A Trusted Safety Verifier for Process Controller Code

Stephen McLaughlin, Saman Zonouz*, Devin Pohly, Patrick McDaniel
Pennsylvania State University, *University of Miami
{smclaugh, djpohly, mcdaniel}@cse.psu.edu, *s.zonouz@miami.edu

***Abstract*—Attackers can leverage security vulnerabilities in control systems to make physical processes behave unsafely. Currently, the safe behavior of a control system relies on a Trusted Computing Base (TCB) of commodity machines, firewalls, networks, and embedded systems. These large TCBs, often containing known vulnerabilities, expose many attack vectors which can impact process safety. In this paper, we present the Trusted Safety Verifier (TSV), a minimal TCB for the verification of safety-critical code executed on programmable controllers. No controller code is allowed to be executed before it passes physical safety checks by TSV. If a safety violation is found, TSV provides a demonstrative test case to system operators. TSV works by first translating assembly-level controller code into an intermediate language, ILIL. ILIL allows us to check code containing more instructions and features than previous controller code safety verification techniques. TSV efficiently mixes symbolic execution and model checking by transforming an ILIL program into a novel *temporal execution graph* that lumps together safety-equivalent controller states. We implemented TSV on a Raspberry Pi computer as a bump-in-the-wire that intercepts all controller-bound code. Our evaluation shows that it can test a variety of programs for common safety properties in an average of less than three minutes, and under six minutes in the worst case—a small one-time addition to the process engineering life cycle.**

I. Introduction

Control systems used in the power grid, industrial automation, and transportation are fundamental parts of modern society. Due to their strategic importance and large attack surfaces, they are becoming attractive targets for attacks leading to physical damage [16], [29]. Most recently, it was revealed that the Stuxnet malware uploaded malicious code to Programmable Logic Controllers (PLCs) to physically damage the centrifuges they controlled [11]. A recent study found that PLC honeypots experienced not only port scanning, but also attempts at modifying control system specific protocols and access to system diagnostics [34]. These vulnerable Internet-connected controllers are exposed by computer search engines such as Shodan [1]. This has led to efforts for securing critical infrastructure based on control systems [17], [31]. Recent research has demonstrated that attacks against PLCs can be partially automated [18]. This makes the protection of PLCs from malicious code injection of paramount importance.

Past efforts to ensure control system safety have focused on securing the control system's Trusted Computing Base (TCB), the set of hardware and software that must function properly to ensure safe behavior. Unfortunately, such approaches are insufficient as software patches are often applied only months after release [26], and new vulnerabilities are discovered on a regular basis [4], [23]. Instead, we focus on significantly reducing the TCB size by verifying the safety of control code immediately before it is executed on the PLC. For our purposes, a *safety property* is any temporal property that can be stated in linear temporal logic. This allows TSV to protect against malicious behaviors beyond those handled by existing failsafe mechanisms, such as captive key interlocks. Existing safety verification methods for PLCs suffer from two main limitations. (*i.*) Those based on model checking experience the state explosion problem when dealing with numerical inputs. (*ii.*) Previous techniques fail to check code using common PLC features such as master control relays and non-volatile data storage (See Section VII).

In this paper, we present the Trusted Safety Verifier (TSV). TSV reduces the Trusted Computing Base (TCB) for safe process execution from the entire control system down to the protected PLC and a single embedded computer administered through a narrow interface. TSV is deployed as a bump-in-the-wire that sits between the control system network and the PLC. All code to be executed on the PLC is first checked against a set of *safety properties* defined by process engineers. Examples of safety properties include bounds on numerical device parameters, e.g., maximum drive velocity and acceleration, and *safety interlocks*, which ensure physically conflicting events do not occur.

This paper makes the following contributions.

- We introduce TSV as by far the most minimal TCB proposed to maintain control system safety properties.
- Our tool can check a much larger variety of PLC code than any previous safety analysis tool. (See Table I in Section VII.)
- TSV is the first binary analysis tool to verify temporal properties, including for programs that take numerical inputs.
- We fully implement TSV and demonstrate its performance in checking a variety of properties against different control systems.

TSV works in a number of steps. First, an assembly-level PLC program, called an *instruction list* (IL), is translated

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '14, 23-26 February 2014, San Diego, CA, USA
Copyright 2014 Internet Society, ISBN 1-891562-35-5
http://dx.doi.org/10.14722/ndss.2014.23043

into the Instruction List Intermediate Language (ILIL). ILIL, which we based on the Vine intermediate language [30], makes all instruction side-effects explicit, thus maintaining all information flows between system registers and memory. In the second step, the ILIL program is symbolically executed (mixed with concrete executions when values are known) to produce a *symbolic scan cycle*, representing all executions of the code by the PLC. Thus, all inputs which produce an equivalent symbolic output are lumped together. Third, the symbolic scan cycle is used to construct a novel *Temporal Execution Graph*, (TEG) which represents multiple repeated executions of the controller code. Finally, the TEG is checked against a set of safety properties specified in linear temporal logic. If any check fails, a counterexample and the offending instructions are provided to the system operator.

TSV Safety properties are the same as those used by process engineers in designing the control system. To maintain the integrity of TSV's platform, new safety rules should be updated by a physically present operator with a special programming device. Given that this only needs to occur when the physical safety requirements of plant machinery change, this measure is far less burdensome than some proposed by large industrial security standards [20].

We implemented a working prototype of TSV on a Raspberry PI embedded computer to check code for Siemens PLCs, the most widely deployed in the world [28]. We evaluated TSV on six case studies representing a diverse set of PLC and control system functionality. These case studies consist of specifications and code that is runnable on several of the most popular PLC architectures. Our implementation can check for typical safety properties like range violations and safety interlocks in a few minutes. If a safety check fails, a useful counterexample and the offending instructions are produced. By using an intermediate language for our analysis, TSV can be extended in the future to handle check code for other proprietary PLC architectures. Because our prototype checks assembly-level code, it effectively checks any PLC program written in higher-level or graphical languages like relay ladder logic, function block diagram, and structured text.

This paper is organized as follows. Section II gives an introduction to PLCs and lays out our assumed threat model. Section III gives an overview of TSV's architecture and use. Section IV and Section V explain the symbolic execution and model checking engines in detail. Section VI describes our prototype implementation and evaluation results. Section VII covers the related work in control system safety and security, and Section VIII concludes.

II. Background and Threat Model

A. Programmable Logic Controllers

A Programmable Logic Controller (PLC) is a digital, multi-input multi-output computer used for real-time automation of physical machinery. They are used in virtually every control application from assembly lines to nuclear power plants. The PLC sits in a tight closed loop with the physical system it controls. Many times per second, the PLC reads sensor measurements, calculates the necessary change to the system, and sends commands to physical machinery to make the changes. The PLC uses a modifiable software program to perform the second step.

A PLC's program is executed continuously as long as the PLC is running. Each execution of the program is called a *scan cycle*, and typically lasts several milliseconds. On each scan cycle, three steps occur. (*i.*) The sensor inputs are buffered into the *input memory* (`I`). (*ii.*) The PLC program is executed to perform calculations based on the input memory and state from the previous scan cycle. (*iii.*) The result of the PLC program is buffered in the *output memory* (`Q`), from where it is transmitted to the plant machinery. In addition to the sensor and machine interfaces, PLCs have a separate *programming interface*, e.g., Ethernet or RS-232 for uploading of new code and data by process engineers and plant operators. TSV's job is to efficiently check any code coming over this interface for safety properties specific to the plant machinery.

In addition to the typical features found in most instruction set architectures PLCs employ a number of special features that TSV must handle.

- **Function Blocks.** PLCs execute code in discrete segments called *function blocks* with fixed entry and exit points. Each function block has a local memory that only it can address. This is different from stack memory in that each function block's local memory exists in the same absolute address space, i.e., not relative to a stack pointer.
- **Timers.** PLCs support hardware timers that evaluate to a Boolean value. A timer starts when its input experiences an edge transition. Once the timer has reached a preset time, its own output goes from low to high.
- **Counters.** A counter is a value in PLC memory that is incremented each time a specified instruction causes a value to go from low to high. This is useful for counting events like the number of times an input wire receives a high signal.
- **Master Control Relays.** A Master Control Relay (MCR) defines a section of PLC code which behaves differently depending on the value of a specific input wire. If the MCR input is false, the code executes normally. If it is true, certain instructions will output a zero value. This is done to halt any machinery in case of an emergency condition.
- **Data Blocks.** PLCs retrieve configuration information about the physical process from blocks of persistent storage called *data blocks* (DBs). Each DB has a unique integer used to qualify any addresses to its data. A special kind of DB, called an *instance* DB is also used to pass parameters to function blocks.
- **Edge Detection.** Certain PLC instructions will only execute after a specific memory value goes from low to high. This requires the CPU to check for low to high transitions before any such instruction executes.

B. Threat Model

In modern control system networks, a security flaw in almost any component can be leveraged to upload malicious code to a PLC. A clear example of this was the Stuxnet virus, which used many potential vectors, including the program

development environment, to propagate to a PLC-connected computer [11]. TSV's aim is to reduce the amount of control system infrastructure needed to guarantee safe behavior of PLC-controlled processes to a single embedded computer and the PLC itself.

It must be possible to notify plant operators when a safety violation is found in some PLC-bound code. If there is a rootkit present on a PLC connected computer, then notifications from TSV may be suppressed at the receiver. To handle this, we assume that there is some narrow, secure interface for notifications. A common way of implementing this in a control environment could be by way of an analog alarm, similar to those sounded when a piece of physical machinery malfunctions. Upon hearing the alarm, plant operators could directly download the safety counterexample, e.g., via serial port. In an emergency situation where immediate PLC access is needed, a physical switch in the plant could be employed to bypass TSV, similar to the emergency stop buttons on most heavy machinery.

We also must assume that the interface for uploading safety properties to TSV is secured. For example, a simple file format could be read from a USB key directly by TSV. While the use of an air-gap may seem to mitigate the advantage of a network connected PLC, safety properties require modification far less frequently than PLC code. Compared to the large numbers of requirements in existing industrial security regulations [20], this is a small additional overhead. We note that TSV is not secure against a privileged insider with physical access to the plant floor.

TSV cannot defend against false data injection attacks, in which a PLC is given forged sensor data. Additional defenses already exist for such attacks based on improved state estimators [5], [27]. Additionally, TSV cannot defend against PLC firmware exploitation, in which case the verified control logic can be completely bypassed by the compromised PLC.

III. System Overview

Figure 1 shows TSV's architecture. TSV sits as a bump-in-the-wire between system operators and the PLC. Any piece of PLC-bound code is intercepted and checked for safety properties, previously supplied by process engineers. To test safety properties written in temporal logic, TSV uses model checking for part of the verification. Model checking suffers from state space explosion on numerical inputs. For example, if a PLC program has a single conditional branch depending on an integer value, the model checker will explore all of the possibilities, i.e., 2^{32} states. To prevent state space explosion, TSV first performs a symbolic execution of the program to *lump* together all inputs that produce the same symbolic output. The resulting state machine is many orders of magnitude smaller than the naive approach.

Symbolic execution of the PLC code occurs in two main steps. First, the PLC code is lifted into an intermediate language designed to make the analysis more generic and explicit. The lifted program is then symbolically executed to generate a mapping from path predicates to symbolic outputs. This mapping contains all possible executions of a single scan

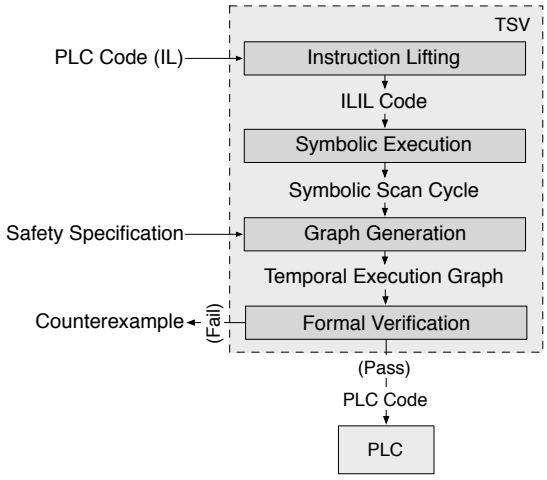


Fig. 1. TSV Architecture.

cycle[1], hence, we refer to it as a *symbolic scan cycle*. Once the symbolic scan cycle is calculated, TSV's model checking component is invoked. The symbolic scan cycle is an important step, as it prevents intra-scan cycle property violations from being rejected by TSV. In other words, if the PLC's variables temporarily violate a safety property at some point during the scan cycle, but the property is not violated when control signals are sent to physical machinery, then TSV will not reject the control program. This is an important distinction from the existing work combining symbolic execution and formal verification.

The formal verification component takes a set of temporal properties, consisting of atomic propositions combined with temporal qualifiers, and verifies they are maintained by a state-based model of the PLC code. We call this model, the *Temporal Execution Graph* (TEG). The TEG is a state machine in which each state transition represents a single scan cycle. Each state in the TEG contains the state and output variables of the PLC program, and a set of Boolean variables for the atomic propositions. Each Boolean represents whether the safety property is true or false in that state. If any path is reachable in which a property is false, then a safety violation is raised.

The TEG is constructed as follows. An initial state is chosen in which all state and output variables are zero. Recalling that the symbolic scan cycle maps path predicates to symbolic values for PLC variables including the output variables, a new state is generated for each path predicate with its PLC variables updated according to the corresponding symbolic output. For each of these newly created states, several more states are generated with different valuates of the atomic propositions. The process then repeats on each new state. See Section V-B for a detailed description of TEG creation.

IV. PLC Code Analysis

TSV verifies safety properties of PLC code using a combination of symbolic execution and model checking. Traditional model checking, which explores states exhaustively,

[1]In practice, it contains all executions that are reachable by scan cycle's hard deadline. See Section IV-B.

$$
\begin{aligned}
\mathit{prog} &::= \mathit{inst}^*\mathit{fun}^* \\
\mathit{fun} &::= \texttt{ident}(\mathit{var})\{\mathit{inst}^*\} \\
\mathit{inst} &::= \texttt{cjmp}\ e,e,e \mid \texttt{jmp}\ e \mid \texttt{label ident} \mid \texttt{ident} := e \\
&\quad \mid \texttt{call ident}(\mathit{var}{=}e) \mid \texttt{ret} \mid \texttt{assert}\ e \\
e &::= \texttt{load}(\texttt{ident},\mathit{addr}) \mid \texttt{store}(\texttt{ident},\mathit{addr},\texttt{int}) \mid e\ \mathit{binop}\ e \\
&\quad \mid \mathit{unop}\ e \mid \mathit{var} \mid \mathit{val} \mid (e) \\
\mathit{binop} &::= +,-,*,/,\texttt{mod},\&,\&\&,<<,\ldots \quad \text{(And signed versions.)} \\
\mathit{unop} &::= - \text{ (Negate)}, \sim \text{ (Bitwise)}, ! \text{ (Boolean)} \\
\mathit{var} &::= \texttt{ident}\ (:\ \tau) \\
\mathit{val} &::= \mathit{mem} \mid \mathit{addr} \mid \texttt{int}\ (:\ \tau) \\
\mathit{mem} &::= \{\mathit{addr} \mapsto \texttt{int},\ \mathit{addr} \mapsto \texttt{int}, \ldots\} \\
\mathit{addr} &::= [\texttt{int} :: \texttt{int} :: \ldots] \\
\tau &::= \texttt{reg1\_t}\ldots\texttt{reg64\_t} \mid \texttt{mem\_t}(\texttt{int}) \mid \texttt{addr\_t}
\end{aligned}
$$

Fig. 2. Simplified ILIL Grammar.

```
 0. // Initialize PLC state.
 1. mem := {} : mem_t(1);          // Main memory.
 2. I   := 0;                      // Input memory qualifier.
 3. Q   := 1;                      // Output memory qualifier.
 4. RLO := 1 : reg1_t;             // Boolean accumulator.
 5. FC  := 0 : reg1_t;             // System status registers.
 6. STA := 0 : reg1_t;
 7. ...
 8.
 9. // A I 0.5
10. STA := load(mem, [I::0::0::0::5]);
11. cjmp FC == 0 : reg1_t,L1,L2;
12. label L1;
13. RLO := STA;
14. label L2;
15. RLO := RLO && STA;
16. FC  := 1 : reg1_t;             // Side effects.
17. ...
18.
19. // = Q 0.1
20. STA := RLO;
21. mem := store(mem, [Q::0::0::0::1], RLO);
22. FC  := 0 : reg1_t;             // Side effects.
23. ...
```

Fig. 3. ILIL Code example (IL in Comments).

suffers from state space explosion when checking programs with numerical inputs. To reduce the state space, a symbolic execution of the program is performed first to lump together all inputs that lead to the same symbolic output. The result is a mapping from path predicates to symbolic outputs. In addition to the features described in Section II, our analysis addresses a number of other novel issues.

- **Lack of High-level Languages.** PLCs have traditionally been programmed either in assembly language or in graphical circuit languages like relay ladder logic. The graphical languages are mere sugar used to make the assembly appear like a circuit, and they add no additional semantic information. We are thus forced to do binary or assembly-level analysis.
- **Hierarchical Addresses.** PLC addresses are not just integer values. They are prefixed by an architecturally fixed number of namespace qualifiers. When analyzing indirect addressing, we must not only consider byte address pointers, but also indirect references to different namespace qualifiers.
- **Multi-indexed Memory.** While most hardware memories only support a single size of memory access, e.g., byte or word addressing, some PLC memories can be addressed at the word-, byte-, and bit-level. This should not be confused with loading different sized registers from a byte address. Multi-indexed memory complicates the dynamic taint analysis needs for mixed execution.

Symbolic execution of IL programs happens in two stages. First, the IL program is lifted into the ILIL intermediate languages for PLC code. Second, a mixed symbolic and concrete execution of the ILIL program is done to generate path predicates that feed into the model checking steps.

A. Instruction List Intermediate Language

Directly analyzing an IL program would be prohibitively difficult. IL syntax and semantics vary widely by vendor, and IL instructions have side effects that can obscure certain control flows. For these reasons, we introduce the IL Intermediate Language (ILIL) as a basis for our analysis. ILIL is based on the Vine Intermediate Language [30] (Hereafter, Vine) used for binary code analysis. We extend Vine to handle several PLC-specific features described below. A simplified grammar for ILIL is shown in Figure 2. Vine features such as casts and memory endianness are omitted for space sake. The full ILIL semantics are in Appendix B.

An ILIL program is a set of top-level instructions followed by function definitions. This may seem strange for a binary analysis, but there are two reasons for the distinction. First, PLC code begins execution in an *Organization Block* (OB), akin to the entries in an operating system's interrupt vector. OBs are implemented in top-level code. Second, on some architectures, OBs make additional calls to function blocks. For each function call, additional ILIL code is generated to handle the parameter passing.

ILIL uses the two basic Vine types *registers* and *memories*. A single register variable is used to represent each CPU register in a particular PLC architecture. They are implemented as bit vectors of size 1, 8, 16, 32, and 64 bits. Memories are implemented differently than in Vine. ILIL Memories are mappings from hierarchical addresses (See next paragraph.) to integers. Memory loads return the integer for a given address. Memory stores return a new copy of the memory with the specified location modified.

In addition to registers and memories, ILIL adds a third type, *addresses*. In Vine, memories are mappings from integers to integers. This is reasonable as most architectures use 32- or 64-bit address registers. This is not sufficient for PLCs which use *hierarchical addresses*. A hierarchical address has several namespace qualifiers before the actual byte or bit address. For example, in Siemens PLCs, addresses have a single namespace qualifier called a memory area. In Allen Bradley, there are three: rack, group, and slot. ILIL addresses are essentially integer lists where the leftmost n entries represent the n namespace qualifiers. We also extend the memory type to include n. Thus, the ILIL statement:

```
mem := {} : mem_t(1);
```

initializes an empty memory with a single namespace qualifier. In some cases, all or part of an address will be stored in memory. To handle loads and stores of hierarchical addresses, we extend the Vine `cast` expression to convert addresses to byte sequences. Note that the number of namespace qualifiers prefixing an address is architecturally fixed, so the number of types is finite.

ILIL instructions have no side effects, making all control flows explicit. As an example, Figure 3 shows the lifted version of the IL instructions:

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

First, the machine state is configured to have a single main memory and two memory areas for input and output. Part of the definition of the system status word is also shown. The *And* instruction consists of three parts. The operand is loaded from memory, combined with an accumulator, and one or more status words are updated. The address `[I::0::0::0::5]` is read, "memory area `I`, dword 0, word 0, byte 0, bit 5." This convention of listing offsets of different sizes allows us to canonically address multi-indexed memories.

The PLC features from Section II-A, as well as several other issues, are handled by IL to ILIL translation as follows.

Timers. For each timer, an unused memory address is allocated. During symbolic execution, an attempt to check the timer value at this address will generate a fresh symbol. In the model checking step, this symbol will be nondeterministic, i.e., it will cause both paths to be explored if used in a branch condition. A similar approach was used by SABOT [18], though our semantics are more flexible in allowing for the case where the timer value changes within a scan cycle, not just between them.

Counters. Counters are implemented in a straightforward manner. For each counter, a memory word is allocated to handle the current value. ILIL instructions are added to check if the counter's condition has transitioned from low to high each time a counter instruction is executed. Once the counter reaches a preset value, attempts to access its address in the counter memory area will return the concrete value true.

Master Control Relays. When an MCR section is reached, a conditional branch is generated depending on the MCR status bit. The false branch contains the original code, and the true branch contains code that modifies instruction results. While the semantics differ by architecture, typically numerical and Boolean instructions all output 0 or false when the MCR is active.

Data Blocks. Data blocks are implemented using the hierarchical address type. When a program opens a data block, a namespace qualifier is created with the index of that data block, e.g., `DB3`. When an access is made into the datablock, the address is prepended with the qualifier, e.g.,, `[DB3::20::1]` for word 41. Each data block is populated with any configuration data blocks accompanying the PLC code.

Edge Detection. For each bit of memory that is checked for a low-to-high edge transition, ILIL code is generated to monitor that bit across scan cycles. If an edge is detected, a separate bit address is set to true. This address is then checked before any dependent instructions are executed.

Flow-sensitive Optimizations. During instruction lifting, additional control flows are added to the program. For example, after an integer addition, an overflow check is added, setting several status registers to either 0 or 1. To prevent additional control flows from leading to path explosion, we only include such checks when a subsequent instruction has an explicit data dependency on the result. For example, if two additions are done in a row followed by a jump that checks an overflow status flag, only the overflow check of the second addition will be included in the lifted code.

Memory Tags. PLCs use strings, sometimes called tags, as human-readable labels on memory locations. A group of tags may be referenced by a single *name*. This leads to a complicated issue with function block parameter passing. If the name of a tag group is passed to a function, the PLC performs a pass-by-value of all tags in the group. As we would like to expose such execution semantics to our analysis, ILIL code is generated to do a pass-by-value of each tagged memory location in the group.

B. ILIL Symbolic Execution

TSV symbolically executes an ILIL program to produce a mapping from path predicates to symbolic outputs. This mapping, called the *symbolic scan cycle*, describes all possible executions of a single PLC scan cycle. Fresh symbols are allocated the first time a previously unwritten memory location is accessed. Thus, if a sensor input `I0.2` is read, then a new symbol `I_0_0_0_2` will be generated and used each subsequent time that same location is read.

Symbolic execution follows all possible paths through a single scan cycle of the program. An SMT solver is used to ensure only feasible paths are followed. Loops are followed for a constant number of iterations. Because PLC scan cycles are terminated at a hard deadline, this number of iterations can be set high enough to ensure TSV explores all iterations that are reachable by the deadline. PLCs allow function calls by indirect reference, e.g., `call FB [MD 0]` where the function block number is stored in `MD 0`. Fortunately, if `MD 0` contains a symbolic value, there is only a small number of possible functions it could resolve to, making the jump successor problem more tractable. Symbolic execution must handle two additional challenges, register type inference and mixed execution.

Register Type Inference. Typically, binary analysis is done on bit vectors using the register sizes of the target architecture. This is sufficient for PLC analysis, except in the common case of real-valued computations. While bit vectors will not work here, we would still like to make some safety assertions about real-valued PLC outputs. TSV relies on opcodes to infer which symbols are real-valued. Initially, all symbols start as uninitialized. The first time an instruction is executed on that symbol, or a variable that symbol propagated to, it is assigned either real or bit vector, depending on the opcode. This has the

minor limitation that if both a real-valued and non-real-valued instruction are executed on the same symbol, the symbolic machine gets stuck. This is however, not common. A symbol's type can be changed only by a cast instruction in the original IL code.

Mixed Execution. PLC programs make heavy use of constants as process parameters. Thus, many instructions can be executed on concrete operands instead of symbolically. Like previous tools, such as Rudder [30], TSV performs a mixed symbolic and concrete execution. An expression produces a concrete result iff all its variables are concrete. This requires dynamically tracking whether each register and memory word is concrete or symbolic. There is a complication here for multi-indexed memories, which can be accessed at the word-, byte-, or bit-level. To handle this, TSV tracks each bit of memory as either symbolic or concrete. Initially, all memory is concrete, and typical taint propagation rules are used to track symbolic bits. We add an additional rule to allows a bit to become concrete again. If a sequential string of symbolic bits are overwritten by a equal or longer string of concrete bits, then the whole string becomes concrete.

V. Model Checking

Because PLCs use *stateful* variables, that retain their values across the scan cycles, analysis of a single scan cycle is not sufficient to check all temporal safety properties. In this section, we describe our technique of model checking a *temporal execution graph* for paths on which safety properties are violated. The results of symbolic execution are used to first construct the TEG to model the state transitions occurring over a scan cycle. Each node of the TEG is then productized with valuations of the atomic proposition in the linear temporal logic (LTL) safety property. Finally, the symbolic variables are removed from each state to produce an abstract graph, which is fed to the model checker. Before exploring this process in more detail, we briefly review LTL as used for safety specifications.

A. Linear Temporal Logic

To formulate control system security requirements, TSV makes use of the linear temporal logic formalism [2], [25]. Let us define A to be a finite set of atomic logical propositions about the system $\{b_1, b_2, \cdots, b_{|A|}\}$, e.g., `relay` R_1 `is open.` and $\Sigma = 2^A$ a finite alphabet composed of the abovementioned propositions. Every element of the alphabet is a possibly empty set of propositions from A, and is denoted by a_i, e.g., $a_i = b_1, b_4, b_9$.

The set of linear temporal logic-based safety requirements is inductively defined by the grammar

$$\varphi ::= \mathit{true} \mid b \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \,\mathbf{U}\, \varphi \mid \mathbf{X}\, \varphi, \tag{1}$$

where $\neg$ and $\vee$ denote negation and logical *OR* operators. $\varphi_i \,\mathbf{U}\, \varphi_j$ denotes "the LTL expression φ_i remains true *until* φ_j becomes true," and $\mathbf{X}\, \varphi_j$ reads "φ_j must be true in the *next* step (execution state)". TSV also makes use of the following redundant notations: $\varphi \wedge \psi$ instead of $\neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi$ instead of $\neg\varphi \vee \psi$, $\mathbf{F}\, \varphi$ (Eventually) instead of $\mathit{true} \,\mathbf{U}\, \varphi$, and $\mathbf{G}\, \varphi$ (Globally) instead of $\neg(\mathit{true} \,\mathbf{U}\, \neg\varphi)$. For example, consider a traffic light system with Boolean variables g_1 and g_2 that activate green lights for intersecting streets when true. The property that both lights are never green at the same time has two atomic propositions: $a \equiv g_1 = \mathit{true}$, and $b \equiv g_2 = \mathit{true}$. The global LTL property is then stated $G\ \neg a\ \vee\ \neg b$.

B. Temporal Execution Graph

Each state in the execution graph stores the symbolic value of each stateful PLC variable. It is noteworthy that TSV performs intermediate variable elimination during the temporal execution graph generation to make sure that values of the symbolic variables are terms over only constants and time-indexed input variables corresponding to PLC input wires scanned during some PLC input-output scan in the past, i.e., there is no intermediate variable involved in the symbolic values of the variables. To clarify, consider a symbolic execution output entity with the assignment statements $O := X + 2$ and $X := I + X$ where O and I are output and input variables, respectively. All variables are initialized to 0 before the first scan cycle. After the first scan cycle the variables will have values: $O \leftarrow I^0 + 2$ and $X \leftarrow I^0$, where I^0 denotes the input variable scanned before the first PLC execution iteration. Similarly, the second scan will result in $O \leftarrow I^0 + I^1 + 4$ and $X \leftarrow I^0 + I^1$. The final expression for O no longer contains X.

The state transitions of the temporal execution graph indicate the feasible paths between scan cycles. Each transition is labeled by the path predicate from one entry in the symbolic scan cycle. A transition is added from a source to destination state iff the path predicate can be satisfied given the symbolic values of PLC variables in the source state. For instance, if the source state has PLC variables[2] $O \leftarrow I^0 + I^1 + 4$, $X_1 \leftarrow I^0 + I^2$, and $X_2 \leftarrow I^0 + I^1$, given the path predicate $X_1 \geq X_2$, then a transition is added because the path predicate is satisfiable under the symbolic values at the source state and the input values, i.e., $I^2 \geq I^1$.

C. Specification-Based Model Refinement

To check temporal properties, a model checker needs to know the truth value of each atomic proposition of the given LTL requirement in each state. The addition of these truth values to the TEG is called model refinement that is described separately here for presentation clarity; however, the model refinement occurs concurrently during the TEG generation (Section V-D). The motivation for this step is that it is impossible to pick a single truth value for an atomic proposition containing symbolically valued PLC variables. In such cases, each state in the TEG is replicated to a set of states for all feasible truth values of the atomic propositions in those states. TSV accordingly updates each replica's path predicate, which captures the input variable conditions for the execution to get to that state, based on the assigned concrete Boolean atomic proposition values. In particular, TSV labels each replica with a conjunctive predicate composed of the state's original predicate P and the predicate representing the concrete atomic proposition values assigned to the replica. For instance, in the case of a single atomic proposition a, the two state replicas will be assigned $P\&a$ and $P\&!a$ as their predicates. Figure 4 shows a more illustrative example

[2]Note that sub-indices and super-indices represent different variables and scan cycle numbers, respectively.

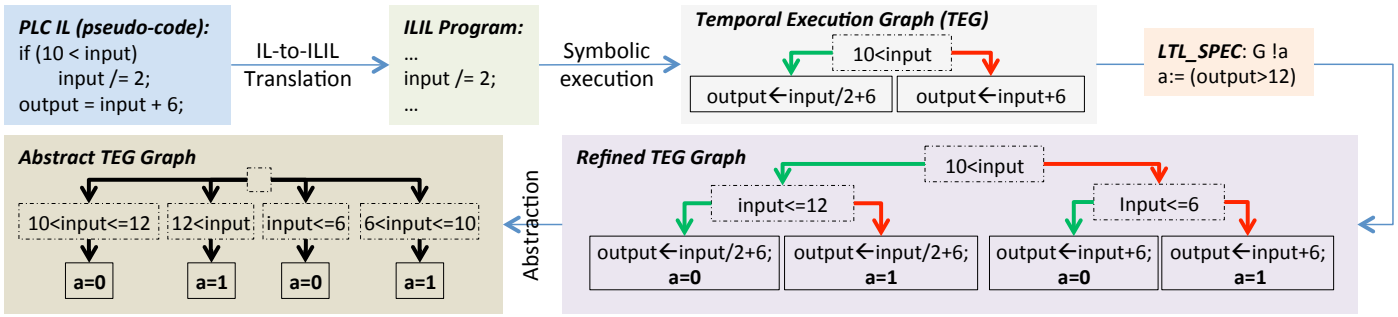


Fig. 4. Example TEG generation for checking of an LTL property.

where the original IL code assigns the output variable value depending on the input variable value. After the IL-to-ILIL conversion, TSV employs the generated TEG graph (for a single scan cycle) along with the given safety requirement to determine concrete atomic proposition values. In particular, each node in the TEG becomes two nodes in the Refined Graph to model both valuations of proposition a. Consequently, TSV further abstracts the refined TEG graph to only include the information that is sufficient for the formal verification phase (Section V-E).

D. TEG Generation Algorithm

This section describes in details the procedure for TEG generation (Algorithm 1)[3]. The main inputs to the algorithm are i) symbolic scan cycle set `ssc`, i.e., symbolic execution outputs that are mappings from path predicates to symbolic PLC variable values; ii) the safety specification of the underlying physical system φ, and iii) the termination deadline γ for the TEG generation algorithm. TSV parses the given LTL safety formula to get the corresponding atomic propositions[4] (Line 1). The TEG generation algorithm starts with initializing the TEG state space Ω by creating an initial state σ where all of the PLC variables/predicates are reset to zero/true (Lines 2-7) that happens when the PLC loads the controller code for the first time.

Regarding the TEG state notion, each state includes three types of information: i) $\sigma_{\text{predicate}}$ denotes the logical predicate as the result of symbolic execution of branch/jump instructions that has been accumulated in the current state through the state transition sequence starting at the initial state σ; ii) $\sigma_{\text{var\_values}}$ indicates the symbolic variable values that have been initiated in the current state; and iii) $\sigma_{\text{prop\_values}}$ represents the concrete Boolean value vector for the atomic propositions in the current state. For the initial state[5], given the reset concrete variable/predicate values, the concrete values for the LTL atomic propositions $\mathtt{A}$ are calculated and stored in $\sigma_{\text{prop\_values}}$ (Line 6); however, for other states storing symbolic values, TSV takes a different approach to assign concrete atomic proposition values as discussed below. The TEG state space

Algorithm 1: GenTEG

Input : The Symbolic scan cycle `ssc`
Input : The LTL safety specification φ
Input : The TEG generation deadline γ
Output: The generated temporal execution graph TEG

1 $\mathtt{A} \leftarrow$ get_atomic_propositions(φ)
2 $\sigma \leftarrow$ create_initial_state()
3 **if** initial_GenTEG_call **then**
4 $\quad \sigma_{\text{predicate}} \leftarrow$ inilialize_predicate(True)
5 $\quad \sigma_{\text{var\_values}} \leftarrow$ inilialize_PLC_variables(False)
6 $\quad \sigma_{\text{prop\_values}} \leftarrow$ concretize_atomic_propositions($\sigma_{\text{vars\_values}}, \mathtt{A}$)
7 $\quad \Omega_s \leftarrow \sigma$
8 $\quad \Omega_e \leftarrow \emptyset$
9 **foreach** Path predicate $\pi \in$ `ssc` **do**
10 $\quad$ symbolic_values $\leftarrow$ `ssc` [π]
11 $\quad$ **foreach** $\alpha \in 2^{\mathtt{A}}$ **do**
12 $\quad\quad \tau \leftarrow \sigma_{\text{predicate}} \bigwedge \text{predicate}(\alpha, \mathtt{A}) \bigwedge \pi$
13 $\quad\quad$ **if** $\neg$satisfiable(τ) **then**
14 $\quad\quad\quad$ continue
15 $\quad\quad \sigma' \leftarrow$ create_state()
16 $\quad\quad [\sigma'_{\text{predicate}}, \sigma'_{\text{prop\_values}}] \leftarrow [\tau, \alpha]$
17 $\quad\quad \sigma'_{\text{var\_values}} \leftarrow$ update($\sigma_{\text{var\_values}}$, symbolic_values)
18 $\quad\quad \sigma'' \leftarrow$ find_equivalent_state(Ω, σ')
19 $\quad\quad$ **if** $\sigma'' \neq$ NULL **then**
20 $\quad\quad\quad$ delete(σ')
21 $\quad\quad$ **else**
22 $\quad\quad\quad \sigma'' \leftarrow \sigma'$
23 $\quad\quad \Omega_s \leftarrow \Omega_s \cup \{\sigma''\}$
24 $\quad\quad \Omega_e \leftarrow \Omega_e \cup \{(\sigma \rightarrow \sigma'')\}$
25 $\quad\quad$ **if** $\gamma <$ elapsed_time **then**
26 $\quad\quad\quad$ return
27 $\quad\quad$ GenTEG(`ssc`, φ, σ'')

Ω_s and set of transitions Ω_e are also initialized to the initial state σ and empty set, respectively, during the initial function call (Line 7-8).

Following the algorithm, a transition is then added for each (path predicate, symbolic output values) mapping in the symbolic scan cycle `ssc` (Line 9) that is satisfiable given the variable values in the initial state σ. The algorithm goes through

[3] A concrete example for the symbolic execution and formal verification of a given controller program is described in Appendix A.

[4] Note that "←" denotes an assignment.

[5] It is assumed that the function `GenTEG` takes a Boolean argument initial_GenTEG_call that denotes whether this is the first call in the recursion chain. Due to presentation simplicity, the variable is not listed in the algorithm's input list explicitly.

a nested loop (Line 11) to be able to assign concrete Boolean values for each atomic proposition on every generated TEG state (Section V-C). TSV produces the conjunctive predicate τ using i) the accumulated state predicate; ii) the path predicate π from `ssc`; and iii) the concrete atomic proposition vector α (Line 12). The satisfiability check is performed in Line 13 that, if satisfiable, allows TSV to create the corresponding state σ' (Lines 15-17) and transition, and update TEG (Lines 23-24).

The update function (Line 17) creates the symbolic variable values for the new state σ'. It takes the symbolic variable values in the source state $\sigma_{\text{var\_values}}$, that captures the PLC's current memory state, as well as the symbolic values from the corresponding program control path in `ssc` (Line 10). Consequently, the update function performs the intermediate variable elimination step (Section V-B) to get rid of intermediate variables in the `ssc` symbolic values, and stores the result in the new state's symbolic variable values $\sigma'_{\text{var\_values}}$.

There is a case in which the state will not be added even when the path predicate is satisfied. If the TEG already contains a state with PLC variables equivalent to the destination state (Line 18), then a transition is added back to the existing state, and the new destination is discarded (Line 20). Two states are considered equivalent if their PLC variables have equal symbolic values. This step enables TSV to avoid unnecessary state space size increase, and hence improves the formal verification efficiency. It is noteworthy that to decrease the false negative rates of the state equivalence checking function, TSV checks for equality after simplifying the symbolic values. For instance, TSV will mark the $X_1 \leftarrow I_k^0 + 2 + 3 \cdot I_k^0$ and $X_2 \leftarrow 4 \cdot I_k^0 + 2$ as equal after the simplification of those expressions' abstract syntax trees.

Finally, TSV calls the TEG generation function `GenTEG` recursively to explore next possible states starting the recently explored state σ''. The recursive graph generation procedure returns under two conditions. First, the procedure returns if all of the states are created and the graph is completely generated. This is the ideal return condition as the complete graph will result in accurate model checking results with a counterexample. Second, the procedure returns of the explored depth, i.e., the number PLC input-output scans, exceeds a predefined bound value (Line 25). This results in a partially generated temporal execution graph that is later used for formal model checking. The bounded graph generation is a suitable solution when the size of the program is large and complete graph generation is too costly.

To summarize, TSV strives for minimality of model state space through three approaches. (*i*.) Symbolic execution lumps as many concrete input values (and hence, scan cycles) together as possible. (*ii*.) In the refinement step, a truth value for a proposition is only added if it is feasible transitioning from the previous state. (*iii*.) As a measure of last resort, TSV will perform bounded model generation when the TEG's diameter becomes too large.

E. Malicious Code Discovery

TSV uses the abstract TEG to perform LTL-based model checking [8] that either allows the code to run on the PLC after passing all checks, or returns a counterexample to system operators in the event that a violation is found. More specifically, the model checker verifies whether the refined temporal execution graph contains any paths in which a temporal property fails to hold. Given a temporal predicate f, TSV negates f[6] and generates a *tableau* $T(\neg f)$. The tableau is a state-based automaton that satisfies every sequence of words that satisfy $\neg f$. Here, a word is an truth assignment to all atomic propositions in f. TSV then computes the product automaton P of T and the TEG. If an accepting path is found in P, then the values of atomic propositions along that path form a counterexample for the temporal property f.

The counterexample can be used to locate the offending lines of code or control flows in the original PLC program. In the event of malicious code injection, this could shed light on the attackers motives, and if a safety violation occurred due to an error, operators can take corrective actions. We demonstrate this functionality in Section VI-C.

VI. Evaluation

We now wish to investigate TSV's efficacy in checking typical safety properties against a representative set of PLC programs. In particular, we designed a set of experiments to verify whether TSV can be useful in real-world practical scenarios by answering the following questions empirically: How accurately do the employed model checking techniques in TSV verify whether a given PLC code is compliant with the requirements? How efficiently does TSV complete the formal verification steps for an uploaded PLC code? How well can TSV scale up for complex security requirements? We start by describing the experimentation control system case studies, and then proceed to examine these questions.

A. Implementation

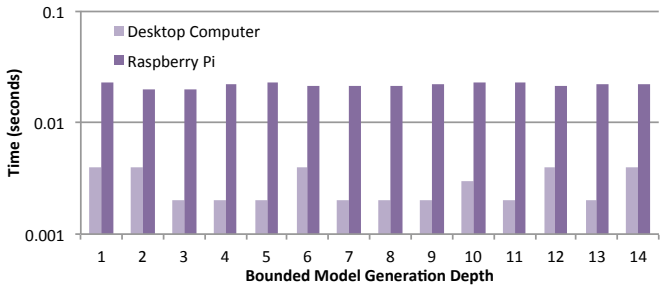
We implemented TSV on a Raspberry Pi embedded computer running Linux kernel 3.2.27. The IL[7] lifting is implemented in 2,933 lines of C++ code, the symbolic execution in 11,724 lines of C++ code, and the TEG generation in 3,194 lines of C++ code. In addition, TSV uses the `Z3` theorem prover [9] both for checking path feasibility during symbolic execution, and for simplifying symbolic variable values during TEG construction. `NuSMV` is used for model checking of the refined TEG [12]. In the case of a safety violation, `Z3` is used to find a concrete input for the path predicate corresponding to the offending output.

B. Control System Case Studies

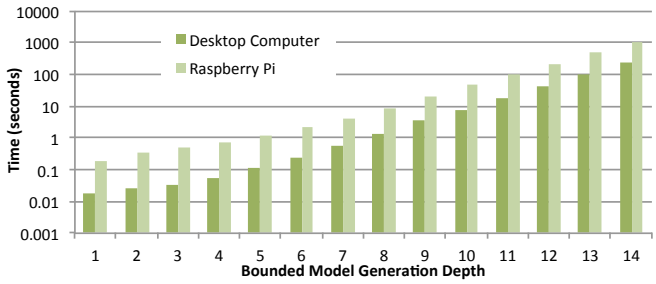
To make sure that TSV can be used for practical safety verification of real-world infrastructures, we deployed TSV on several real-world Siemens PLC programs for different

[6] In our implementations, before the logic negation, the given LTL formula is first reduced to the corresponding computation tree logic formula [3]. We find the details outside the scope of this paper and the interested reader is referred to [8].

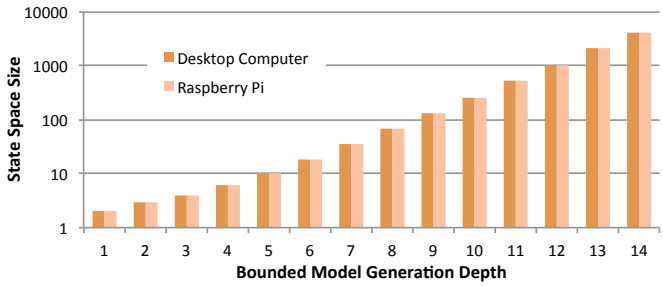
[7] To support other programming languages, a new source code lifter needs to be developed to generate the ILIL code. However, due to the syntactical similarities between most of existing PLC programming languages, the lifter may not be needed to be developed from scratch.



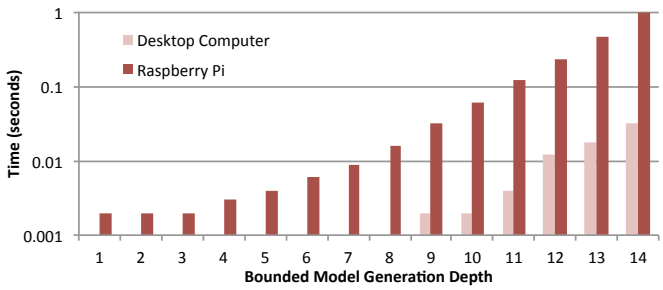
(a) Initial Model Creation



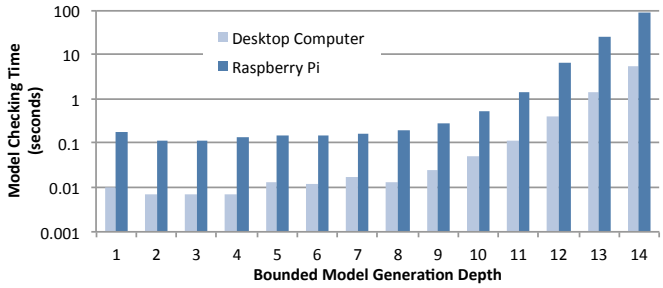
(b) Temporal Execution Graph Generation



(c) Temporal Execution Graph Cardinality



(d) Model Translation



(e) Symbolic Model Checking

Fig. 5. Performance Analysis of the Traffic Light Control System.

industrial control system settings. Our examples are runnable on several of the most popular PLC architectures[8].

- **PID controller.** (Proportional Integral Derivative) The most common type of controller for real-valued states. A PID controller attempts to minimize the error between the actual state, e.g., the temperature in a room, and a desired state. This is done by adjusting a controlled quantity, e.g., heating element, by a weighted sum of the error, and its integral and derivative.
  - *Safety requirement:* (*i*.) The controlled quantity may not exceed a constant value.
- **Traffic light.** Traffic lights at a four way intersection are governed by Boolean output signals, with a single Boolean value for each color and direction. Light changes are timer-based.
  - *Safety requirements:* (*i*.) Orthogonal green lights should not be ON simultaneously, i.e., $G\neg(g_1 \wedge g_2)$ where Boolean variable g_i denotes the i-th green light. (*ii*.) A red light should always be proceeded by a yellow light.
- **Assembly way.** Items are moved down an assembly line by a belt. A light barrier detects when an item has arrived at an assembly station, and an arm moves to hold the item in place. A part is then assembled with the item, and the barrier removed. The process repeats further down the line.
  - *Safety requirements:* (*i*.) No arm can come down until the belt stops moving. (*ii*.) The belt should not move while the arm is down.
- **Stacker.** A series of items are loaded onto a platform by a conveyor belt. Once a light barrier detects that the platform is full, the items are combined by a melter, and the resulting product is pushed onto a lift and lowered for removal.
  - *Safety requirements:* (*i*.) The product should never be pushed out while the melter is working. (*ii*.) No more items should be loaded once the platform is full.
- **Sorter.** A conveyor belt passes packages by a barcode scanner. Depending on the scanned code, one of three arms extends to push the package into an appropriate bin.
  - *Safety requirements:* (*i*.) No more than one arm should extend at each time instant. (*ii*.) An arm extends only after the barcode scanning is complete.
- **Rail Interlocking.** As opposed to the other programs, which drive the actions of a system, a railway interlocking checks that a sequence of engineer commands will not cause conflicting routes between two or more trains.

[8]Specifically, the Instruction List samples run on Siemens and Rockwell PLCs accounting for 50% of PLC market share [28].

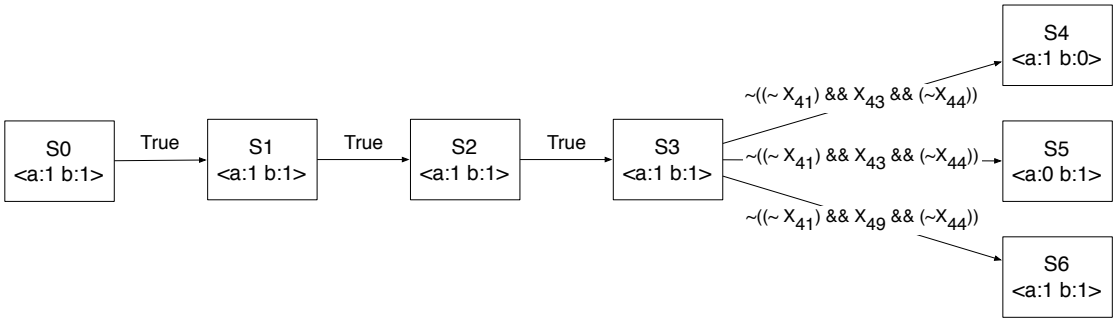


Fig. 6. Generated Temporal Execution Graph (model checking bound = 4).

- *Safety requirements:* (*i*.) There should never be conflicting routes. (*ii*.) No inputs are read after the checking procedure starts execution.

C. Example Safety Violation

To demonstrate the full usage of TSV, we show the steps that occur when attempting to upload code containing a safety violation. For this example, we modified the traffic light controller to switch both directions to green after cycling through the correct states once. Specifically, we appended the following code to the traffic light program.

```
... original program ...

RESET       ;; Reset logic accumulator.
A M 0.5     ;; Check for trigger state.
JC ATTACK   ;; Jump to attack code (if triggered).
JMP END     ;; Skip attack code.
ATTACK:
SET
= Q 0.0     ;; Set first green light.
= Q 0.3     ;; Set second green light.
END: NOP
```

The malicious program was analyzed by TSV against an interlock property prohibiting both lights from being simultaneously green. The model checker produced the concrete counterexample:

Cycle	Timer					
	1	2	3	4	5	6
1	f	f	f	f	f	t
2						f

This states that a violation was detected on the scan cycle where light timers 1-5 are false, and timer 6 switches from true to false. The next step is to identify the line of code where the violation occured. First, the ILIL interpreter preloads the concrete counterexample values for each timer variable. Next, the ILIL version of the program is instrumented with an assertion of the violated property after each line:

```
assert load(mem, [Q::0::0::0::0]) == 0 : reg1_t ||
       load(mem, [Q::0::0::0::3]) == 0 : reg1_t;
```

This simply states that at least one of the output memory locations for green lights must be off. The instrumented program is then executed with the concrete timer values. The assertion fails exactly after the line that stores 1 in `[Q::0::0::0::3]`. If the operator so desired, an execution trace of instructions and memory values leading up to this point could also be produced. Even if the original IL program is obfuscated, the ability to execute on a concrete counterexample will quickly point system operators to the offending instruction.

The example above verifies the state invariants for a simple safety requirement only for presentation clarity. However, as discussed later, TSV verified our case study PLC programs for more complex temporal safety requirements (Section VI-B) using the execution history information across input-output scan cycles that was encoded in the generated TEG graph.

D. Performance

We measured the run times for individual TSV components while checking the safety properties for each test case. Figure 5 shows the results for a sample use case (the traffic light control system) for up to 14 steps during bounded model generation. This allows for exploration of control systems with up to 14 consecutive unique state outputs. This is significantly more than Stuxnet's malicious code, which used a state machine with three unique outputs to manipulate centrifuge speed [11][9]. One could imagine an attack that evades detection by counting to 15 before violating a safety property. In this case, any control logic capable of producing a non-repeating chain of more than 14 unique outputs could also be rejected. This bound could be set higher if required for the legitimate plant functionality. The results are shown for running TSV on a desktop computer with a 3.4 GHz processor and Raspberry Pi with a 700 MHz processor.

The initial processing of the symbolic scan cycle is shown in Figure 5(a). For all cases, this step requires less than 22*ms*. Once TSV creates the initial PLC program models, it starts building the temporal execution graph, which is the main source of overhead. Figure 5(b) shows how long TSV needs to complete the graph generation phase. The majority of time in this phase is spent performing recursive exploration of the TEG to set concrete values for atomic propositions. A complete graph generation for 14 input-output scans takes 2 and 17 minutes on a desktop computer and Raspberry Pi respectively. However, as expected, trimming the analysis horizon limit to 10 reduces the graph generation time requirement significantly—down to < 10 seconds on a desktop computer and 1 minute on Raspberry Pi. Figure 5(c) shows the corresponding state space sizes for the generated graphs.

[9]We are currently working with several parties to obtain a disassembled copy of Stuxnet's PLC code.

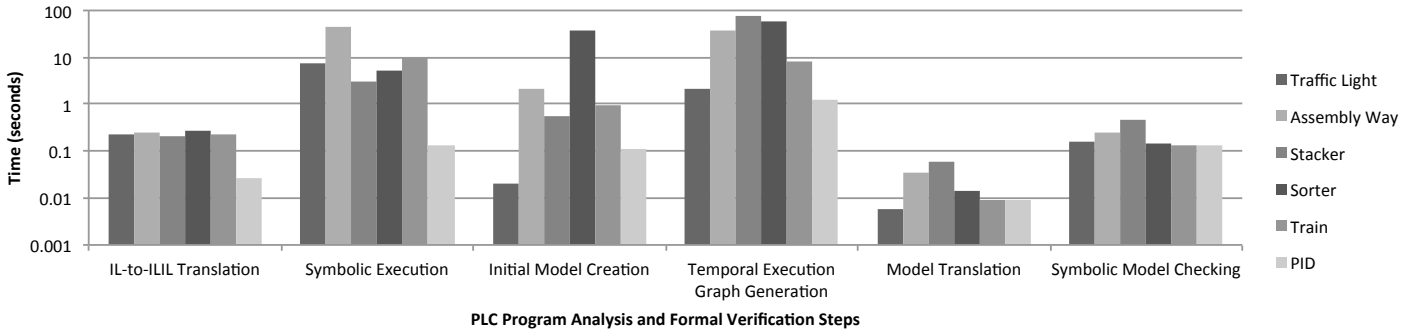


Fig. 7. Time Requirements for All Case Studies on Raspberry Pi.

The reported numbers, only $4K$ states for a full 14 horizon analysis, proves the effectiveness of the usage of symbolic execution at reducing the state space size.

Figure 6 shows a sample generated execution graph for the Assembly Way case study with a model checking bound of 4. The safety requirement included two atomic propositions a and b. Thus, each state is assigned with a pair of concrete atomic propositions, and the state transitions are labeled with the path predicates as Boolean expressions in infix order. For readability purposes, we did not include the symbolic variables and their values in each state. The atomic propositions are both true regardless of the input values in states $S0$, $S1$, $S2$, and $S3$. However, the input values affect the atomic propositions starting in state $S3$. Out of $S3$'s four possible children, $|\{a,b\}|^2 = 4$, three have been created. Only the path condition for $\langle a:0\ b:0\rangle$ was not satisfiable.

TSV runs the symbolic model checking engine on the refined and atomic proposition-level abstract temporal execution graph. Figure 5(d) shows the run times to translate the abstract TEG into the model checker's syntax, which is not a significant source of overhead. Figure 5(e) shows the time requirement results for the symbolic model verification that takes no more than 10 and 90 seconds, on the desktop and Raspberry Pi respectively. In summation, the total average overheads of less than three minutes for checking with bound 10 are within reason for an analysis that is only executed once when new code is uploaded. Of course, in the case of malicious code uploading, this bound does not affect productivity, as safety checks are done independently of plant execution under the previous, legitimate code.

We ran the same experiments for all of our case studies. Figure 7 shows how much each analysis step contributes to verification for each case study on the Raspberry Pi with bound 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 construction of the TEG for the train interlocking. This was caused by checking the feasibility of very large path predicates in the symbolic scan cycle. Despite the variance between use cases, it is clear that the net overhead is within reasonable bounds for all case studies. Figure 8 shows the state space cardinality for the generated temporal execution graphs for the case studies. It is noteworthy that there is not a direct correlation between the state space size and the overall analysis time requirement, e.g., the Train case study results in the smallest state space and yet requires the largest amount of time to finish the overall analysis.

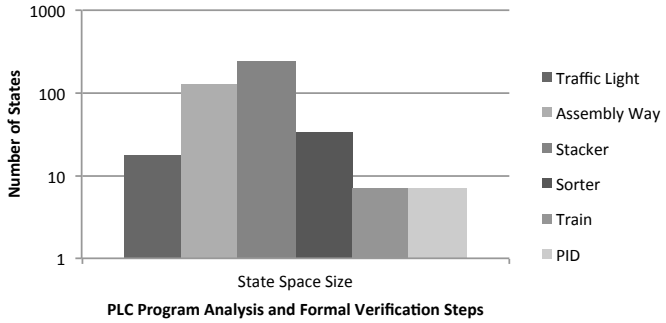


Fig. 8. State Space Size for All Case Studies on Raspberry Pi.

E. Scalability

To make sure that TSV can be used for real-world PLC code verifications, it is crucial that it can handle safety properties of realistic sizes, i.e., number of atomic propositions, efficiently. To that end, we investigated typical and frequently-used linear temporal logic-based software specification formula[10] [10], where the largest predicate includes 5 atomic propositions. Figure 9 shows the results of our experiments with TSV that can handle requirement predicates with 9 atomic propositions within approximately 2 minutes on average. It is noteworthy that handling additional safety properties only requires rerunning the atomic proposition value concretization on the temporal execution graph. Consequently, 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 (see Section VI-D).

VII. Related Work

We now review several previous approaches to safety verification of PLC software. The set of approaches reviewed here 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 to PLC analysis. Existing tools for binary analysis of general purpose programs are omitted as they do not handle PLC architectural traits like multi-indexed memories.

[10]http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml.

TABLE I. COMPARISON OF ANALYZED FEATURES WITH RELATED WORK. RELATED APPROACHES ARE ABBREVIATED: SAT=SAT SOLVING, THM=THEOREM PROVING, MOD=MODEL CHECKING

	Approach	Boolean Logic	Enum	Numeric	Cond. Branching	Function Blocks	MCR	Nested Logic	Timers	Counters	Pointers	Data Blocks	Edge Detection
Park et al. [22]	SAT	✓											
Groote et al. [14]	SAT	✓							✓				
Homer [15]	Thm	✓	✓	✓	✓								
Biha [21]	Thm	✓	✓	✓	✓				✓				
SABOT [18]	Mod	✓							✓				
Canet et al. [6]	Mod	✓	✓		✓								
TSV	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

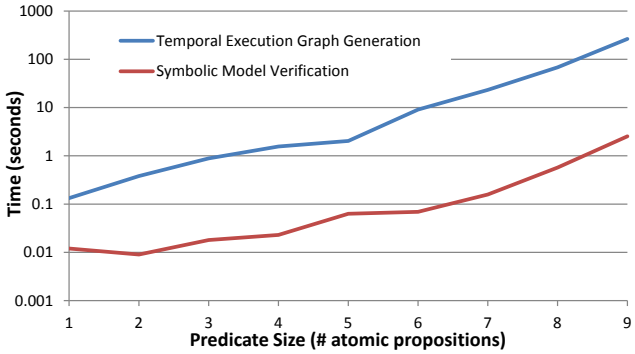


Fig. 9. Scalability Analysis for Various Predicates

The most basic approaches are those using SAT-based model checking. Park et al. [22] handle only Boolean logic. This had the advantage of being able to analyze larger sequence-based control systems, but is only narrowly applicable. Groote et al. [14] employs a similar technique, but is able to handle timers by modeling the exact wall clock execution time. This assumes that the approximate time taken for each scan cycle is known, and fails if scan cycle times vary too greatly depending on input. An improved handling of timers can be found in SABOT [18], which models their termination as a nondeterministic Boolean value. Additionally, TSV's improvements over SABOT allow for virtually all PLC programs to be analyzed, as opposed to exclusively Boolean variables and timers, which SABOT is limited to.

The two theorem proving based approaches [15], [21] handle numerical instructions, but not do not implement rules for overflow checks or mixed bit vector and integer arithmetic. The model checking approach used by Canet et al. [6], uses the same modeling as TSV for conditional branches, but does not implement numerical instructions, which lead to state space explosion. Our use of symbolic execution eliminates this explosion problem. TSV's ability to handle more PLC features (in most cases all available features) than previous work is thanks to the use of ILIL, which reduces side effects that would otherwise require many high-level modeling rules into a small set of low-level primitives.

We now review some representative past efforts at securing control systems. Stouffer et al. [31] present a series of NIST guideline security architectures for the industrial control systems that cover supervisory control and data acquisition systems, distributed control systems, and PLCs. Such guidelines are also used in the energy industry [20], [32]. It has, however, been argued that compliance with these standards can lead to a false sense of security [24], [33].

There have also been efforts to build novel security mechanisms for control systems. Mohan et al. [19] introduced a monitor that dynamically checks the safety of plant behavior. A similar approach using model based intrusion detection was proposed in [7]. Goble [13] introduce mathematical analysis techniques to evaluate various aspects, such as safety and reliability, of a given control system including the PLC devices quantitatively. However, the proposed solution focuses mainly on accidental failures and does not investigate intentionally malicious actions.

Compared with existing binary analysis tools, TSV is more apt for verifying temporal properties. For example, platforms such as BitBlaze [30], are aimed mainly at comparing binary programs, identifying malicious behavior, and exploit generation. Additionally, compared with the existing work combining symbolic execution and model checking to reduce state space explosion, TSV is the only solution enabling binary-level analysis.

PLC vendors themselves have included some rudimentary security measures into their solutions. Based on market data by Schwartz et al. [28], we studied the security measures used by PLCs accounting for 74% of market share. This included PLCs from Siemens (31%), Rockwell (22%), Mitsubishi Electric (13%), and Schneider Electric (8%). We found that all four vendors use only password authorization, typically with a single privilege level. Furthermore, password authentication measure can be disabled in all four systems. Additionally, certain Siemens systems use client-side authentication. This allows the attacker to completely bypass authentication by implementing his own client for uploading malicious code.

VIII. CONCLUSIONS

In this paper, we presented TSV, a trusted verification platform for programmable logic controllers, that allows last step security verification of the control commands right before they affect the physical system. TSV achieves a reasonable efficiency via using a new hybrid symbolic execution-enable model checking algorithm. We implemented a real-world prototype of the TSV framework on an independent Raspberry PI chip with minimal attack surface. Our evaluation results shows

that TSV can be deployed as a bump-in-the-wire portable device for efficient and practical verification of the control programs before they are uploaded to programmable logic controllers.

ACKNOWLEDGEMENTS

This material is based upon work supported by the Office of Naval Research under Award Number N00014-12-1-0462, and Advanced Research Projects Agency-Energy under Award Number 20130028603. The authors would like to thank Tim Yardley for his constructive comments and technical help with the project.

REFERENCES

[1] Shodan. http://www.shodanhq.net, 2013.

[2] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14:1–14:64, 2011.

[3] Mordechai Ben-Ari. *Mathematical logic for computer science*. Springer, 2012.

[4] Dillon Beresford. Exploiting Siemens Simatic S7 PLCs. In *Black Hat USA*, 2011.

[5] Rakesh B. Bobba, Katherine M. Rogers, Qiyan Wang, Himanshu Khurana, Klara Nahrstedt, and Thomas J. Overbye. Detecting false data injection attacks on dc state estimation. In *Proceedings of the First Workshop on Secure Control Systems (SCS)*, 2010.

[6] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. Towards the Automatic Verification of PLC Programs Written in Instruction List. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 4, pages 2449–2454, 2000.

[7] Steve Cheung, Bruno Dutertre, Martin Fong, Ulf Lindqvist, Keith Skinner, and Alfonso Valdes. Using Model-based Intrusion Detection for SCADA Networks. In *Proceedings of the SCADA Security Scientific Symposium*, 2007.

[8] E. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. In *Formal Methods in System Design*, pages 415–427. Springer-Verlag, 1994.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[10] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st international conference on Software engineering (ICSE)*, pages 411–420. ACM, 1999.

[11] Nicolas Falliere, Liam O. Murchu, and Eric Chien. W32.Stuxnet Dossier. Technical report, Symantic Security Response, October 2010.

[12] Orlando Ferrante, Luca Benvenuti, Leonardo Mangeruca, Christos Sofronis, and Alberto Ferrari. Parallel NuSMV: a NuSMV Extension for the Verification of Complex Embedded Systems. In *Computer Safety, Reliability, and Security*, pages 409–416. Springer, 2012.

[13] William M Goble. *Control Systems Safety Evaluation and Reliability*. International Society of Automation, 2010.

[14] J.F. Groote, S.F.M. van Vlijmen, and J.W.C. Koorn. The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. In *Tenth Annual Conference on Systems Integrity, Software Safety and Process Security*, pages 57–68, June 1995.

[15] Ralf Huuck. Semantics and Analysis of Instruction List Programs. *Electronic Notes in Theoretical Computer Science*, 115:3–18, 2005.

[16] John Leyden. Polish Teen Derails Tram after Hacking Train Network. http://www.theregister.co.uk/2008/01/11/tram_hack/, 2008.

[17] Patrick McDaniel and Stephen McLaughlin. Security and Privacy Challenges in the Smart Grid. *IEEE Security and Privacy*, 7:75–77, 2009.

[18] Stephen McLaughlin and Patrick McDaniel. SABOT: specification-based payload generation for programmable logic controllers. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 439–449, New York, NY, USA, 2012.

[19] Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. S3A: Secure System Simplex Architecture for Enhanced Security of Cyber-Physical Systems. http://arxiv.org, 2012.

[20] National Energy Regulatory Comission. NERC CIP 002 1 - Critical Cyber Asset Identification, 2006.

[21] Sidi Ould Biha. A Formal Semantics of PLC Programs in Coq. In *IEEE 35th Annual Computer Software and Applications Conference (COMPSAC)*, pages 118–127. IEEE, 2011.

[22] Taeshin Park and Paul I Barton. Formal Verification of Sequence Controllers. *Computers & Chemical Engineering*, 23(11):1783–1793, 2000.

[23] Dale G. Peterson. Project Basecamp at S4. http://www.digitalbond.com/2012/01/19/project-basecamp-at-s4/, January 2012.

[24] Ludovic Pièrre-Cambacédès, Marc Trischler, and Göran N. Ericsson. Cybersecurity Myths on Power Control Systems: 21 Misconceptions and False Beliefs. *IEEE Transactions on Power Delivery*, 2011.

[25] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[26] Jonathan Pollet. Electricity for Free? The Dirty Underbelly of SCADA and Smart Meters. In *Proceedings of Black Hat USA 2010*, July 2010.

[27] Henrik Sandberg, André Teixeira, and Karl H. Johansson. On security indices for state estimators in power networks. In *Proceedings of the First Workshop on Secure Control Systems (SCS)*, 2010.

[28] Moses D Schwartz, John Mulder, Jason Trent, and William D Atkins. Control System Devices: Architectures and Supply Channels Overview.

[29] Jill Slay and Michael Miller. Lessons Learned from the Maroochy Water Breach. In *Critical Infrastructure Protection*, pages 73–82. Springer, 2007.

[30] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information systems security*, pages 1–25. Springer, 2008.

[31] Keith Stouffer, Joe Falco, and Karen Scarfone. Guide to Industrial Control Systems (ICS) Security. *NIST Special Publication*, 800:82, 2008.

[32] U.S. Department of Energy Office of Electricity Delivery and Energy Reliability. A Summary of Control System Security Standards Activities in the Energy Sector, October 2005.

[33] Joe Weiss. Are the NERC CIPS making the grid less reliable. *Control Global*, 2009.

[34] Kyle Wilhoit. Who's Really Attacking Your ICS Equipment. Trend Micro, 2013.

APPENDIX

A. Example Property Check

In this section, we provide a simple property check involving TSV's main four steps executed over the traffic light controller code. For the code, and intermediate artifacts: ILIL code, symbolic scan cycle, and TEG, are all too long to fit in the space provided, so we provide illustrative examples from each.

The traffic light control program manipulates a set of Boolean variables, `Q 0.0` - `Q 0.5` representing the six lights facing two opposing directions. This is done in two steps. First, a set of internal state variables, denoted by `M 0.`x, track which state the system is in. The duration of each system state is dictated by a timer `T` x. Second, each light is turned on only if one or more state variables tell it to. As an example of the first step, the following statement checks whether a light needs to change from red to green based on a timer expiring. (The "check not first run" instruction makes sure that the PLC was not just turned on.) The lines marked "bookkeeping" are needed to decompile the program into a graphical language.

```
A(
O     T      6    ;; Check red light timer.
O     M      0.0  ;; Check already green.
ON    M      0.6  ;; Check not first run.
)
AN    T      1    ;; Check green light timer.
=     L     20.0  ;; Bookkeeping.
A     L     20.0  ;; Bookkeeping.
BLD   102         ;; Bookkeeping.
=     M      0.0  ;; Set green light state.
A     L     20.0  ;; Bookkeeping.
L     S5T#10S     ;; 10 second timer.
SD    T      1    ;; Start green light timer.
```

The green light itself is then activated by the statement:

```
A     M      0.0  ;; Check for green state.
=     Q      0.2  ;; Activate green light.
```

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

```
// (9) AND M      0.0
STA := cast(low, reg1_t, load(mem, [M::0::0::0::0]));
RLO := RLO && STA;
FC := 1 : reg1_t;
OR := 0 : reg1_t;

// (10) ST Q      0.2
OR := 0 : reg1_t;
STA := RLO;
FC := 0 : reg1_t;
mem := store(mem, [Q::0::0::0::2], RLO);
```

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

```
// The green output variable.
(declare-const Q_0_0_0_2 Bool)

// The state variable.
(declare-const M_0_0_0_0 Bool)

[M_0_0_0_0] -> (and (or (or T_6 M_0_0_0_0)
                        (not M_0_0_0_6)) (not T_1))
[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[11], 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

[11]For presentation clarity, we did not include the generated TEG graph with larger model generation bounds here.

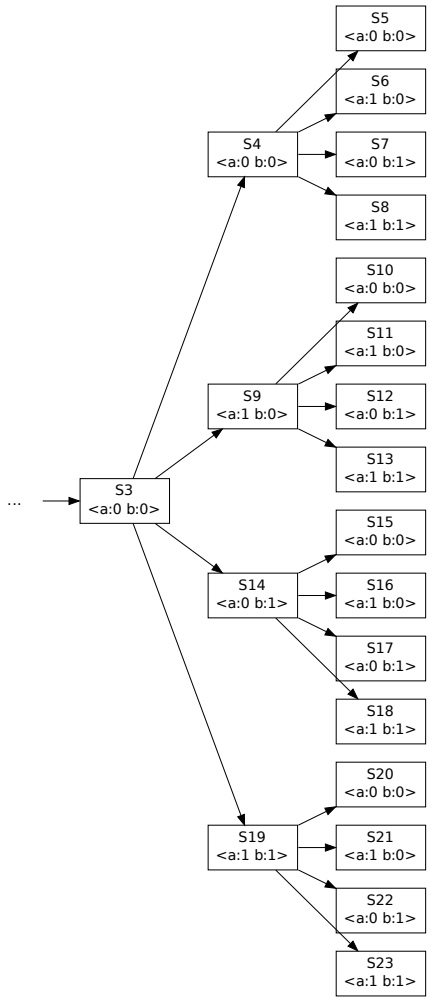


Fig. 10. Partial TEG without Symbolic State Matching

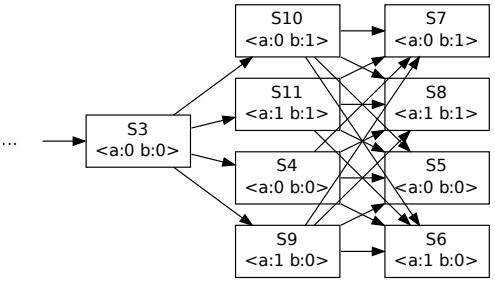


Fig. 11. Partial TEG with Symbolic State Matching

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., $G\ \neg\ (\neg a\ \&\ \neg b)$ where $a := (Q\_0\_0\_0\_2 = 0)$ and $b := (Q\_0\_0\_0\_5 = 0)$.

```
-- specification  G !(a & b)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
-> State = S0 <-
  b = TRUE
  a = TRUE
-> State: S1 <-
```

```
-> State: S2 <-
-> State: S3 <-
  b = FALSE
  a = FALSE
-> State: S10 <-
  b = TRUE
-> State: 18 <-
  a = TRUE
State trace: S0 S1 S2 S3 S10 S8
```

The model checking engine comes up with a counterexample for the code that shows the state sequence in TEG that causes the violation of the given safety requirement.

B. ILIL Semantics

The operational semantics of ILIL extend those of the Vine intermediate language to include function blocks, scoped variable resolution, hierarchical addresses, loads and stores to multi-indexed memories, and casts of hierarchical addresses.

Contexts. The ILIL machine state consists of the following contexts.

- Σ - The function call stack.
- Λ - Label to instruction number mapping.
- Π - Instruction pointer to instruction mapping.
- Φ - Function name to entry point mapping.
- Δ - The global variable context.
- ℓ - The local variable context.
- p - The instruction pointer.
- Γ - The type context.

Overview. The operational consist of instructions and expressions. The consequence of each instruction is of the form $\Sigma,\Delta,\ell,p,i \;\hookrightarrow\; \Sigma',\Delta',\ell',p',i'$, meaning the call stack, global and local variable contexts, instruction pointer, and current instruction are transformed from the left hand side to the right hand side after execution of the instruction i. Similarly, The consequence of each expression is of the form $\Delta,\ell \vdash e \Downarrow v$, meaning that under the global and local variable contexts, the expression e evaluates to the value v.

Operational semantics of instructions:

$$\frac{\Sigma = \cdot \quad \Delta \vdash e \Downarrow v \quad \Delta' = \Delta[\mathtt{x} \leftarrow v] \quad \Pi \vdash p+1 : i}{\Sigma,\Delta,\ell,p,\mathtt{x} := e \;\hookrightarrow\; \Sigma,\Delta',\ell,p+1,i}\ \textsf{assign–g}$$

$$\frac{\Sigma \neq \cdot \quad \ell \vdash e \Downarrow v \quad \ell' = \ell[\mathtt{x} \leftarrow v] \quad \Pi \vdash p+1 : i}{\Sigma,\Delta,\ell,p,\mathtt{x} := e \;\hookrightarrow\; \Sigma,\Delta,\ell',p+1,i}\ \textsf{assign–l}$$

$$\frac{\Sigma' = \Sigma;(\ell,p+1) \quad \Delta,\ell \vdash e \Downarrow v \quad \Phi \vdash f : p' \quad \Pi \vdash p' : i}{\Sigma,\Delta,\ell,p,\mathtt{call}\ f(\mathtt{x}\ e) \;\hookrightarrow\; \Sigma',\Delta,\{\mathtt{x} : v\},p',i}\ \textsf{call}$$

$$\frac{\Pi \vdash p' : i}{\Sigma;(\ell',p'),\Delta,\ell,p,\mathtt{ret} \;\hookrightarrow\; \Sigma,\Delta,\ell',p',i}\ \textsf{ret}$$

$$\frac{\Delta,\ell \vdash e \Downarrow v \quad \Lambda \vdash v : p' \quad \Pi \vdash p' : i}{\Sigma,\Delta,\ell,p,\mathtt{jmp}\ e \;\hookrightarrow\; \Sigma,\Delta,\ell,p',i}\ \textsf{jmp}$$

$$\frac{\Delta,\ell \vdash e_1 \Downarrow 1 \quad \Delta,\ell \vdash e_2 \Downarrow v \quad \Lambda \vdash v : p' \quad \Pi \vdash p' : i}{\Sigma,\Delta,\ell,p,\mathtt{cjmp}\ e_1,e_2,e_3 \;\hookrightarrow\; \Sigma,\Delta,\ell,p',i}\ \textsf{cjmp–t}$$

$$\frac{\Delta,\ell \vdash e_1 \Downarrow 0 \quad \Delta,\ell \vdash e_3 \Downarrow v \quad \Lambda \vdash v : p' \quad \Pi \vdash p' : i}{\Sigma,\Delta,\ell,p,\mathtt{cjmp}\ e_1,e_2,e_3 \;\hookrightarrow\; \Sigma,\Delta,\ell,p',i}\ \textsf{cjmp–f}$$

$$\frac{\Pi \vdash p+1 : i}{\Sigma,\Delta,\ell,p,\mathtt{label}\ s \;\hookrightarrow\; \Sigma,\Delta,\ell,p+1,i}\ \textsf{label}$$

$$\frac{\Delta,\ell \vdash e \Downarrow 1 \quad \Pi \vdash p+1 : i}{\Sigma,\Delta,\ell,p,\mathtt{assert}\ e \;\hookrightarrow\; \Sigma,\Delta,\ell,p+1,i}\ \textsf{assert–t}$$

$$\frac{\Delta,\ell \vdash e \Downarrow 0}{\Sigma,\Delta,\ell,p,\mathtt{assert}\ e \;\hookrightarrow\; \bullet,\bullet,\{\text{“err”} : \text{“}e\text{”}\},\bullet,\bullet}\ \textsf{assert–f}$$

Operational semantics of expressions:

$$\frac{\begin{array}{c}\Delta,\ell \vdash e_1 \Downarrow v_1 \quad \Delta,\ell \vdash e_2 \Downarrow v_2 \quad \mathit{size}(v_2) = m \\ \Gamma \vdash v_1 : \mathtt{mem\_t}(\tau_{\mathit{endian}},m) \quad n = \#\ \mathit{bytes}\ \tau_{\mathit{reg}} \\ v = v_1[v_2 \ldots v_2 + n]\ \text{ in }\ \tau_{\mathit{endian}}\ \text{ order}\end{array}}{\Delta,\ell \vdash \mathtt{load}(e_1,e_2,\tau_{\mathit{reg}}) \Downarrow v}\ \textsf{load–bytes}$$

$$\frac{\begin{array}{c}\Delta,\ell \vdash e_1 \Downarrow v_1 \quad \Delta,\ell \vdash e_2 \Downarrow v_2 \quad \mathit{size}(v_2) = m+1 \\ \Gamma \vdash v_1 : \mathtt{mem\_t}(\tau_{\mathit{endian}},m) \quad \tau_{\mathit{reg}} = \mathtt{reg1\_t} \\ v = v_1[v_2]\end{array}}{\Delta,\ell \vdash \mathtt{load}(e_1,e_2,\tau_{\mathit{reg}}) \Downarrow v}\ \textsf{load–bit}$$

$$\frac{\begin{array}{c}\Delta,\ell \vdash e_1 \Downarrow v_1 \quad \Delta,\ell \vdash e_2 \Downarrow v_2 \quad \Delta,\ell \vdash e_3 \Downarrow v_3 \quad \mathit{size}(v_2) = m \\ \Gamma \vdash v_1 : \mathtt{mem\_t}(\tau_{\mathit{endian}},m) \quad n = \#\ \mathit{bytes}\ \tau_{\mathit{reg}} \\ v = v_1[v_2 \ldots v_2 + n \leftarrow v_3]\ \text{ in }\ \tau_{\mathit{endian}}\ \text{ order}\end{array}}{\Delta,\ell \vdash \mathtt{store}(e_1,e_2,e_3,\tau_{\mathit{reg}}) \Downarrow v}\ \textsf{store–bytes}$$

$$\frac{\begin{array}{c}\Delta,\ell \vdash e_1 \Downarrow v_1 \quad \Delta,\ell \vdash e_2 \Downarrow v_2 \quad \Delta,\ell \vdash e_3 \Downarrow v_3 \quad \mathit{size}(v_2) = m+1 \\ \Gamma \vdash v_1 : \mathtt{mem\_t}(\tau_{\mathit{endian}},m) \quad \tau_{\mathit{reg}} = \mathtt{reg1\_t} \\ v = v_1[v_2 \leftarrow v_3]\end{array}}{\Delta,\ell \vdash \mathtt{store}(e_1,e_2,e_3,\tau_{\mathit{reg}}) \Downarrow v}\ \textsf{store–bit}$$

$$\frac{\Delta,\ell \vdash e_1 \Downarrow v_1 \quad \Delta' = \Delta[\mathtt{x} \leftarrow v_1] \quad \Delta',\ell \vdash e_2 \Downarrow v}{\Delta,\ell \vdash \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2\ \mathtt{end} \Downarrow v}\ \textsf{let}$$

$$\frac{\Delta,\ell \vdash e_1 \Downarrow v_1 \quad \Delta,\ell \vdash e_2 \Downarrow v_2 \quad v = v_1 \diamond_b v_2}{\Delta,\ell \vdash e_1 \diamond_b e_2 \Downarrow v}\ \textsf{bop}$$

$$\frac{\Delta,\ell \vdash e_1 \Downarrow v_1 \quad v = \diamond_u v_1}{\Delta,\ell \vdash \diamond_u e_1 \Downarrow v}\ \textsf{uop}$$

$$\frac{\Delta,\ell \vdash e \Downarrow v_1 \quad v = \text{higher } \tau_{\mathit{reg}} \text{ bits of } v_1}{\Delta,\ell \vdash \mathtt{cast}(\mathtt{high},\tau_{\mathit{reg}},e) \Downarrow v}\ \textsf{cast–u}$$

$$\frac{\Delta,\ell \vdash e \Downarrow v_1 \quad v = \text{lower } \tau_{\mathit{reg}} \text{ bits of } v_1}{\Delta,\ell \vdash \mathtt{cast}(\mathtt{low},\tau_{\mathit{reg}},e) \Downarrow v}\ \textsf{cast–l}$$

$$\frac{\Delta,\ell \vdash e \Downarrow v_1 \quad v = v_1 \text{ sign-extended to } \tau_{\mathit{reg}} \text{ bits}}{\Delta,\ell \vdash \mathtt{cast}(\mathtt{signed},\tau_{\mathit{reg}},e) \Downarrow v}\ \textsf{cast–s}$$

$$\frac{\Delta,\ell \vdash e \Downarrow v_1 \quad v = v_1 \text{ zero}-\text{extended to } \tau_{\mathit{reg}} \text{ bits}}{\Delta,\ell \vdash \mathtt{cast}(\mathtt{unsigned},\tau_{\mathit{reg}},e) \Downarrow v}\ \textsf{cast–u}$$

$$\frac{\Delta,\ell \vdash e \Downarrow v_1 \quad v = \mathsf{pack}(\mathsf{v_1})}{\Delta,\ell \vdash \mathtt{cast}(\mathtt{ptr},\mathtt{addr\_t},e) \Downarrow v}\ \textsf{cast–ptr}$$

$$\frac{\Delta,\ell \vdash e \Downarrow v_1 \quad v = \mathsf{unpack}(\mathsf{v_1})}{\Delta,\ell \vdash \mathtt{cast}(\mathtt{addr},\mathtt{reg32\_t},e) \Downarrow v}\ \textsf{cast–addr}$$

$$\frac{\ell \vdash \mathtt{x} : v}{\ell \vdash \mathtt{x} \Downarrow v}\ \textsf{var–local}$$

$$\frac{\Delta \vdash \mathtt{x} : v \quad \mathtt{x} \notin \ell}{\Delta,\ell \vdash \mathtt{x} \Downarrow v}\ \textsf{var}$$

$$\frac{}{\Delta,\ell \vdash v \Downarrow v}\ \textsf{value}$$