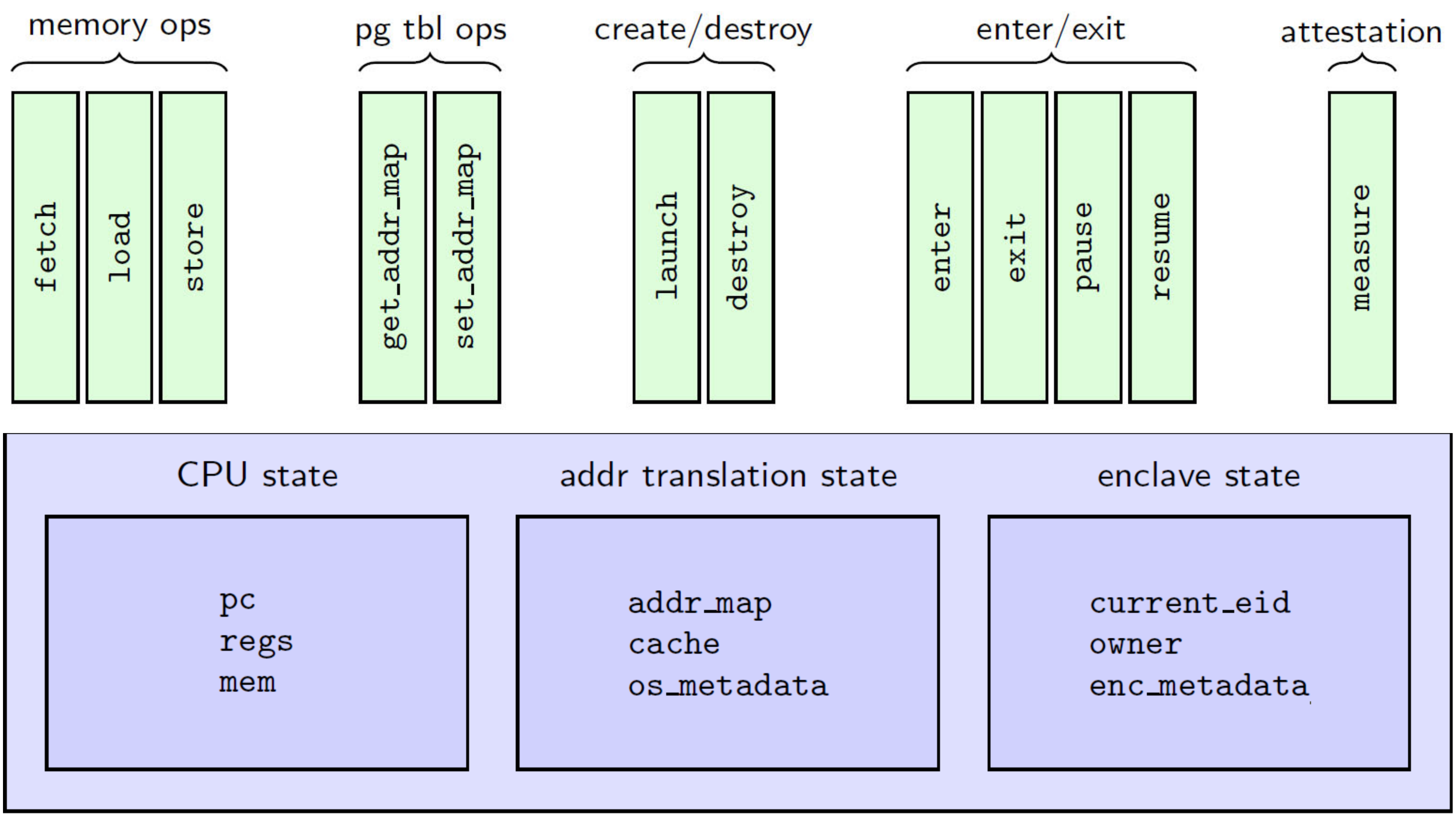
Trusted Abstract Platform (TAP)

What primitives must a platform provide in order to ensure secure remote execution?

TAP models an idealized enclave platform:

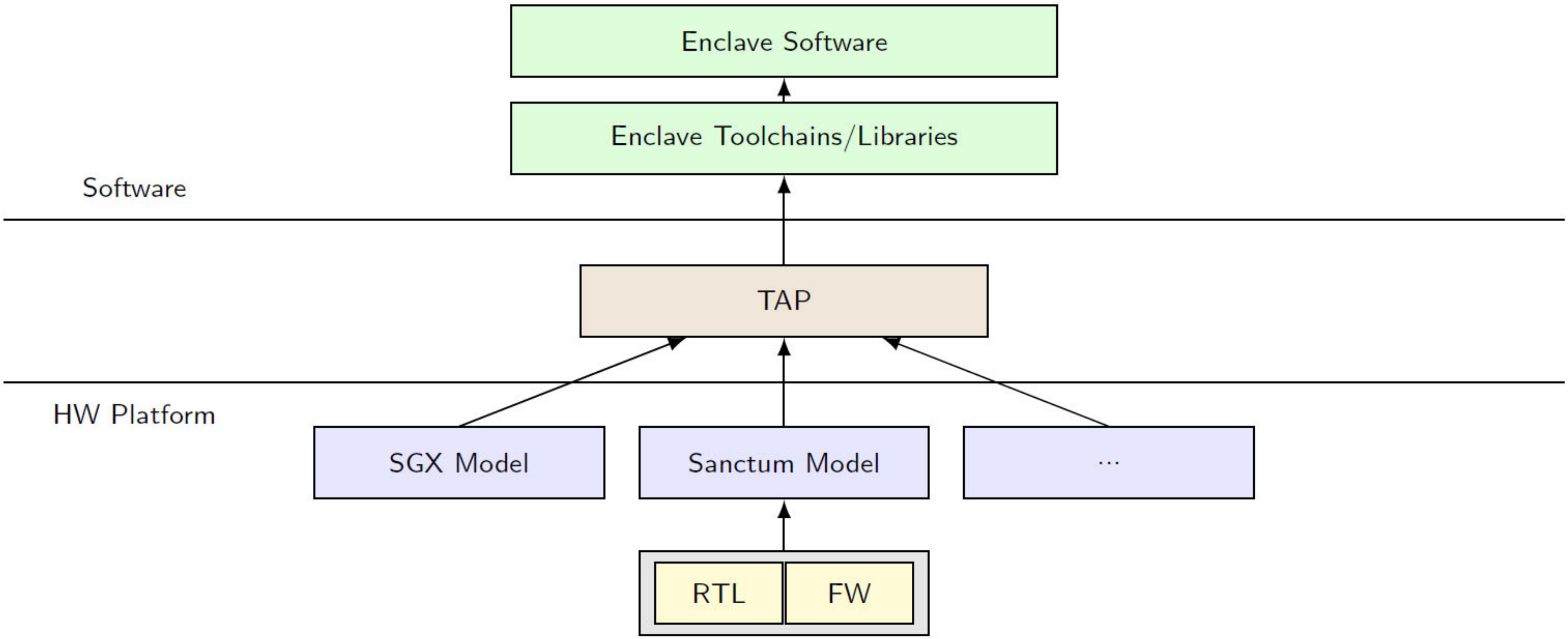
- Independent of platform-specific instruction sets, APIs, etc.
- Allows modeling a range of software adversaries
- Compare security guarantees of different enclave platforms

TAP Model



memory ops
pg tbl ops
create/destroy
enter/exit
attestation
fetch
load
store
get_addr_map
set_addr_map
launch
destroy
enter
exit
pause
resume
measure
CPU state
addr translation state
enclave state
pc
regs
mem
addr_map
cache
os_metadata
current_eid
owner
enc_metadata

How is the TAP Useful?



- For SW, TAP is an abstraction of enclave functionality
- For HW platform designers, TAP is a formal specification

https://github.com/0tcb/TAP

Adversary Model

TAP has a parameterized adversary model.

Adversary		
M		
MC		
MCP		

Adversary Model

TAP has a parameterized adversary model.

Adversary	Tamper	
M MC MCP	invoke any TAP operation with arbitrary operands	

- Tamper: defines how the adversary can modify platform state

Adversary Model

TAP has a parameterized adversary model.

Adversary	**Tamper**	**Observe**
M	invoke any TAP	only Memory state
MC	operation with	only Memory and Cache state
MCP	arbitrary operands	Memory, Cache, Page table state

- Tamper: defines how the adversary can modify platform state
- Observation fn: what platform state is adversary-visible?

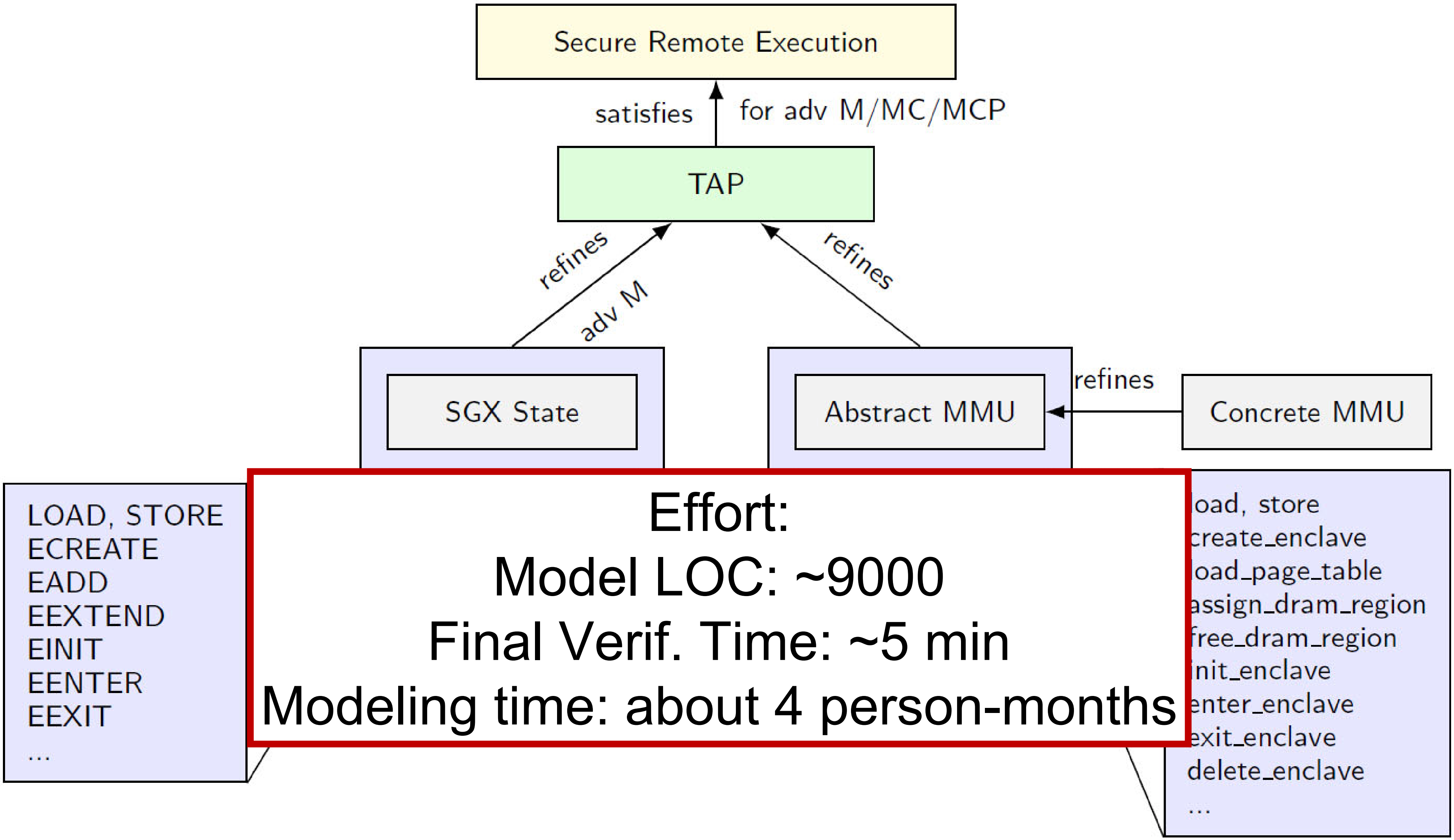
Does TAP satisfy Secure Remote Execution?

Property	Adversary M	Adversary MC	Adversary MCP
Measurement	✓	✓	✓
Integrity	✓	✓	✓
Confidentiality	✓	✓[1]	✓[2]

[1]if cache sets are partitioned
[2]if cache sets are partitioned and enclave page tables are private

SGX is secure for adversary M
Sanctum is secure for adversary MCP

Do SGX/Sanctum refine the TAP?



Models and proofs written in Boogie (which in turn uses Z3)

Lessons Learned from Trusted Platform Modeling/Verification Effort

- *Need* ***better modeling language*** to model both sequential software and concurrent hardware
  - Boogie excellent for sequential software, but not a good match for the hardware portions
  - Traditional hardware verification languages not a good fit for software components
- *Need* ***more automation*** in the verification process
  - Generation of inductive invariants
  - Generation of assume/guarantee contracts
  - Verification of hyperproperties (2-safety properties) for integrity, confidentiality, etc.
- Need ***incremental & compositional*** model synthesis & verification

Outline

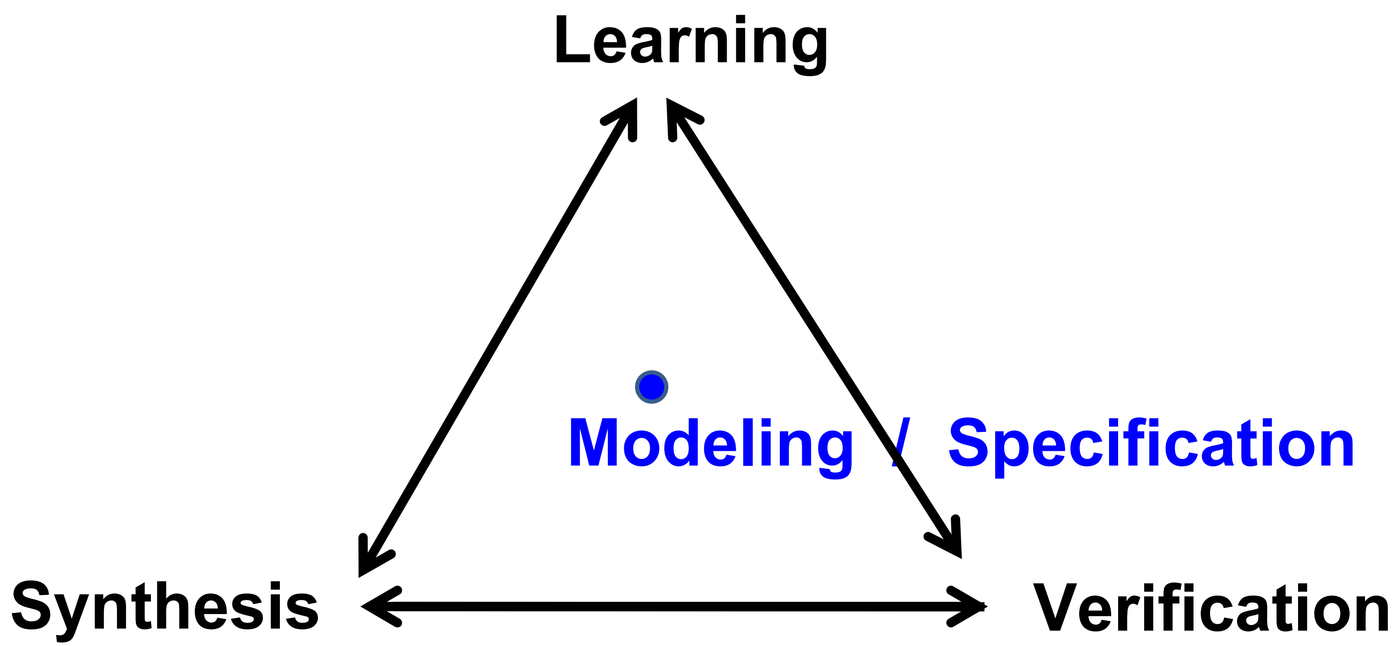
- Motivating Problem: Verification of Trusted Platforms
- Formal Inductive Synthesis and Oracle-Guided Inductive Synthesis
- UCLID5 Modeling, Verification, & Synthesis System
- Conclusion & Future Work

Artifacts Synthesized in Verification

[S. A. Seshia, DAC 2012; Proc. IEEE, November 2015]

- Inductive invariants
- Abstraction functions / abstract models
- Auxiliary specifications (e.g., pre/post-conditions, function summaries)
- Simulation relations
- Environment assumptions / Env model / interface specifications
- Interpolants
- Ranking functions
- Intermediate lemmas for compositional proofs
- Theory lemma instances in SMT solving
- Patterns for Quantifier Instantiation
- …

Formal Modeling & Specification is Central



Example: Verification by Reduction to Synthesis

- Transition System
  - Init: I

    $$x = 1 \wedge y = 1$$

  - Transition Relation: δ

    $$x' = x+y \;\wedge\; y' = y+x$$

- Property: Ψ = G $(y \geq 1)$
- Attempted Proof by Induction:

  $$(\; y \geq 1 \wedge\; x' = x+y \;\wedge\; y' = y+x \;) \Rightarrow\; y' \geq 1$$

➢ Fails. Need to Strengthen Invariant: Find ϕ s.t.

$$x \geq 1 \;\wedge y \geq 1 \wedge\; x' = x+y \;\wedge\; y' = y+x \;\Rightarrow\; x' \geq 1 \wedge\; y' \geq 1$$

- Safety Verification → Invariant Synthesis

One Reduction from Verification to Synthesis

NOTATION

Transition system $M = (I, \delta)$

Safety property $\Psi = G(\psi)$

VERIFICATION PROBLEM

Does M satisfy Ψ?

SYNTHESIS PROBLEM

Synthesize ϕ s.t.

$$I \Rightarrow \phi \wedge \psi$$

$$\phi \wedge \psi \wedge \delta \Rightarrow \phi' \wedge \psi'$$

Two Reductions from Verification to Synthesis

NOTATION

Transition system M = (I, δ)

Safety property Ψ = G(ψ)

VERIFICATION PROBLEM

Does M satisfy Ψ?

SYNTHESIS PROBLEM #1

Synthesize ϕ s.t.

$$I \Rightarrow \phi \wedge \psi$$

$$\phi \wedge \psi \wedge \delta \Rightarrow \phi' \wedge \psi'$$

SYNTHESIS PROBLEM #2

Synthesize $\alpha : S \rightarrow \hat{S}$ where

$\alpha(M) = (\hat{I}, \hat{\delta})$

s.t.

$\alpha(M)$ satisfies Ψ

iff

M satisfies Ψ

Common Framework for both Reductions: Formal Inductive Synthesis

[Seshia, Proc. IEEE 2015]

Synthesis of:-

- Inductive Invariants
  - Choose templates for invariants
  - Infer likely invariants from tests (examples)
  - Check if any are true inductive invariants, possibly iterate

- Abstraction Functions
  - Choose an abstract domain
  - Use Counter-Example Guided Abstraction Refinement (CEGAR)

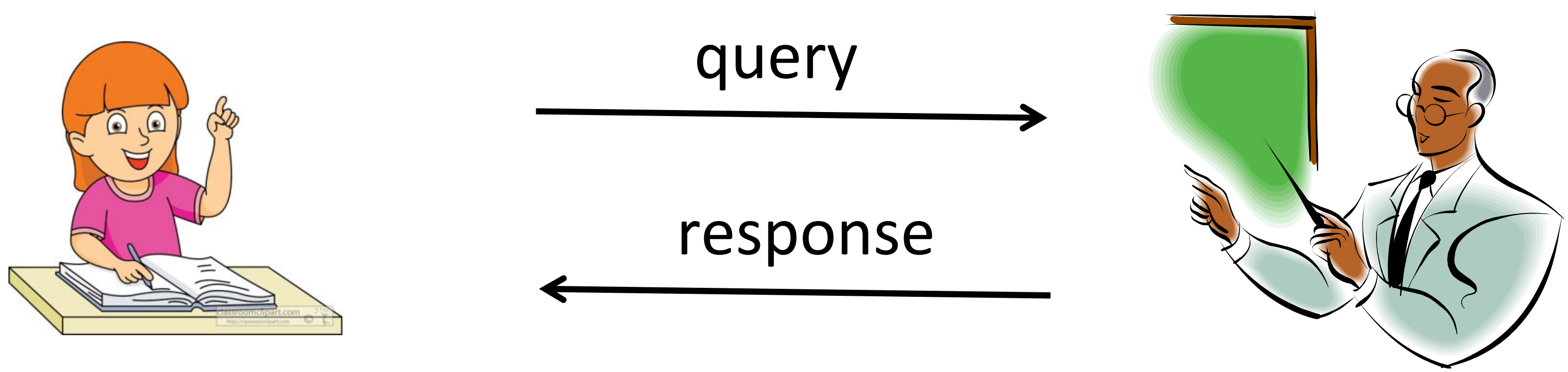
From CEGIS to Oracle-Guided Inductive Synthesis

Inductive Synthesis: Learning from Examples (ML)

Formal Inductive Synthesis: Learn from Examples *while satisfying a Formal Specification*

General Approach: **Oracle-Guided Learning**

Combine Learner with Oracle (e.g., Verifier) that answers Learner's Queries

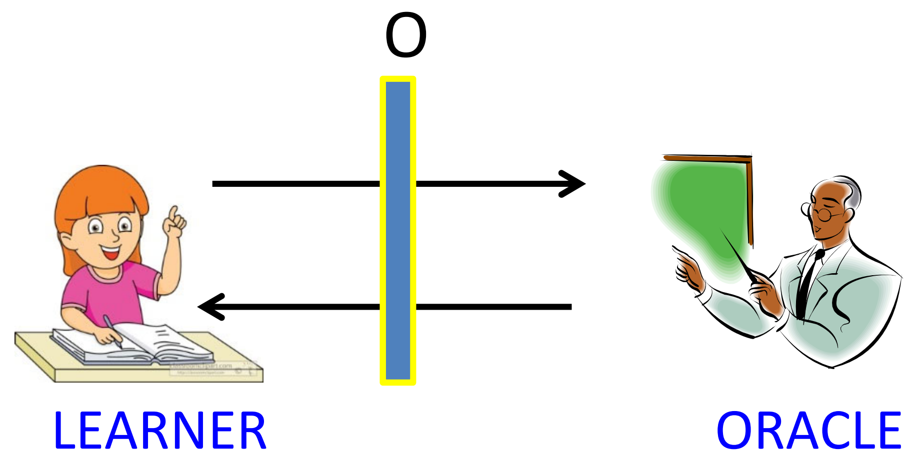


LEARNER ORACLE

[Jha & Seshia, "A Theory of Formal Synthesis via Inductive Learning", 2015, Acta Informatica 2017.]

Formal Inductive Synthesis

- Given:
  - Class of Artifacts C
  - Formal specification ϕ
  - Domain of examples D
  - Oracle Interface O
    - Set of (query, response) types



- Find, by adhering to O, an $f \in C$ that satisfies ϕ
  - i.e. O defines protocol to access to D or ϕ

- To solve this: Design/Select *BOTH* Learner and Oracle

[Jha & Seshia, "A Theory of Formal Synthesis via Inductive Learning", 2015; 2017]

Common Oracle Query Types

(for trace property ϕ)

(more examples in [Jha & Seshia, 2017])

LEARNER

Positive Witness

$x \in \phi$, if one exists, else $\perp$

Negative Witness

$x \notin \phi$, if one exists, else $\perp$

Membership: Is $x \in \phi$?

Yes / No

Equivalence: Is $f = \phi$?

Yes / No + $x \in \phi \oplus f$

Subsumption/Subset: Is $f \subseteq \phi$?

Yes / No + $x \in f \setminus \phi$

Distinguishing Input: $f, X \subseteq f$

f' s.t. $f' \neq f \wedge X \subseteq f'$, if it exists; o.w. $\perp$

ORACLE

Comparison*

[see also, Jha & Seshia, 2015; 2017]

Feature	Formal Inductive Synthesis	Machine Learning
Concept/Program Classes	Programmable, Complex	Fixed, “Simple”
Learning Algorithms	General-Purpose Solvers	Specialized
Learning Criteria	Exact, w/ Formal Spec	Approximate, w/ Cost Function
Oracle-Guidance	*Common (can select/design Oracle)*	*Rare (black-box oracles)*

* Between typical inductive synthesizer and machine learning algo

Query Types for Counterexample-Guided Inductive Synthesis (CEGIS)

Positive Witness

$x \in \phi$, if one exists, else $\perp$

Counterexample to f?

Yes + counterexample x / $\perp$

Finite memory vs
Infinite memory

Type of counter-example given

Concept class: Any set of recursive languages

Some Initial Theoretical Results on CEGIS

[Jha & Seshia, 2015; Jha, Seshia, Zhu, 2016]

- Finite-sized Concept/Program Classes:
  - Teaching Dimension [Goldman & Kearns '90] is a lower bound on query complexity
  - TD of n-dimensional rectangles is O(n), of n-dimensional octagons is $O(n^2)$
  - Relevance for Invariant Inference
- Infinite-sized Concept Classes:
  - Analyze CEGIS variants for "learning in the limit" [Gold, 1967]
  - Minimizing counterexamples does not change learnability
  - Getting "positive-bounded" counterexamples can enable one to learn more than standard CEGIS when learner buffer size is finite
- Much more to be investigated!!!

Outline

- Motivating Problem: Verification of Trusted Platforms
- Formal Inductive Synthesis and Oracle-Guided Inductive Synthesis
- UCLID5 Modeling, Verification, & Synthesis System
- Conclusion & Future Work

Recap: Lessons Learned from TAP Modeling/Verification Effort

- *Need better modeling language* to model both sequential software and concurrent hardware
- *Need more automation* in the verification process
  - Synthesis of verification artifacts (auxiliary specs, etc.)
- Need *incremental* model synthesis & verification

UCLID5: A New Formal Modeling and Verification System

https://github.com/uclid-org/uclid

Background: Original UCLID Modeling & Verification System (2001-2014)

[Bryant, Lahiri, Seshia, CAV 2002]

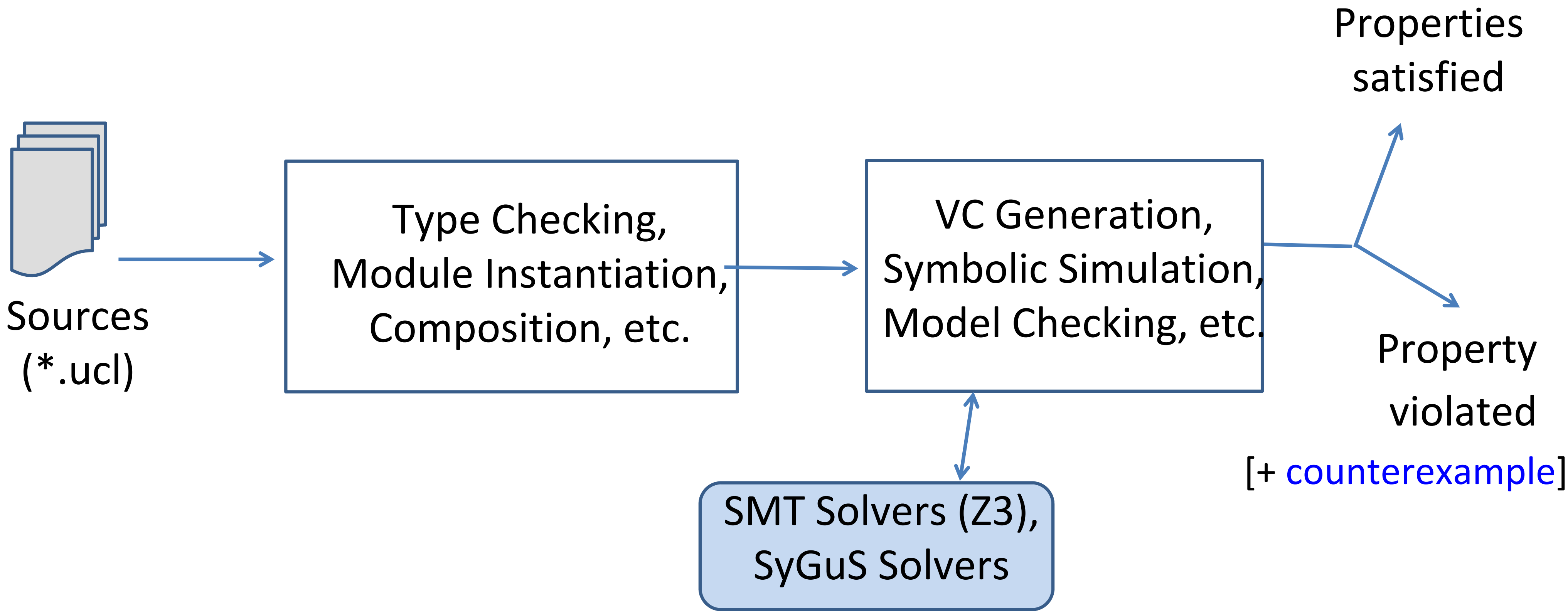
- One of the first satisfiability modulo theories (SMT) solvers and SMT-based verifiers
- Term-level modeling
  - Model transition systems using first-order logic with background theories
  - Verification based on bounded unrolling of transition relation
    - Bounded Model Checking
    - (k-)Induction
    - Checking Simulation (Correspondence Checking)
- Wide range of applications:
  - Processor verification, protocol verification, finding security vulnerabilities, etc.

Desired Features for Verification Tools

Desired Feature\Tool	ABC	NuXMV	Boogie	Coq	UCLID	UCLID5
Expressive Types (bits -> words -> terms)	Weak	Medium	Strong	Strong	Strong	Strong
High Degree of Automation	Strong	Strong	Medium	Weak	Medium	Strong
Wide Variety of Verification Methods	Strong	Medium	Medium	Strong	Medium	Strong
Modular Specification & Verification	Strong	Medium	Strong	Medium	Weak	Strong
Support for Sequential updates (Seq. software)	Weak	Weak	Strong	Strong	Weak	Strong
Support for Concurrent updates (Synchronous HW)	Strong	Strong	Weak	Medium	Strong	Strong
Support for Meaningful Counterexample Generation	Strong	Medium	Weak	Weak	Medium	Strong

Strong Medium Weak

UCLID5 Verifier



Supported Types in UCLID5

- Booleans
- Bit-vectors
- Integers (unbounded)
- Enumerated Types
- Arrays
- Records
- Uninterpreted functions & predicates

➢ Most theories supported by SMT solvers

Structure of a UCLID5 Module

```
module example {
    // type & (input, output, state) variable declarations
    type …
    var …
    // define macros
    define <macro-name> …
    // procedures
    procedure <proc-name> … { …  }
    // transition relation
    init { … }  // define set of initial states
    next { … }  // define transition relation
    // module specifications
    invariant … // invariant property
    property[LTL] … // linear temporal logic property
    // control block - proof script within module defines 
verification
    control { … }
}
```

Specification & Verification with UCLID5

- Control block specifies proof script within a module
- Specifications
  - Seq. Programs: Pre/Post-Conditions, Asserts, Assumes
  - Invariants, Linear Temporal Logic
  - Simulation/Refinement Checking
  - 2-Safety Hyperproperties
- Use of Syntax-Guided Synthesis (SyGuS) for automated synthesis of model/specifications (e.g. invariants)
- Subsumes verification capabilities of original UCLID system
  - Bounded model checking, k-induction, simulation checking
  - *Seq. program verification*
  - *Hyperproperty verification*
- Supports Modular Specification & Verification

Brief Demo of UCLID5

- Proving Determinism of a Simple CPU that implements Isolated Memory Regions (over-simplified version of enclaves)

Outline

- Motivating Problem: Verification of Trusted Platforms
- Formal Inductive Synthesis and Oracle-Guided Inductive Synthesis
- UCLID5 Modeling, Verification, & Synthesis System
- Conclusion & Future Work

Conclusion

- Confluence of Trends:
  - Tight connection between Verification and Synthesis
  - Data-driven design meets Model-based design
  - Machine Learning can enhance Verification & Synthesis
  - Systems becoming more heterogeneous (HW-SW, cyber-physical, etc.)
- Formal Tools must Address and Leverage these Trends
  - Motivating Example: Platform Security
- UCLID5: A New Formal System
  - Leverages the theory of Formal Inductive Synthesis
  - Supports diverse specification/verification/modeling tasks
  - Supports compositional (modular) reasoning
  - Open source, publicly available

Thank you!

Key References:

- "UCLID5: Integrating Modeling, Verification, Synthesis, and Learning", Seshia & Subramanyan, MEMOCODE 2018. http://github.com/uclid-org/uclid/
- "A Formal Foundation for Secure Remote Execution of Enclaves", Subramanyan et al., CCS 2017.
- "Combining Induction, Deduction, and Structure for Verification and Synthesis", Seshia, Proc. IEEE 2015.
- "A Theory of Formal Synthesis via Inductive Learning", Jha and Seshia, Acta Informatica 2017.
- Original UCLID paper: Bryant, Lahiri, and Seshia, CAV 2002.