UCLID5: Integrating Modeling, Verification, Synthesis, and Learning

Sanjit A. Seshia
Professor
EECS, UC Berkeley

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

More Connections

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

Learning

Synthesis  Verification

UCLID5: A new formal tool that blends verification, synthesis, and learning
A Few Assumptions about your background: the “What” of SAT, SMT, Model Checking
Boolean Satisfiability (SAT)

Is there an assignment to the \( p_i \) variables s.t. \( \phi \) evaluates to 1?
Satisfiability Modulo Theories (SMT)

Is there an assignment to the $x, y, z, w$ variables s.t. $\phi$ evaluates to 1?
Model Checking

• Broad Defn:
  A collection of *algorithmic methods*
  based on *state space exploration*
  used to verify if a *system satisfies a formal specification*.

• Original Defn: (Clarke)
  A technique to check if a *finite-state system is a model of (satisfies) a temporal logic property*. 
Outline

- Motivating Problem: Verification of Trusted Platforms
- Formal Inductive Synthesis, Syntax-Guided 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?

Confidentiality
Secrets are not leaked to adversary

Application Attacks
(e.g. Heartbleed)

Browser

www.bank.com

username
passphrase

Internet

Bank Server

Protocol / Network Attacks
(e.g. man in the middle attack)

Software Infrastructure Attacks
(e.g. kernel malware)

Operating System/VM

Hardware

Hardware Attacks
(e.g. trojan circuits, bugs in microarch., untrusted IP, unspecified/under-specified behavior)
Enclaves and Trusted Hardware

Enclave memory is protected: only enclave code can access it

All trusted computation happens within enclaves
World View with Enclaves

Software Trusted Computing Base (TCB) contains only enclaves

VC3: Trustworthy Data Analytics in the Cloud [Schuster et. al.'15]
Bugs in Enclaves can be Exploited

Heartbleed-like bugs, side channel leaks

Desiderata:

- Outputs from enclave are always encrypted
- Side channels do not leak secrets
- Guarantees on machine code execution
“Bugs” in Hardware (e.g. at the Microarchitectural Level) can be Exploited

- Meltdown
- Spectre

**Software Level**
- App
- Operating System
- Hypervisor

**Hardware Level**
- Intel SGX
- RISC-V Sanctum

**Trusted Hardware**
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]

Questions:
- 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

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 $\llbracket e \rrbracket$
- 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]
Enclave Confidentiality

Adversary observations must be a deterministic function of enclave public outputs (and nothing else)

![Diagram showing adversarial observations and proof obligations]

color key: assumptions are in blue while proof obligations are in red
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:
  - Fetch
  - Load
  - Store

- PG Tbl Ops:
  - Get_addr_map
  - Set_addr_map

- Create/Destroy:
  - Launch
  - Destroy

- Enter/Exit:
  - Enter
  - Exit
  - Pause
  - Resume

- Attestation:
  - Measure

CPU State:
- PC
- Regs
- Mem

Addr Translation State:
- Addr_map
- Cache
- OS_metadata

Enclave State:
- 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
TAP has a parameterized adversary model.

<table>
<thead>
<tr>
<th>Adversary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td></td>
</tr>
<tr>
<td>MC</td>
<td></td>
</tr>
<tr>
<td>MCP</td>
<td></td>
</tr>
</tbody>
</table>
Adversary Model

TAP has a parameterized adversary model.

<table>
<thead>
<tr>
<th>Adversary</th>
<th>Tamper</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>invoke any TAP operation</td>
</tr>
<tr>
<td>MC</td>
<td>with arbitrary operands</td>
</tr>
<tr>
<td>MCP</td>
<td></td>
</tr>
</tbody>
</table>

Tamper: defines how the adversary can modify platform state
Adversary Model

TAP has a parameterized adversary model.

<table>
<thead>
<tr>
<th>Adversary</th>
<th>Tamper</th>
<th>Observe</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>invoke any TAP operation with arbitrary operands</td>
<td>only Memory state</td>
</tr>
<tr>
<td>MC</td>
<td></td>
<td>only Memory and Cache state</td>
</tr>
<tr>
<td>MCP</td>
<td></td>
<td>Memory, Cache, Page table state</td>
</tr>
</tbody>
</table>

- Tamper: defines how the adversary can modify platform state
- Observation fn: what platform state is adversary-visible?
Does TAP satisfy Secure Remote Execution?

<table>
<thead>
<tr>
<th>Property</th>
<th>Adversary M</th>
<th>Adversary MC</th>
<th>Adversary MCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Integrity</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>✓</td>
<td>✓¹</td>
<td>✓²</td>
</tr>
</tbody>
</table>

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?

Secure Remote Execution

satisfies for adv M/MC/MCP

TAP

refines adv M

refines

SGX State

Abstract MMU

Concrete MMU

Effort:
Model LOC: ~9000
Final Verif. Time: ~5 min
Modeling time: about 4 person-months

LOAD, STORE
ECREATE
EADD
EEXTEND
EINIT
EENTER
EEXIT
...

load, store
create_enclave
load_page_table
assign_dram_region
free_dram_region
init_enclave
enter_enclave
exit_enclave
delete_enclave
...

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, Syntax-Guided 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

Learning

Modeling / Specification

Synthesis ↔ Verification
Example: Verification by Reduction to Synthesis

• Transition System
  – Init: \( I \)
    \[
    x = 1 \land y = 1
    \]
  – Transition Relation: \( \delta \)
    \[
    x' = x + y \land y' = y + x
    \]

• Property: \( \Psi = G (y \geq 1) \)

• Attempted Proof by Induction:
  \[
  ( y \geq 1 \land x' = x + y \land y' = y + x ) \implies y' \geq 1
  \]

  \( \neg \) Fails. Need to Strengthen Invariant: Find \( \phi \) s.t.
  \[
  x \geq 1 \land y \geq 1 \land x' = x + y \land y' = y + x \implies x' \geq 1 \land y' \geq 1
  \]

• Safety Verification \( \rightarrow \) Invariant Synthesis
One Reduction from Verification to Synthesis

**NOTATION**
Transition system $M = (I, \delta)$
Safety property $\Psi = G(\psi)$

**VERIFICATION PROBLEM**
Does $M$ satisfy $\Psi$?

**SYNTHESIS PROBLEM**
Synthesize $\phi$ s.t.

\[ I \Rightarrow \phi \land \psi \]
\[ \phi \land \psi \land \delta \Rightarrow \phi' \land \psi' \]
Two Reductions from Verification to Synthesis

NOTATION
Transition system $M = (I, \delta)$
Safety property $\Psi = G(\psi)$

VERIFICATION PROBLEM
Does $M$ satisfy $\Psi$?

SYNTHESIS PROBLEM #1
Synthesize $\phi$ s.t.
$I \Rightarrow \phi \land \psi$
$\phi \land \psi \land \delta \Rightarrow \phi' \land \psi'$

SYNTHESIS PROBLEM #2
Synthesize $\alpha : S \rightarrow \hat{S}$ where
$\alpha(M) = (\hat{I}, \hat{\delta})$
s.t.
$\alpha(M)$ satisfies $\Psi$ iff
$M$ satisfies $\Psi$
Syntax-Guided Synthesis (SyGuS) Problem

- Fix a background theory $T$: fixes types and operations

- Function to be synthesized: name $f$ along with its type
  - General case: multiple functions to be synthesized

- Inputs to SyGuS problem:
  - Specification $\varphi$
    - Typed formula using symbols in $T$ + symbol $f$
  - Set $E$ of expressions given by a context-free grammar that use symbols in $T$

- Computational problem:
  - Output $e$ in $E$ such that $\varphi[f/e]$ is valid (in theory $T$)
SyGuS Example

- Theory QF-LIA
  Types: Integers and Booleans
  Logical connectives, Conditionals, and Linear arithmetic
  Quantifier-free formulas

- Function to be synthesized: \( f \) (int \( x \), int \( y \)) : int

- Specification: \( (x \leq f(x,y)) \land (y \leq f(x,y)) \land (f(x,y) = x \lor f(x,y) = y) \)

- Candidate Implementations: Linear expressions
  \( \text{LinExp} := x \mid y \mid \text{Const} \mid \text{LinExp} + \text{LinExp} \mid \text{LinExp} - \text{LinExp} \)

- No solution exists
SyGuS Example

- Theory QF-LIA

- Function to be synthesized: \(f\) (int \(x\), int \(y\)) : int

- Specification: \((x \leq f(x,y)) \land (y \leq f(x,y)) \land (f(x,y) = x \lor f(x,y) = y)\)

- Candidate Implementations: Conditional expressions with comparisons
  
  Term := x \mid y \mid Const \mid If-Then-Else (Cond, Term, Term)
  
  Cond := Term <= Term \mid Cond \& Cond \mid \sim Cond \mid (Cond)

- Possible solution:
  
  If-Then-Else \((x \leq y, y, x)\)
Solving SyGuS

- Is SyGuS same as solving SMT formulas with quantifier alternation?

- SyGuS can sometimes be reduced to Quantified-SMT, but not always
  - Set E is all linear expressions over input vars x, y
    SyGuS reduces to Exists a,b,c. Forall X. \( \varphi [ f/ ax+by+c] \)
  - Set E is all conditional expressions
    SyGuS cannot be reduced to deciding a formula in LIA

- Existing work on solving Quantified-SMT formulas suggests solution strategies for SyGuS

- In general, SyGuS problems are undecidable unless grammars (and logic) are suitably restricted [Caulfield, Rabe, Seshia, Tripakis, 2015].
**SyGuS as Oracle-Guided Learning**

Concept class: Set E of expressions

Examples: Concrete input values
Counterexample-Guided Inductive Synthesis (CEGIS): Example

- Specification: \((x \leq f(x,y)) \& (y \leq f(x,y)) \& (f(x,y) = x \mid f(x,y) = y)\)

- Set E: All expressions built from \(x, y, 0, 1, \text{Comparison}, +, \text{If-Then-Else}\)

```
Examples = { }  
```

Examples = \{ \}\n
```
Candidate f(x,y) = x
```

```
Example (x=0, y=1)
```

![Diagram](Diagram_CEGIS_Example.png)
CEGIS Example

- Specification: \((x \leq f(x,y)) \land (y \leq f(x,y)) \land (f(x,y) = x \lor f(x,y) = y)\)

- Set E: All expressions built from \(x,y,0,1,\) Comparison, +, If-Then-Else

Examples = \(\{(x=0, y=1)\}\)

Candidate
\(f(x,y) = y\)

Example
\((x=1, y=0)\)
CEGIS Example

- Specification: \((x \leq f(x,y)) \& (y \leq f(x,y)) \& (f(x,y) = x \mid f(x,y) = y)\)

- Set E: All expressions built from \(x, y, 0, 1, \text{Comparison}, +, \text{If-Then-Else}\)

Examples = 
\{(x=0, y=1) \\
(x=1, y=0) \\
(x=0, y=0) \\
(x=1, y=1)\}

Candidate 
ITE (x \leq y, y, x)

Learning Algorithm  
Verification Oracle

Success
SyGuS Solutions

- CEGIS approach (Solar-Lezama, Seshia et al)

- Basic CEGIS learning strategies (circa 2013) based on:
  - Enumerative (search with pruning): Udupa et al (PLDI’13)
  - Symbolic (solving constraints): Jha et al (ICSE’10, PLDI’11)
  - Stochastic (probabilistic walk): Schkufza et al (ASPLOS’13)

- Many new approaches available today (see www.sygus.org)
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

Formal Inductive Synthesis

• Given:
  – Class of Artifacts C
  – Formal specification $\phi$
  – Domain of examples D
  – Oracle Interface $O$

  • Set of (query, response) types

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

• To solve this: Design/Select BOTH Learner and Oracle

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

<table>
<thead>
<tr>
<th>Desired Feature</th>
<th>Tool \ ABC</th>
<th>NuXMV</th>
<th>Boogie</th>
<th>Coq</th>
<th>UCLID</th>
<th>UCLID5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressive Types (bits -&gt; words -&gt; terms)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High Degree of Automation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wide Variety of Verification Methods</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modular Specification &amp; Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Support for Sequential updates (Seq. software)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Support for Concurrent updates (Synchronous HW)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Support for Meaningful Counterexample Generation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Strong**
- **Medium**
- **Weak**
UCLID5 Verifier

Sources (/*.ucl) → Type Checking, Module Instantiation, Composition, etc. → VC Generation, Symbolic Simulation, Model Checking, etc. → Properties satisfied

SMT Solvers (Z3), SyGuS Solvers

Property violated [+ counterexample]
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

```plaintext
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
UCLID5 Demos

• Proving Determinism of a Simple CPU that implements Isolated Memory Regions (oversimplified version of enclaves)

• LTL Bounded Model Checking

• Invariant Checking via SyGuS

• Trusted Abstract Platform Proof in UCLID5
  – Half the model size in Boogie
  – Similar time to verify
A Few Sample Applications of UCLID5

• Verifying secure speculation (absence of “Spectre-Meltdown style” attacks) [Cheang et al., CSF 2019]

• Verifying the Keystone open-source trusted computing platform [work-in-progress]
Transient Execution Attacks

• Transient execution attacks
  – Spectre, Meltdown, and Foreshadow
  – Side-channel leaks

• Mitigations
  – Software mitigations
    • compiler extensions (e.g. /QSpectre), retpolines, page table isolation
  – Hardware mitigations
    • constant time memory loads, cache partitioning, access time randomization, obfuscation of timers

• Problems to be addressed
  – Formulating a general property to capture transient execution attacks
  – Formalizing general attacker models
  – Automated verification of absence of transient execution attacks
Problem Statement: Verifying Secure Speculation

Given a platform model, an adversary model, and a program, determine if the program is vulnerable to a transient execution attack

```c
void victim_function_v01(size_t x) {
    if (x < array1_size) {
        temp &= array2[array1[x] * 512];
    }
}
```

```c
void victim_function_v01(size_t x) {
    if (x < array1_size) {
        std::atomic_thread_fence(std::memory_order_seq_cst);
        temp &= array2[array1[x] * 512];
    }
}
```

Secure Speculation Property

void victim_function_v01(size_t x) {
    if (x < array1_size) {
        temp &= array2[array1[x] * 512];
    }
}

module main {
  instance t1, t2 : program(speculative : (false));
  instance t3, t4 : program(speculative : (true));
  ...
  assume (t1.pc == t2.pc && t3.pc == t4.pc);
  assume (t1.obs == t2.obs);
  invariant same_obs : (t3.obs == t4.obs);
}

000001be: sub victim_function_v01()
00000193:
00000194: v359 := mem[0x201030, el]:u64 - RDI
00000195: CF := mem[0x201030, el]:u64 < RDI
0000019a: ZF := 0 = v359
0000019b: when CF | ZF goto %000001ba
0000019c: goto %0000019d
...

module main {
    instance t1, t2 : program(speculative : (false));
    instance t3, t4 : program(speculative : (true));
    ...
    assume (t1.pc == t2.pc && t3.pc == t4.pc);
    assume (t1.obs == t2.obs);
    invariant same_obs : (t3.obs == t4.obs);
}

Evaluation

- Paul Kocher’s list of 15 bounds check bypass examples
- Vectre transpiler: [https://github.com/kkmc/vectre_transpiler](https://github.com/kkmc/vectre_transpiler)
- Bounded model checking for exploit finding (5 steps)
- Inductive model checking for verification (1 step)

<table>
<thead>
<tr>
<th>Example</th>
<th>Ex1</th>
<th>Ex5</th>
<th>Ex7</th>
<th>Ex8</th>
<th>Ex10</th>
<th>Ex11</th>
<th>Ex15</th>
<th>Fig. 3c</th>
<th>NI</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMC</td>
<td>6.6 secs</td>
<td>9.0 secs</td>
<td>10.2 secs</td>
<td>5.7 secs</td>
<td>9.6 secs</td>
<td>6.4 secs</td>
<td>5.8 secs</td>
<td>6.6 secs</td>
<td>12.9 secs</td>
</tr>
<tr>
<td>Induct</td>
<td>5.0 secs</td>
<td>5.0 secs</td>
<td>5.7 secs</td>
<td>4.6 secs</td>
<td>5.8 secs</td>
<td>5.9 secs</td>
<td>4.8 secs</td>
<td>4.8 secs</td>
<td>5.4 secs</td>
</tr>
</tbody>
</table>
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/SyGuS
  – Supports diverse specification/verification/modeling tasks
  – Supports compositional (modular) reasoning
  – Open source, publicly available
Key References:


