Presented at WoSCA 2020, July 19th

SMT-based effective formalization of reference types in Solidity

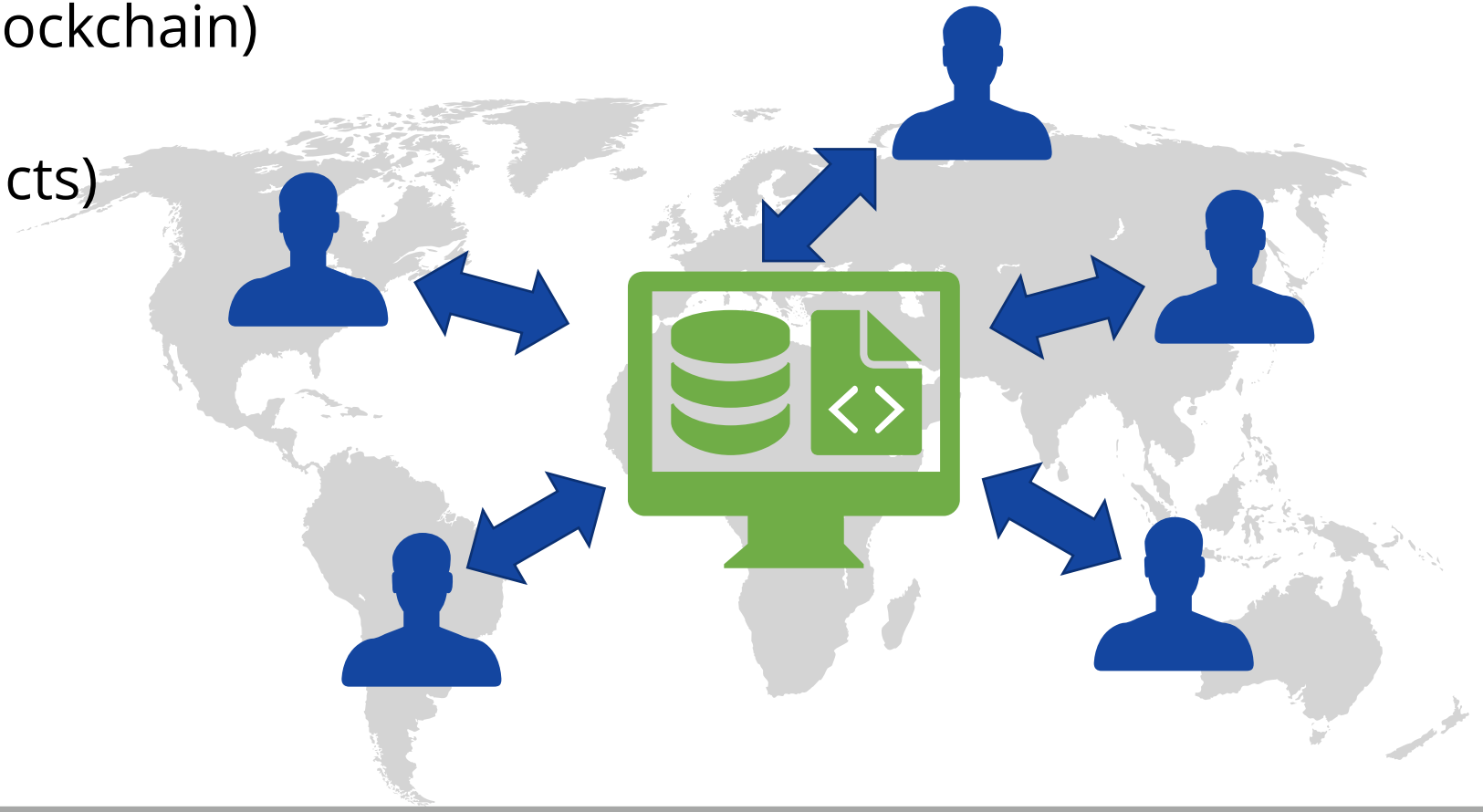
Ákos Hajdu[1], Dejan Jovanović[2]

[1]*Budapest University of Technology and Economics*
[2]*SRI International*

Solidity Smart Contracts

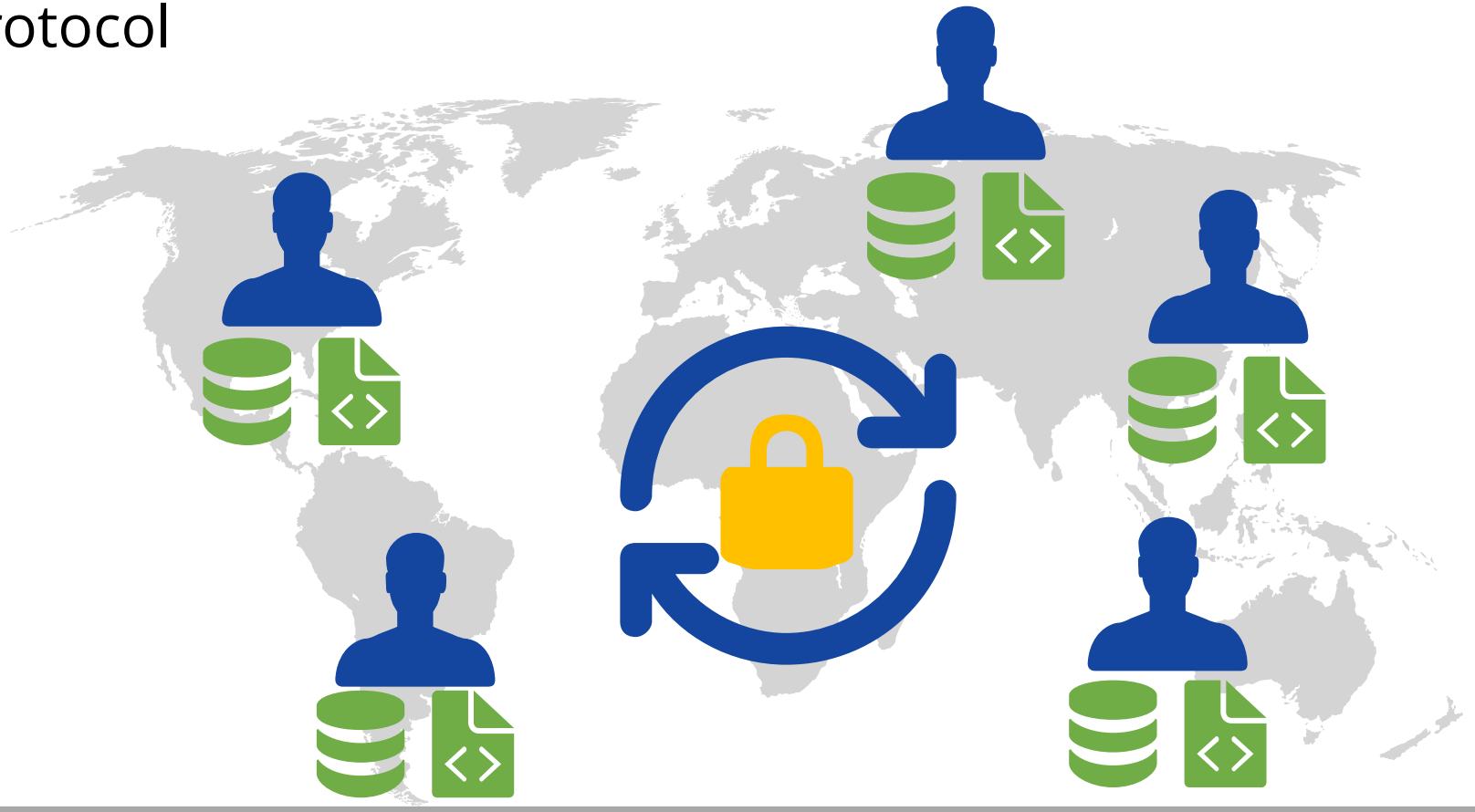
Decentralized Computing Platforms

- Conceptually a single-world-computer abstraction
  - Example: Ethereum
  - Store data (blockchain)
  - Execute code (smart contracts)



Decentralized Computing Platforms

- In reality
  - No trusted central party
  - Consensus protocol



Solidity Smart Contracts

```
contract DataStorage {
  struct Record { bool set; int[] data; }

  mapping(address=>Record) private records;

  function append(address at, int d) public {
    Record storage r = records[at];
    r.set = true;
    r.data.push(d);
  }
  function get(address at) public view returns (int[] memory ret) {
    require(isset(records[at]));
    ret = records[at].data;
  }
  function isset(Record storage r) internal view returns (bool s) {
    s = r.set;
  }
}
```

Complex datatype

State variable(s): permanent storage (blockchain)

Function(s): called with transactions

Parameters, return values, local variables in transient memory

Pointers to storage in internal scope

Verification Landscape

- Bytecode-level tools
  - Slither, Mythril, KEVM, …
  - Various formalizations
  - Mostly vulnerable patterns
    - Limited effectiveness and automation for high-level properties
- Solidity-level tools
  - SMTchecker, solc-verify, VeriSol, KSolidity, …
  - High-level, functional properties
  - Usually based on SMT
    - Modular verification, bounded model checking, symbolic execution
  - Precise formalization required

Memory model lacks detailed and effective formalization

Target Language

- Simple SMT-based program
  - Types: primitive, datatype, array
  - Variable declarations
  - Statements: assign, assume, if-then-else
  - Expressions: identifier, array read/write, datatype constructor, member selector, conditional, basic arithmetic
- Can be expressed in any SMT-based tool
  - Boogie, Why3, Dafny, …
  - Check by translating to SSA

```
Point(x : int, y : int)

pts : [int]Point

pts[0] := Point(1, 2)
pts[1].x := pts[0].x + 1
```

Formalizing the Solidity memory model

Overview

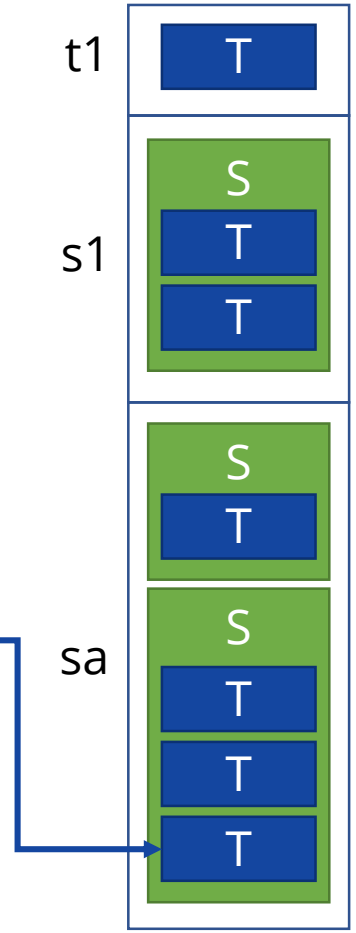
- Storage: value semantics

```
contract C {
  struct S { int x; T[] ta; }
  struct T { int z; }

   T   t1;
   S   s1;
   S[] sa;
}
```

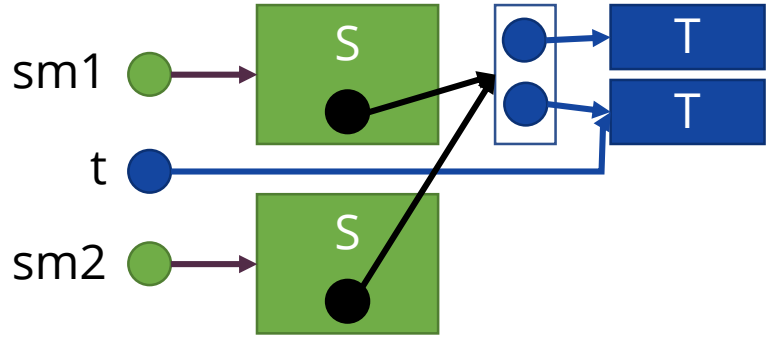
- Local storage pointers

```
function f() public {
  T storage tp = sa[1].ta[2];
  g(tp);
}
function g(T storage t) internal {
  t.z = 5;
}
```



- Memory: reference semantics

```
function f() public pure {
   S memory sm1 = S(1, new T[](2));
   T memory t = sm.ta[1];
   S memory sm2 = S(2, sm1.ta);
}
```



Encoding the Memory

- Standard heap model (per type)
  - Pointer: SMT integer
  - Struct: SMT datatype
  - Array: SMT array + length
  - No null, default values recursively

```
S memory sm1 = S(1, new T[](2));
```

```
sm1 : int
heap_T[0] := T_mem(0)
heap_T[1] := T_mem(0)
heap_TA[2] := T_memarr([0, 1], 2)
heap_S[3] := S_mem(1, 2)
sm1 := 3
```

Allocation counter

```
struct T { int z; }
struct S { int x; T[] ta; }
```

```
T_mem(z : int)
T_memarr(arr : [int]int, len : int)
S_mem(x : int, ta : int)

heap_T  : [int]T_mem
heap_TA : [int]T_memarr
heap_S  : [int]S_mem
```

```
sm1.ta[1].z;
```

```
heap_T[heap_TA[heap_S[3].ta].arr[1]].z
```

Encoding the Memory

```
struct T { int z; }
struct S { int x; T[] ta; }
```

- Scope limited to a single transaction
- Non-aliasing and new allocations
  - Require quantifiers in the general case (decidable fragment)

```
function f(S memory sm) {
  ... = S(...)
}
```

New allocations should not alias with sm

Allocation counter

sm

sm.ta

sm.ta[i] (for each i)

```
assume(sm < refcnt)
assume(heap_S[sm].ta < refcnt)
forall 0 <= i < heap_TA[heap_S[sm].ta].len:
  assume(heap_TA[heap_S[sm].ta].arr[i] < refcnt)
```

Encoding the Storage

- Encode with SMT datatypes without heaps
  - Non-aliasing and deep copy ensured out-of-the-box
  - Especially useful in modular verification
    - Otherwise many framing conditions for functions

```
struct T { int z; }
struct S { int x; T[] ta; }
```

```
contract C {
  T   t1;
  S   s1;
  S[] sa;
}
```

T_{stor}(z : int)
$T_{storarr}$(arr : [int]T_{stor}, len : int)
S_{stor}(x : int, ta : $T_{storarr}$)
$S_{storarr}$(arr : [int]S_{stor}, len : int)

t1: T_{stor}
s1: S_{stor}
sa: $S_{storarr}$

Local storage pointers?

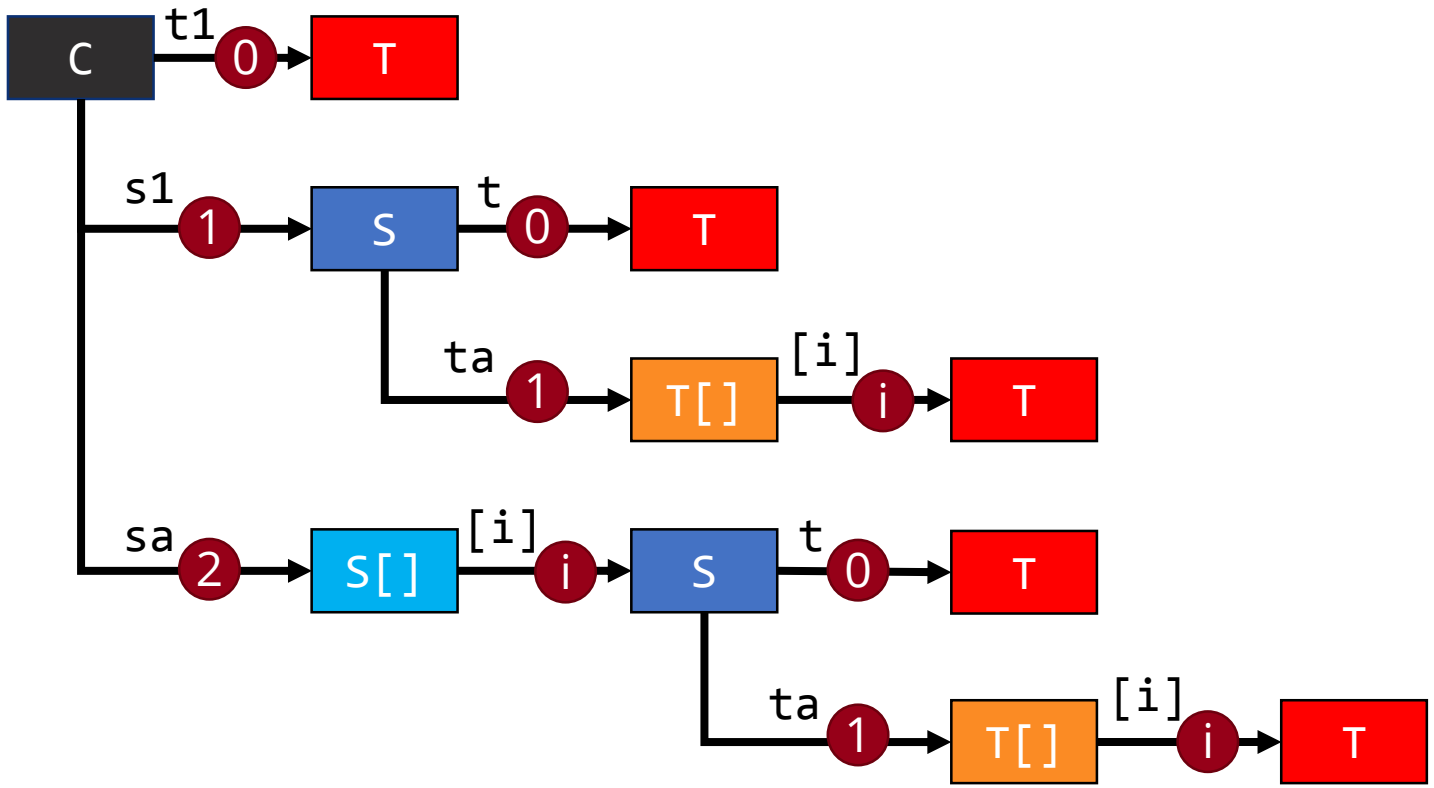
*Limited support for recursive data types, not used in practice.

Local Storage Pointers

- Storage is a finite-depth tree of values*
- Each element identified by path → encode with SMT integer array

```
contract C {
  struct T {
    int z;
  }
  struct S {
    int x;
    T   t;
    T[] ta;
  }

  T   t1;
  S   s1;
  S[] sa;
}
```



Local Storage Pointers

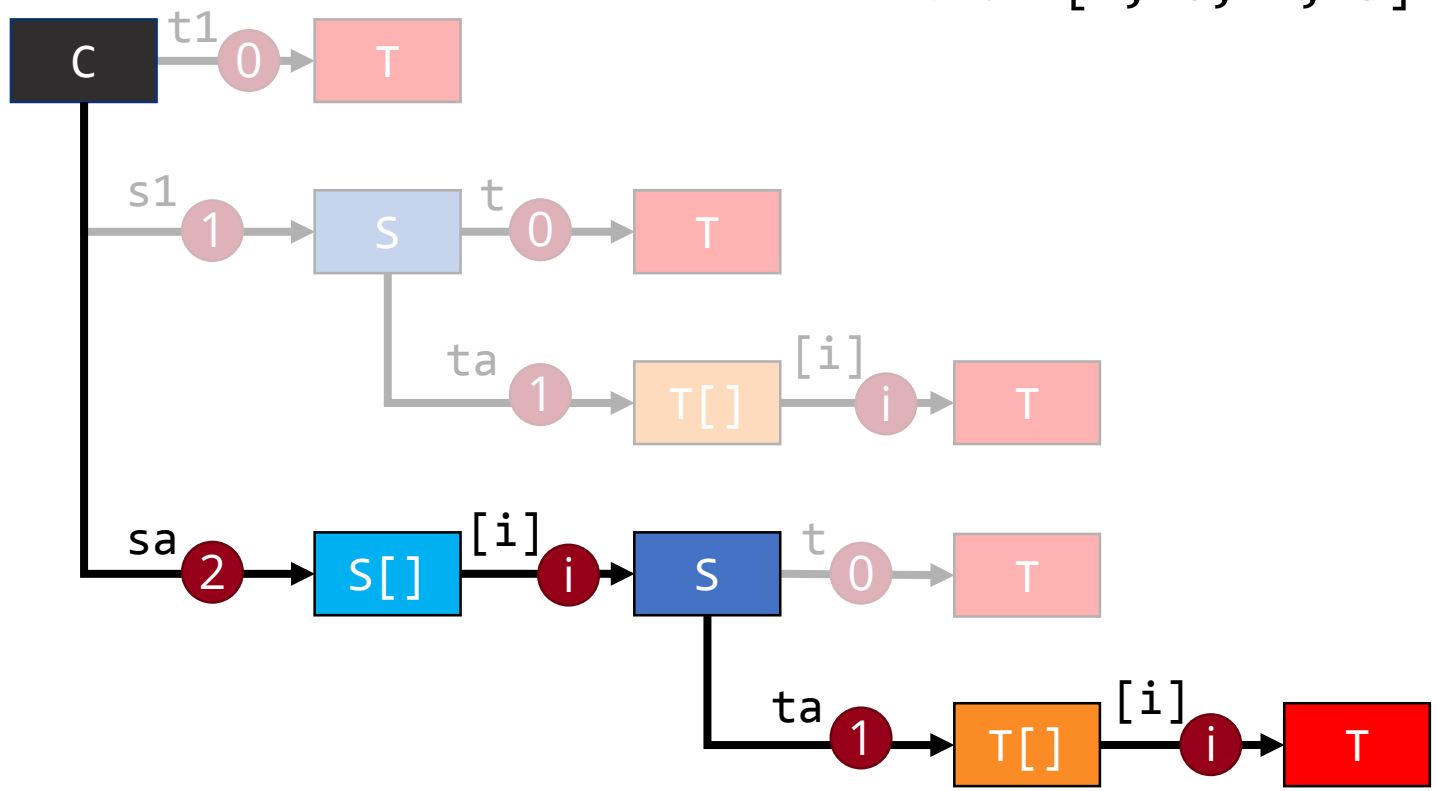
- Packing: expression to SMT array
  - Fit expression to tree

```
T storage t = sa[8].ta[5];
```

```
t : [int]int
t := [2, 8, 1, 5]
```

```
contract C {
  struct T {
    int z;
  }
  struct S {
    int x;
    T   t;
    T[] ta;
  }

  T   t1;
  S   s1;
  S[] sa;
}
```



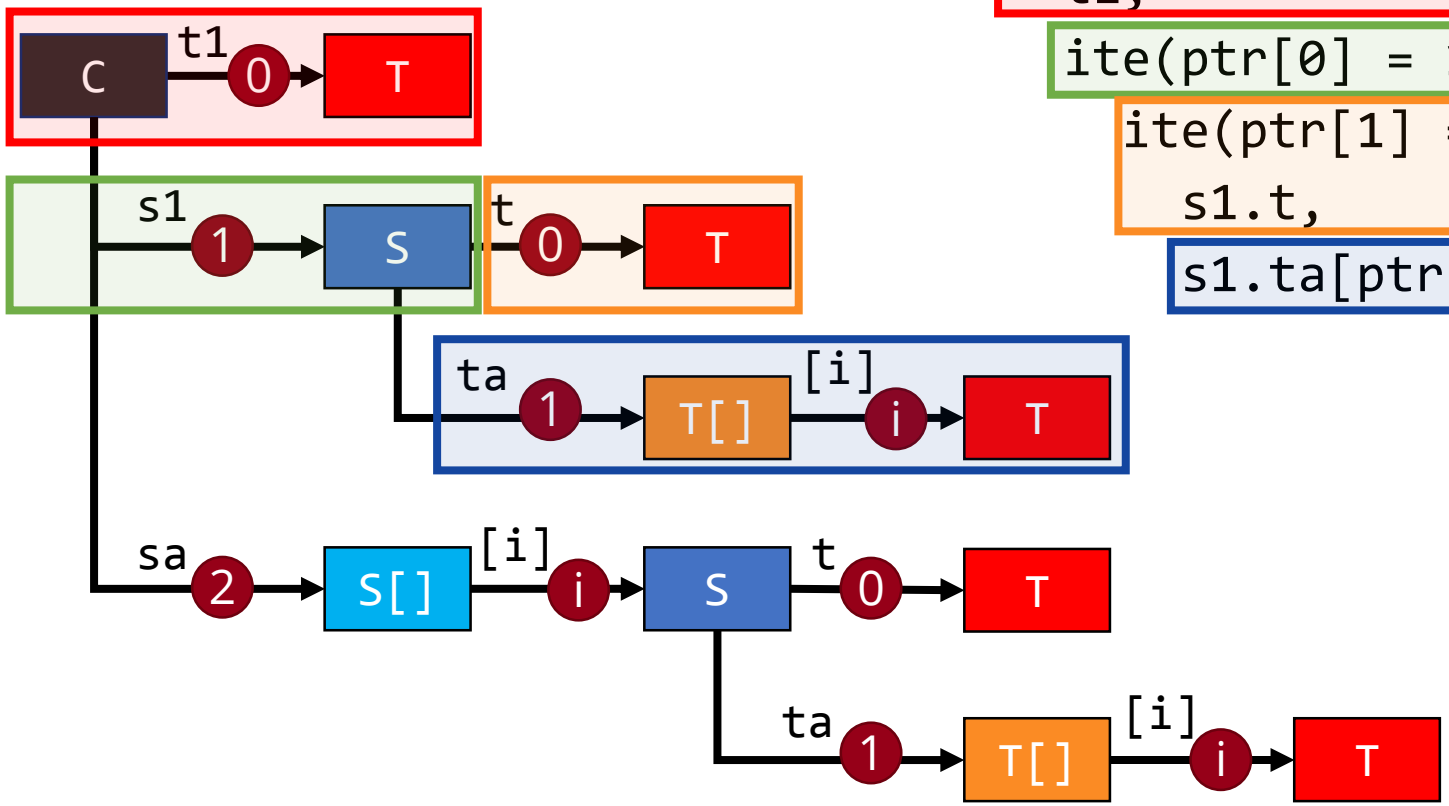
Local Storage Pointers

- Unpacking: SMT array to expression
  - Conditional based on tree

```
function f(T storage ptr) {
  ... ptr.z;
}
```

```
contract C {
  struct T {
    int z;
  }
  struct S {
    int x;
    T   t;
    T[] ta;
  }

  T   t1;
  S   s1;
  S[] sa;
}
```




```
ite(ptr[0] = 0,
  t1,
  ite(ptr[0] = 1,
    ite(ptr[1] = 0,
      s1.t,
      s1.ta[ptr[2]]), … ).z
```

Assignments Between Data Locations

LHS/RHS	Storage	Memory	Storage ptr.
Storage	Deep copy	Deep copy	Deep copy
Memory	Deep copy	Pointer assign	Deep copy
Storage ptr.	Pointer assign	Error	Pointer assign

- Ensured by construction
- Ensured by pack/unpack
- Need manual copying
  - Requires quantifiers in the general case

Details in Paper

ESOP 2020 - arxiv.org/abs/2001.03256

$$\mathcal{T}(\texttt{bool}) \doteq \mathit{bool}$$
$$\mathcal{T}(\texttt{address}) \doteq \mathcal{T}(\texttt{int}) \doteq \mathcal{T}(\texttt{uint}) \doteq \mathit{int}$$
$$\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storage}) \doteq [\mathcal{T}(K)]\mathcal{T}(V)$$
$$\mathcal{T}(\texttt{mapping(}K\texttt{=>}V\texttt{) storptr}) \doteq [\mathit{int}]\mathit{int}$$
$$\mathcal{T}(T\texttt{[}n\texttt{] storage}) \doteq \mathcal{T}(T\texttt{[] storage})$$
$$\mathcal{T}(T\texttt{[}n\texttt{] storptr}) \doteq \mathcal{T}(T\texttt{[] storptr})$$
$$\mathcal{T}(T\texttt{[}n\texttt{] memory}) \doteq \mathcal{T}(T\texttt{[] memory})$$

$$\mathcal{S}[\![T\ id]\!] \doteq [id : \mathcal{T}(T)];\ \mathcal{A}(id, \mathsf{defval}(T))$$
$$\mathcal{S}[\![T\ id = expr]\!] \doteq [id : \mathcal{T}(T)];\ \mathcal{A}(id, \mathcal{E}(expr))$$
$$\mathcal{S}[\![\texttt{delete}\ e]\!] \doteq \mathcal{A}(\mathcal{E}(e), \mathsf{defval}(\mathsf{type}(e)))$$
$$\mathcal{S}[\![l_1, \ldots, l_n = r_1, \ldots, r_n]\!] \doteq [tmp_i : \mathcal{T}(\mathsf{type}(r_i))] \text{ for } 1 \leq i \leq$$
$$\mathcal{A}(tmp_i, \mathcal{E}(r_i)) \quad \text{for } 1 \leq i \leq$$
$$\mathcal{A}(\mathcal{E}(l_i), tmp_i) \quad \text{for } n \geq i \geq$$
$$\mathcal{S}[\![e_1\texttt{.push(}e_2\texttt{)}]\!] \doteq \mathcal{A}(\mathcal{E}(e_1).arr[\mathcal{E}(e_1).length], \mathcal{E}(e_2))$$
$$\mathcal{E}(e_1).length := \mathcal{E}(e_1).length + 1$$
$$\mathcal{S}[\![e\texttt{.pop()}]\!] \doteq \mathcal{E}(e).length := \mathcal{E}(e).length - 1$$
$$\mathcal{A}(\mathcal{E}(e).arr[\mathcal{E}(e).length], \mathsf{defval}(\mathsf{arrtype}(\mathcal{E}(e))))$$

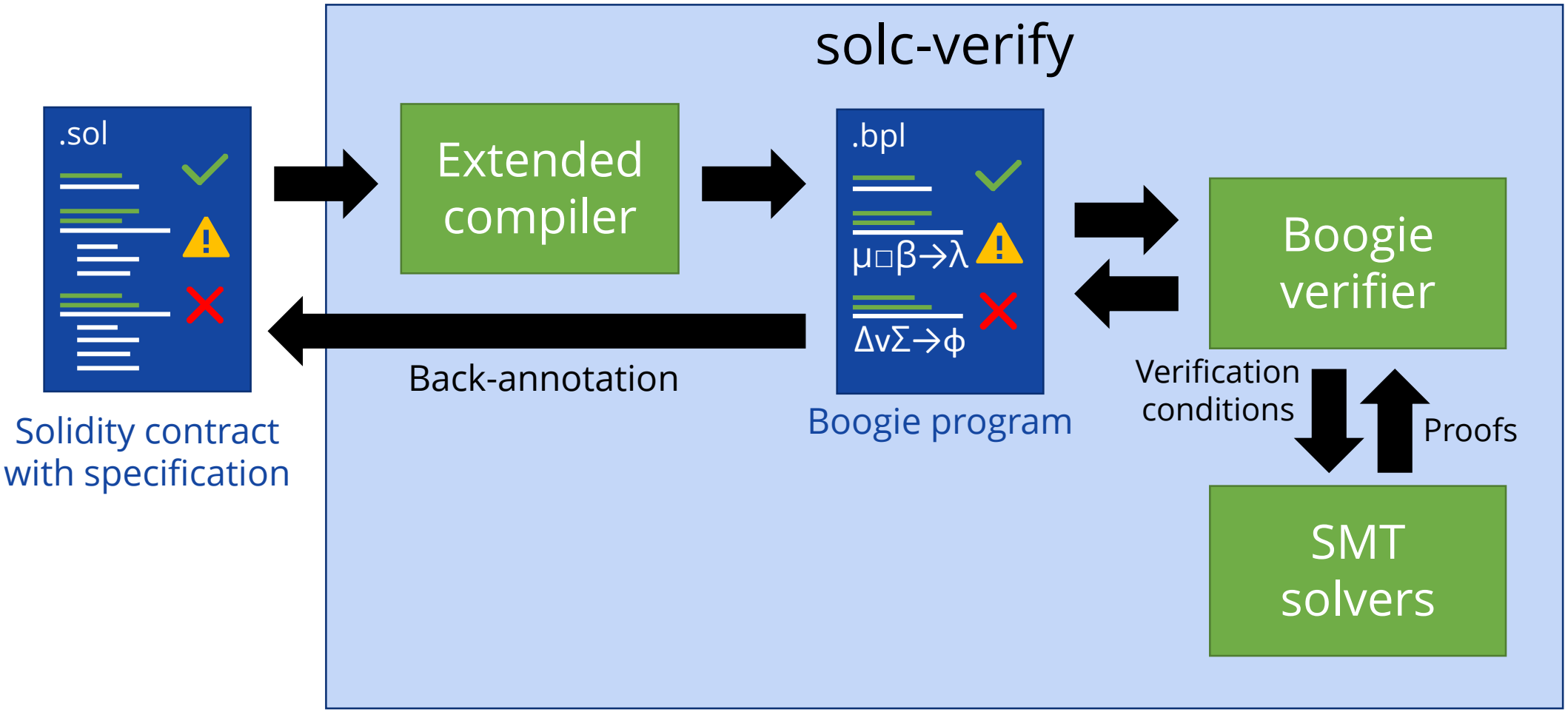
$$\mathcal{A}_S(lhs : \texttt{s}, rhs : \texttt{s}) \doteq lhs := rhs$$
$$\mathcal{A}_S(lhs : \texttt{s}, rhs : \texttt{m}) \doteq \mathcal{A}(lhs.m_i, structheap_{\mathsf{type}(rhs)}[rhs].m_i) \text{ for each } m_i$$
$$\mathcal{A}_S(lhs : \texttt{s}, rhs : \texttt{sp}) \doteq \mathcal{A}_S(lhs, \mathsf{unpack}(rhs))$$
$$\mathcal{A}_S(lhs : \texttt{m}, rhs : \texttt{m}) \doteq lhs := rhs$$
$$\mathcal{A}_S(lhs : \texttt{m}, rhs : \texttt{s}) \doteq lhs := refcnt := refcnt + 1$$
$$\mathcal{A}(structheap_{\mathsf{type}(lhs)}[lhs].m_i, rhs.m_i) \text{ for each } m_i$$
$$\mathcal{A}_S(lhs : \texttt{m}, rhs : \texttt{sp}) \doteq \mathcal{A}_S(lhs, \mathsf{unpack}(rhs))$$
$$\mathcal{A}_S(lhs : \texttt{sp}, rhs : \texttt{s}) \doteq lhs := \mathsf{pack}(rhs)$$
$$\mathcal{A}_S(lhs : \texttt{sp}, rhs : \texttt{sp}) \doteq lhs := rhs$$

```
def unpack(ptr):
    return unpack(ptr, tree(type(ptr)), empty, 0);
def unpack(ptr, node, expr, d):
    result := empty;
    if node has no outgoing edges then result := expr;
    if node is contract then
        foreach edge node --id (i)--> child do
            result := ite(ptr[d] = i, unpack(ptr, child, id, d + 1), result);
    if node is struct then
                                                  → child do
                                                              );
```

Tool & Evaluation

Implementation

- github.com/SRI-CSL/solidity
- arxiv.org/abs/1907.04262
- youtu.be/1q2gSm3NuQA



Specification Annotations

- Solidity provides
  - require, assert
- Annotation language
  - Features
    - Pre/postconditions
    - Contract level invariants
    - Loop invariants
    - Access control (modifies)
    - Events *in progress*
  - Solidity expressions (side effect free)
    - Extra: sum over collections
    - Quantifiers *in progress*

FMBC talk tomorrow!
fmbc.gitlab.io/2020

```
/// @notice invariant x == y
contract C {
  int x;
  int y;

  /// @notice precondition x == y
  /// @notice postcondition x == (y + n)
  /// @notice modifies x
  function add_to_x(int n) internal {
    x = x + n;
    require(x >= y);
  }

  /// @notice modifies x if n > 0
  /// @notice modifies y if n > 0
  function add(int n) public {
    require(n >= 0);
    add_to_x(n);
    /// @notice invariant y <= x
    while (y < x) {
      y = y + 1;
    }
  }
}
```

Compared Tools

- solc-verify (our tool) github.com/SRI-CSL/solidity
  - Modular verifier based on Boogie/SMT and the presented encoding
- Mythril github.com/ConsenSys/mythril
  - Symbolic execution engine running over bytecode
- VeriSol github.com/microsoft/verisol
  - Modular/BMC tool based on Boogie/SMT
  - Heap-based modeling of both memory and storage
- SMTchecker github.com/ethereum/solidity
  - SMT-based intra-function analyzer built into the compiler

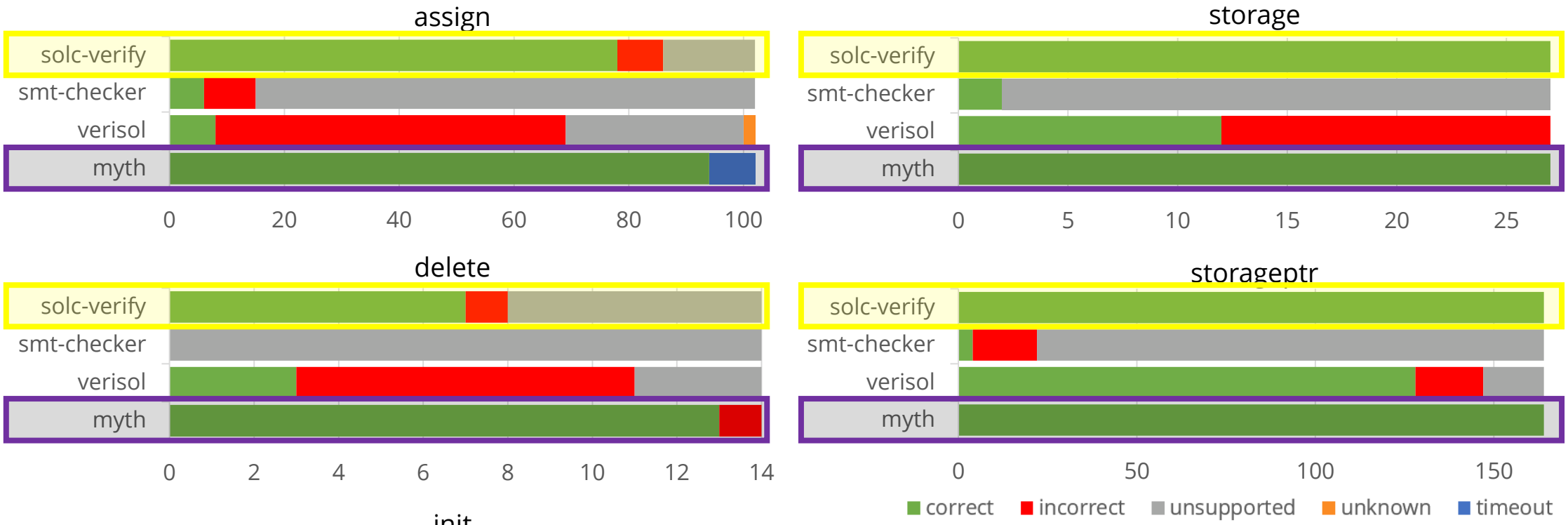
Tests

- "Real world" contracts: limited for evaluating memory semantics
  - Many old versions, new features are rare
  - Many toy examples, overrepresented categories
  - Complex contracts depend on other features

- Manually developed tests
  - 325 test cases organized into categories
    - Assign, delete, init, storage, storage pointer
  - Exercise a specific feature, check result with assertion

```
contract InitMemoryArrayFixedSize {
  function test() public pure {
    int[2] memory a;
    assert(a.length == 2);
    assert(a[0] == 0);
    assert(a[1] == 0);
   }
}
```

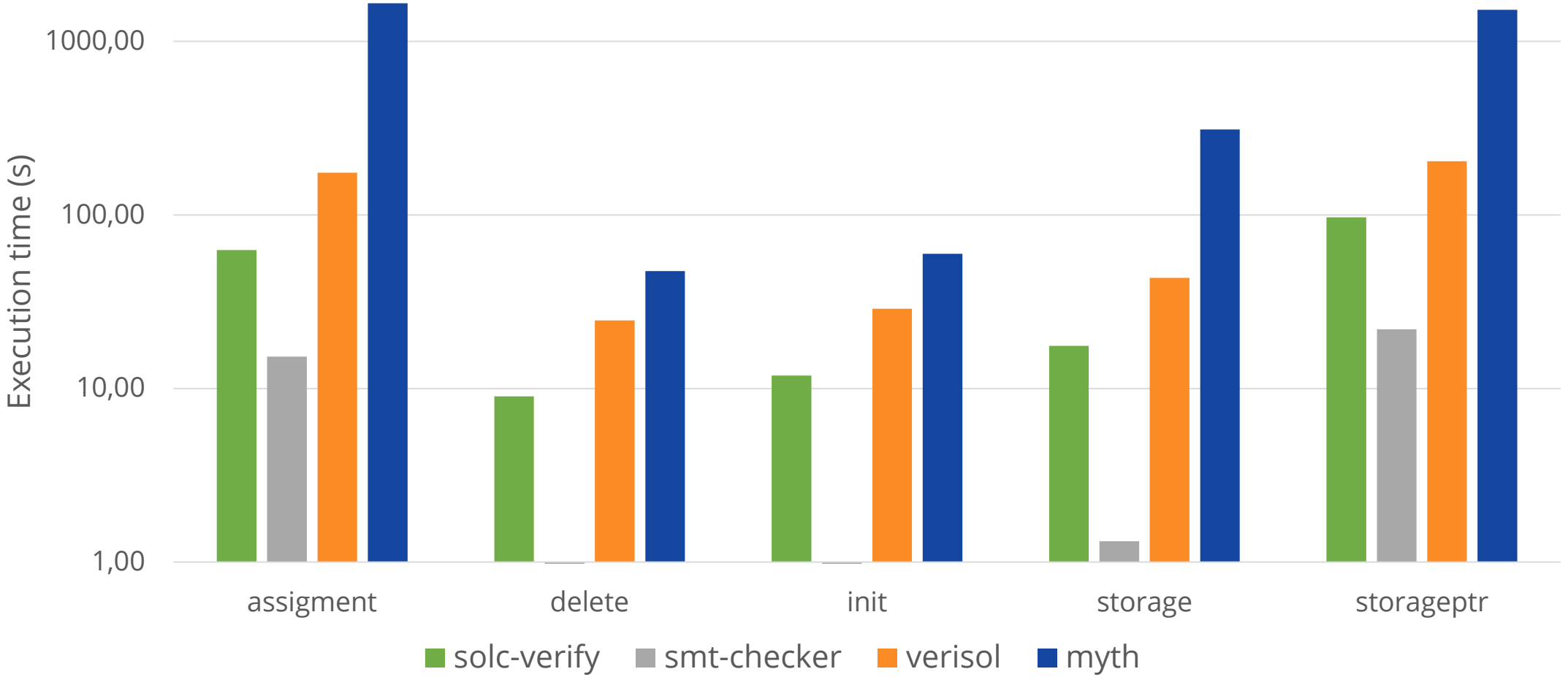
Results



- Bytecode level is precise
- solc-verify comes close
  - Some unimplemented features

Mythril bug report: github.com/ConsenSys/mythril/issues/1282

Results

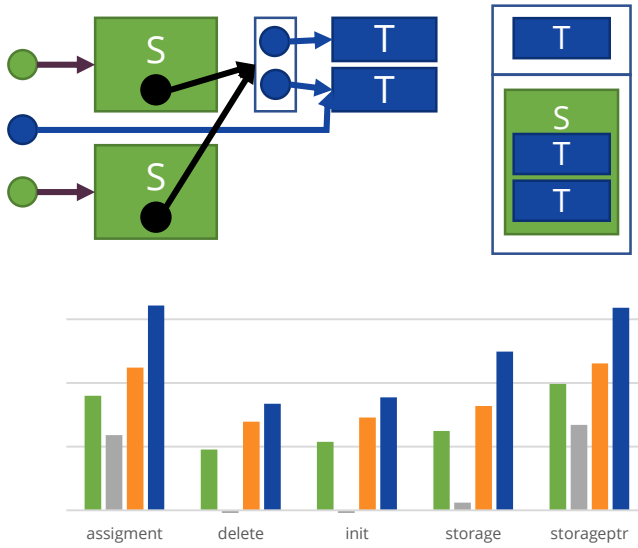


- Low computational cost for solc-verify

Summary

Summary

- SMT-based formalization of Solidity reference types
  - Memory: standard heap
  - Storage: values
  - Local storage pointers: encode path



- Implementation
  - solc-verify: modular verifier
  - Extensive set of test cases
  - Effective: on par with bytecode-level tools, at low computational cost

github.com/SRI-CSL/solidity
arxiv.org/abs/2001.03256
arxiv.org/abs/1907.04262

hajduakos.github.io
csl.sri.com/users/dejan

ftsrg