A Trusted Safety Verifier for Process Controller Code

Stephen McLaughlin, Devin Pohly, Patrick McDaniel and Saman Zonouz
February 24, 2014
Control Systems Under Threat

Programmable Controllers are Insecure By Design
Programmable Controllers are Insecure By Design

Control systems not robust enough for security patches
Control Systems Under Threat

Programmable Controllers are Insecure By Design

Control systems not robust enough for security patches
A new perspective on SCADA

Supervisory Control and Data Acquisition
A new perspective on SCADA

Tens of Millions of Files

Hundreds of Millions of LoC

Anything cloud
A new perspective on SCADA

- Tens of Millions of Files
- Hundreds of Millions of LoC
- Anything cloud

Far smaller codebase

- Hundreds of Devices
We would like to directly protect the physical process, regardless of the integrity of the IT perimeter.
A PLC’s life
A PLC’s life

Turn left 2°
A PLC’s life

Turn left 2°

Turn left!
A PLC’s life

Turn left 2°

Turn left!
A PLC’s life

???

Turn left!
A PLC’s life

Turn left 2°

Turn left!
A PLC’s life

Turn left 2°

Turn left!
A PLC’s life

Not quite …

Turn left!

1°
A PLC’s life

Not quite …

Turn left!

2°
A PLC’s life

BINGO!

Stop!

2°
A PLC’s life
A PLC’s life

Control Logic

L DW2 ;; Degrees
< ID255
ST Q 3.2
A PLC’s life

Control Logic

L DW2 ;; Degrees
< ID255
ST Q 3.2
A PLC’s life

Control Logic

L DW2 ;; Degrees
< ID255
ST Q 3.2

Control Signal

Sensor Measurements
A PLC’s life

Control Logic

```
L DW2     ;; Degrees
< ID255
ST Q 3.2
```

Control Signal

Scan Cycle

Sensor Measurements
A PLC’s life

Applications

- Chemical processing
- Railroad safety
- Manufacturing
- Traffic Control
- PID Control
A PLC’s life

L DW2 ;; Degrees
< ID255
ST Q 3.2

;;; Spin out
;;; of control!
S Q 3.2
A PLC’s life

L DW2 ;; Degrees
< ID255
ST Q 3.2

;; Spin out
;; of control!
S Q 3.2
A PLC’s life

```
L DW2 ;; Degrees
< ID255
ST Q 3.2

;; Spin out
;; of control!
S Q 3.2
```

“TSV”

```
;;;; Spin out
;;;; of control!
```

“The Arm must remain within a quarter hemisphere at all times”
Trusted Safety Verifier

• **Goal**: Only allow code to be run on a PLC if it satisfies a set of engineer-supplied safety properties.

• **Challenges**:
  - Existing tools not up to the task.
  - Control systems are *stateful*, requiring temporal properties.
  - State space explosion with existing analysis techniques.
Consider some PLC code

A I 0.5 ;; And input bit 5
= Q 0.1 ;; Store at output bit 1
...
Consider some PLC code

A I 0.5 ;; And input bit 5
= Q 0.1 ;; Store at output bit 1
...

- Side effects
Consider some PLC code

```
A I 0.5 ;; And input bit 5
= Q 0.1 ;; Store at output bit 1
...```

- Side effects
- PLC special features
Consider some PLC code

A I 0.5 ;; And input bit 5
= Q 0.1 ;; Store at output bit 1
...

- Side effects
- PLC special features
- Architecture dependent
Consider some PLC code

A I 0.5 ;; And input bit 5
= Q 0.1 ;; Store at output bit 1

- Side effects
- PLC special features
- Architecture dependent

Instruction List
Intermediate Language

// A I 0.5
STA := load(mem, [I::0::0::0::5]);
cjmp FC == 0 : reg1_t, L1, L2;
label L1;
RLO := STA;
label L2;
RLO := RLO && STA;
FC := 1 : reg1_t;

// = Q 0.1
STA := RLO;
mem := store(mem, [Q::0::0::0::1], RLO);
FC := 0 : reg1_t;
Based on the Vine IL

\[
\text{prog ::= } \text{inst}^\star \text{fun}^\star \\
\text{fun ::= ident(var)\{inst\}^*} \\
\text{inst ::= jmp e, e, e | label ident | ident ::= e} \\
\text{ | call ident(var=e) | ret | assert e} \\
\text{e ::= load(ident, addr) | store(ident, addr, int) | e binop e} \\
\text{ | unop e | var | val | (e)} \\
\text{binop ::= +, -, *, /, mod, &, &&, <<, \ldots} \quad \text{(And signed versions.)} \\
\text{unop ::= – (Negate), ~ (Bitwise), ! (Boolean)} \\
\text{var ::= ident (: \tau)} \\
\text{val ::= mem } \boxed{\text{addr}} \text{ | int (: \tau)} \\
\text{mem ::= \{addr \mapsto \text{int}, addr \mapsto \text{int}, \ldots\}} \\
\boxed{\text{addr ::= [int :: int :: \ldots]}} \\
\tau ::= \text{reg1_t} \ldots \text{reg64_t} \quad \boxed{\text{mem_t(int)}} \quad \boxed{\text{addr_t}}
\]
ILIL Analysis

// A I 0.5
STA := load(mem, [I::0::0::0::5]);
cjmp FC == 0 : reg1_t,L1,L2;
label L1;
RLO := STA;
label L2;
RLO := RLO && STA;
FC :=1 : reg1_t;

// = Q 0.1
STA := RLO;
mem := store(mem, [Q::0::0::0::1], RLO);
FC := 0 : reg1_t;

(assert ((and (not (and X_3 X_2))
            (not X_5)))) :
[X_5] -> (X_2)
[X_7] -> (and (and X_1 true)
               (not X_3))
[X_0] -> (X_2)
Systems and Internet Infrastructure Security Laboratory (SIIS)

ILIL Analysis

// A I 0.5
STA := load(mem, [I::0::0::0::5]);
c jmp FC == 0 : reg1_t, L1, L2;
label L1;
RLO := STA;
label L2;
RLO := RLO && STA;
FC := 1 : reg1_t;

// = Q 0.1
STA := RLO;
mem := store(mem, [Q::0::0::0::0::1], RLO);
FC := 0 : reg1_t;

Symbolic Sensor Values

(assert ((and (not (and X_3 X_2))
           (not X_5)))) :
[X_5] -> (X_2)
[X_7] -> (and (and X_1 true)
              (not X_3))
[X_0] -> (X_2)
ILIL Analysis

// A I 0.5
STA := load(mem, [I::0::0::0::5]);
cjmp FC == 0 : reg1_t,L1,L2;
label L1;
RLO := STA;
label L2;
RLO := RLO && STA;
FC :=1 : reg1_t;

// = Q 0.1
STA := RLO;
mem := store(mem, [Q::0::0::0::1], RLO);
FC := 0 : reg1_t;

Symbolic Control Signal

(assert ((and (not (and X_3 X_2))
            (not X_5)))
  [X_5] -> (X_2)
  [X_7] -> (and (and X_1 true)
                (not X_3))
  [X_0] -> (X_2)
(assert ((and (not (and X_3 X_2))
    (not X_5)))

[X_5] -> (X_2)
[X_7] -> (and (and X_1 true)
    (not X_3))
[X_0] -> (X_2)

G (!a)
a: (and X_0 X_5)
Checking Temporal Properties

Linear Temporal Logic

\[ \varphi ::= \text{true} \mid b \mid \lnot \varphi \mid \varphi \lor \varphi \mid \varphi \mathbin{U} \varphi \mid \mathbf{X} \varphi, \]

\[
G (\neg a) \\
a: (\text{and } X_0 X_5)
\]

(assert ((and (not (and X_3 X_2))
           (not X_5)))
   (X_5) -> (X_2)
[X_7] -> (and (and X_1 true)
           (not X_3))
[X_0] -> (X_2)
Checking Temporal Properties

(assert ((and (not (and X_3 X_2))
       (not X_5))) :
[X_5] -> (X_2)
[X_7] -> (and (and X_1 true)
       (not X_3))
[X_0] -> (X_2)

G (!a)
a: (and X_0 X_5)
Checking Temporal Properties

• Input Variables
• State Variables

\[
\text{(assert ((and (not (and X_3 X_2)) (not X_5))) : [X_5] -> (X_2))}
\]

\[
\text{[X_7] -> (and (and X_1 true) (not X_3))}
\]

\[
\text{[X_0] -> (X_2)}
\]

\[
\text{G (!a)}
\]

\[
\text{a: (and X_0 X_5)}
\]

\[
\text{(assert ((and (not (and X_3 X_2)) (not X_5))) : [X_5] -> (X_2))}
\]

\[
\text{[X_5] -> (X_2)}
\]

\[
\ldots
\]

???

???

???

???
Checking Temporal Properties

- Input Variables
- State Variables

Reachability determined by SMT solver

G (!a)
a: (and X_0 X_5)
Temporal Execution Graph (TEG)

The green light itself is then activated by the statement:

\[ AM_0.0 ;; \text{Check for green state.} \]
\[ =Q_0.2 ;; \text{Activate green light.} \]

The lifted version of above two lines of code is as follows.

\[
// (9) AND M_0.0  \\
\text{STA} := \text{cast(low, reg1_t, load(mem, [M::0::0::0::0])};  \\
\text{RLO} := \text{RLO} && \text{STA};  \\
\text{FC} := 1 : \text{reg1_t};  \\
\text{OR} := 0 : \text{reg1_t};  \\
// (10) ST Q_0.2  \\
\text{OR} := 0 : \text{reg1_t};  \\
\text{STA} := \text{RLO};  \\
\text{FC} := 0 : \text{reg1_t};  \\
\text{mem} := \text{store(mem, [Q::0::0::0::2], \text{RLO});  \\
\]

The resulting symbolic scan cycle constraint for the green light is as follows.

\[
\text{// The green output variable.}  \\
\text{(declare-const Q_0_0_0_2 Bool)}  \\
\text{// The state variable.}  \\
\text{(declare-const M_0_0_0_0 Bool)}  \\
\text{[M_0_0_0_0]} -> (\text{and} (\text{or} (\text{or} T_6 M_0_0_0_6) \text{(not} M_0_0_0_0)) \text{(not} T_1))  \\
\text{[Q_0_0_0_2]} -> (M_0_0_0_0)  \\
\]

Consequently, TSV made use of the produced symbolic scan cycle to generate its corresponding temporal execution graph with 24 states that is partially shown in Figure 10. Here, we also show how the usage of symbolic state matching to avoid creation of equivalent states helps TSV to save the TEG memory requirement and consequently improve the overall TSV performance. Figure 11 illustrates the generated TEG graph with 12 states partially for the same controller program while the symbolic state matching engine was on. As shown, several states in Figure 10 have been lumped together in Figure 11 as the result of being equivalent. Because of such state lumpings, there are several states with more than one incoming transitions. The generated TEG graphs with larger model generation bounds resulted in the same growth pattern of 4 states per depth, i.e., the graph size grows linearly for this particular case with the model generation bound because of condition-free controller program.

Finally, TSV employed the generated TEG graph (Figure 11) to verify whether the safety requirement holds if the abovementioned controller program runs on a PLC. The following shows the model checking results to check whether both of the green lights can be on at the same time, i.e.,

\[ G \neg (a \& b) \]

where \( a = (Q_{0 0 0 2} = 0) \) and \( b = (Q_{0 0 0 5} = 0) \).

-- specification \( G \neg (a \& b) \) is false

-- as demonstrated by the following execution sequence

Trace Description: LTL Counterexample
Performance

TEG Depth bounded at 14

![Bar chart showing time (s) for different approaches: Lifting, SymEx, TEG1, TEG2,3, Trans, NuSMV2. The x-axis represents the approaches, and the y-axis represents time in seconds. The chart illustrates the relative time taken by each approach, with TEG2,3 and NuSMV2 taking the most time.]
Performance

Figure 8. State Space Size for All Case Studies on Raspberry Pi.

Traffic Light

Table of Contents

I. Introduction
II. Related Work
III. System Architecture
IV. Security Analysis
V. Experimental Results
VI. Conclusion

Figure 7 shows how much each analysis step contributes to the total overhead. The overhead includes the time required for lifting, symbolic execution, model checking, and verification. The analysis steps are labeled as:

- Lifting
- SymEx
- TEG1
- TEG2,3
- Trans
- NuSMV2

The figure shows the time requirement for each step, with Traffic Light as a sample case study.

For the Traffic Light case study, the overhead is within reasonable bounds for all steps. Despite the variance between use cases, it is clear that the overhead is less than 10 and 90 seconds, on the desktop and Raspberry Pi respectively. In summation, the total average overheads of less than 10 and 90 seconds, on the desktop and Raspberry Pi respectively. In summation, the total average overheads of less than 10 and 90 seconds, on the desktop and Raspberry Pi respectively. In summation, the total average overheads of less than 10 and 90 seconds, on the desktop and Raspberry Pi respectively.
Performance

Figure 6 shows a sample generated execution graph for the AssemblyWay case study. The graph illustrates the symbolic execution process, with node labels representing different states and transitions. The figure highlights the large symbolic scan cycle, which is a significant factor in the total analysis time.

Figure 7 shows the time requirements for all case studies on Raspberry Pi. The chart includes labels for different steps in the model checking process: Initial Model Creation, Symbolic Model Checking, and Translation of the Temporal Execution Graph. The time is measured in seconds, with bars indicating the duration for each step. The results indicate that the time requirement to process every new security predicate is often negligible because the execution graph generation is the dominant factor in TSV's overall performance overhead.

The reported numbers show that the time required for checking with bounds 10 and 90 seconds, on the desktop and Raspberry Pi, are within reasonable bounds for all use cases. Despite the variance between use cases, it is clear that the costliest test case for symbolic execution was the AssemblyWay, which explored the most feasible paths. The single costliest operation was the construction of the TEG for the AssemblyWay, which took more than three minutes for checking with bound 10.

Figure 8 shows the state space cardinality for the generated TEGs, with the results for the symbolic model verification that takes no more than 6. Requirements for each step vary due to different factors. The costliest test case for symbolic execution was the AssemblyWay, which explored the most feasible paths. The single costliest operation was the construction of the TEG for the AssemblyWay, which took more than three minutes for checking with bound 10.

Figure 9 shows the results of our experiments with TSV that can handle requirement predicates. We ran the same experiments for all of our case studies. The graph shows how much each analysis step contributes to the overall runtime, with bars indicating the percentage of time spent on each step. The figure highlights the large symbolic scan cycle, which is a significant factor in the total analysis time. The results indicate that the time required for checking with bounds 10 and 90 seconds, on the desktop and Raspberry Pi, are within reasonable bounds for all use cases.
Performance

Stacker

Large State Space

Time (s)

0.001
0.01
0.1
1
10
100

Lifting SymEx TEG1 TEG2,3 Trans NuSMV2

PLC'Program'Analysis'and'Formal'Verifica<on'Steps'

Fig. 7. Time Requirements for All Case Studies on Raspberry Pi.

Fig. 8. State Space Size for All Case Studies on Raspberry Pi.
The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The safety requirement included two atomic propositions $\mathbf{a}$ and $\mathbf{b}$, thus each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.

The reported numbers, only 4 out of 11, were not satisfiable. Thus, each state is assigned with a pair of concrete propositions starting in state $S_0$, and for each state, their values in each state. The atomic propositions are fine-grained and atomic proposition-level abstract temporal execution condition for children, propositions.
Performance

Figure 8 shows the state space cardinality for the generated net overhead is within reasonable bounds for all case studies. Despite the variance between use cases, it is clear that the of very large path predicates in the symbolic scan cycle.

Figure 7. Time Requirements for All Case Studies on Raspberry Pi.

Train interlocking. This was caused by checking the feasibility of $\text{AssemblyWay}$, which explored the most feasible paths. The single costliest operation was construction of the TEG for the $\text{AssemblyWay}$ case study with a model checking bound $\text{TSV}$ runs the symbolic model checking engine on the re-

The reported numbers, only 4 out of 11, represent the most applicable in terms of ability to run directly on PLC code without requiring engineers to author an additional high-level system model. As shown in Table I, our approach can check more features than any previous approach.

We now review several previous approaches to safety verification for PLC software. The set of approaches reviewed here represent the most applicable in terms of ability to run on PLC analysis. Existing tools for binary analysis of general purpose programs are omitted as they do not handle PLC to PLC analysis. Our tool, TSV, is written in C++ using the NuSMV 2.3 model checker. NuSMV 2.3 is written in C/C++ and its aim is to provide a component-based framework for designing algorithms and tools for symbolic model checking.

Figure 8 shows the state space cardinality for the generated net overhead is within reasonable bounds for all case studies. Despite the variance between use cases, it is clear that the of very large path predicates in the symbolic scan cycle.

The reported numbers, only 4 out of 11, represent the most applicable in terms of ability to run directly on PLC code without requiring engineers to author an additional high-level system model. As shown in Table I, our approach can check more features than any previous approach.

We now review several previous approaches to safety verification for PLC software. The set of approaches reviewed here represent the most applicable in terms of ability to run on PLC analysis. Existing tools for binary analysis of general purpose programs are omitted as they do not handle PLC to PLC analysis. Our tool, TSV, is written in C++ using the NuSMV 2.3 model checker. NuSMV 2.3 is written in C/C++ and its aim is to provide a component-based framework for designing algorithms and tools for symbolic model checking.

Fig. 7. Time Requirements for All Case Studies on Raspberry Pi.

Time (s) %

<table>
<thead>
<tr>
<th></th>
<th>Lifting</th>
<th>SymEx</th>
<th>TEG1</th>
<th>TEG2,3</th>
<th>Trans</th>
<th>NuSMV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>

To make sure that TSV can be used for real-world PLC verification, it is crucial that it can handle safety requirements.

As shown in Figure 7, the time requirement to process every new security predicate is the largest amount of time to finish the overall analysis. The reason for an analysis that is only executed once when new code is uploaded, this bound does not affect productivity, as safety checks are done independently of plant execution under the previous, legitimate code.
Performance

![Bar chart showing time requirements for various case studies and tools.]

- **Time (s)**
  - Lifting
  - SymEx
  - TEG1
  - TEG2,3
  - Trans
  - NuSMV2

- **Cases:**
  - PID
  - Train
  - Sorter
  - Stacker
  - Traffic Light

- **Tools:**
  - TSV (Symbolic Execution)
  - NuSMV2
  - PID

- **Note:**
  - The chart illustrates the time requirements for different case studies and tools, with PID being the most time-consuming case.
  - The diagram shows the comparison of various execution times across different case studies and tools, highlighting the performance of TSV compared to others.
Related Work

<table>
<thead>
<tr>
<th>Approach</th>
<th>Boolean Logic</th>
<th>Enum</th>
<th>Numeric</th>
<th>Cond. Branching</th>
<th>Function Blocks</th>
<th>MCR</th>
<th>Nested Logic</th>
<th>Timers</th>
<th>Counters</th>
<th>Pointers</th>
<th>Data Blocks</th>
<th>Edge Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Park et al. [22]</td>
<td>SAT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Groote et al. [14]</td>
<td>SAT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biha [21]</td>
<td>Thm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SABOT [18]</td>
<td>Mod</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Canet et al. [6]</td>
<td>Mod</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSV</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Related Work

No efficient reduction

<table>
<thead>
<tr>
<th>Approach</th>
<th>Boolean Logic</th>
<th>Enum</th>
<th>Numeric</th>
<th>Cond. Branching</th>
<th>Function Blocks</th>
<th>MCR</th>
<th>Nested Logic</th>
<th>Timers</th>
<th>Counters</th>
<th>Pointers</th>
<th>Data Blocks</th>
<th>Edge Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Park et al. [22]</td>
<td>SAT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Groote et al. [14]</td>
<td>SAT</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biha [21]</td>
<td>Thm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SABOT [18]</td>
<td>Mod</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Canet et al. [6]</td>
<td>Mod</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSV</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Related Work

**Hard to formalize IL instructions with side-effects**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Boolean Logic</th>
<th>Enum</th>
<th>Numeric</th>
<th>Cond. Branching</th>
<th>Function Blocks</th>
<th>MCR</th>
<th>Nested Logic</th>
<th>Timers</th>
<th>Counters</th>
<th>Pointers</th>
<th>Data Blocks</th>
<th>Edge Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Park et al. [22]</td>
<td>SAT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Groote et al. [14]</td>
<td>SAT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biha [21]</td>
<td>Thm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SABOT [18]</td>
<td>Mod</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Canet et al. [6]</td>
<td>Mod</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSV</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

**TABLE I. COMPARISON OF ANALYZED FEATURES WITH RELATED WORK**
No symbolic state lumping

<table>
<thead>
<tr>
<th>Approach</th>
<th>Boolean Logic</th>
<th>Enum</th>
<th>Numeric</th>
<th>Cond. Branching</th>
<th>Function Blocks</th>
<th>MCR</th>
<th>Nested Logic</th>
<th>Timers</th>
<th>Counters</th>
<th>Pointers</th>
<th>Data Blocks</th>
<th>Edge Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Park et al. [22]</td>
<td>SAT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Groote et al. [14]</td>
<td>SAT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biha [21]</td>
<td>Thm</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SABOT [18]</td>
<td>Mod</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Canet et al. [6]</td>
<td>Mod</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSV</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thanks!

Steve: smclaugh@cse.psu.edu
Saman: s.zonouz@miami.edu