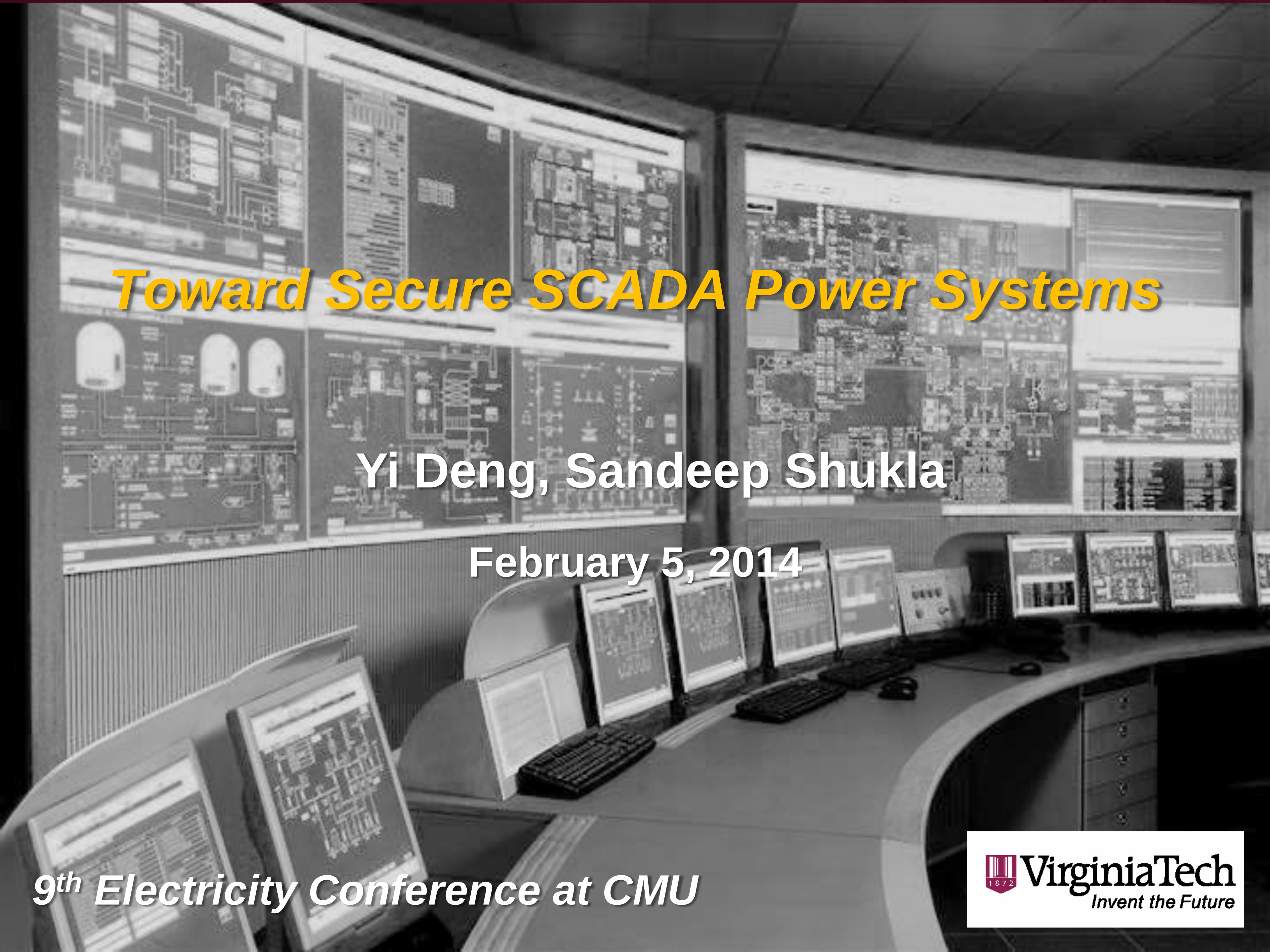


Toward Secure SCADA Power Systems

Yi Deng, Sandeep Shukla

February 5, 2014

9th Electricity Conference at CMU

VirginiaTech
Invent the Future

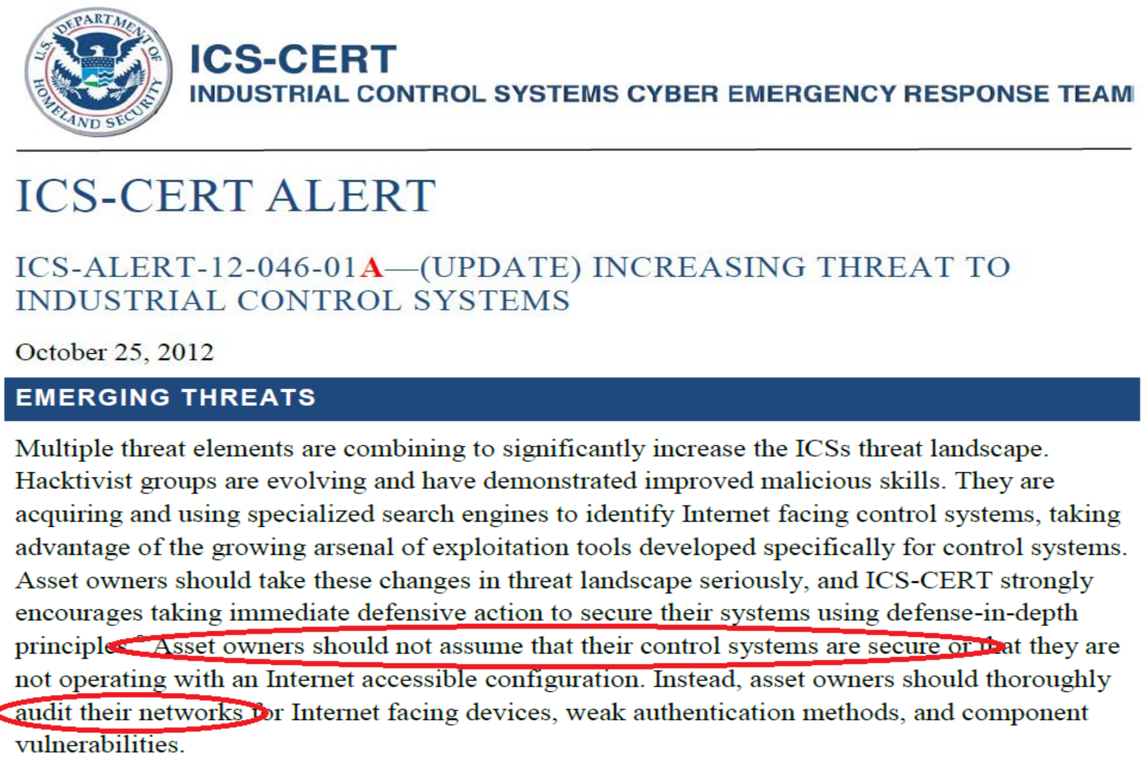
Outline

1. Introduction
2. Model-driven, behavior learning for SCADA software vulnerability assessment
3. Static analysis techniques for SCADA components for ICS Security

1: Introduction: the existing situation of SCADA system/software and their vulnerabilities

Official Security Alerts

- ICS-CERT alerts that the industrial control systems are not 100% secure, so the operators should thoroughly audit their networks.
  - Search to identify Internet facing control systems.
  - Public posting of IP address of ICS devices
- Organizations
  - DoE, DHS, GAO
  - NIST, FERC, NERC



ICS-CERT
INDUSTRIAL CONTROL SYSTEMS CYBER EMERGENCY RESPONSE TEAM

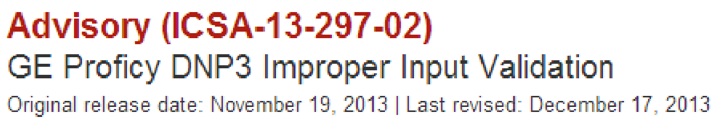
ICS-CERT ALERT

ICS-ALERT-12-046-01A—(UPDATE) INCREASING THREAT TO INDUSTRIAL CONTROL SYSTEMS

October 25, 2012

EMERGING THREATS

Multiple threat elements are combining to significantly increase the ICSs threat landscape. Hacktivist groups are evolving and have demonstrated improved malicious skills. They are acquiring and using specialized search engines to identify Internet facing control systems, taking advantage of the growing arsenal of exploitation tools developed specifically for control systems. Asset owners should take these changes in threat landscape seriously, and ICS-CERT strongly encourages taking immediate defensive action to secure their systems using defense-in-depth principles. Asset owners should not assume that their control systems are secure or that they are not operating with an Internet accessible configuration. Instead, asset owners should thoroughly audit their networks for Internet facing devices, weak authentication methods, and component vulnerabilities.



Advisory (ICSA-13-297-02)
GE Proficy DNP3 Improper Input Validation
Original release date: November 19, 2013 | Last revised: December 17, 2013

VirginiaTech Invent the Future

Vulnerabilities in SCADA

- SCADA vulnerabilities, ease of attack and impact

Category	Ease of Attack	Severity of Impact
Clear Text Communication	🟨	🟥
Account Management	🟥	🟥
Weak or No Authentication	🟧	🟧
Coding Practices	🟨	🟨
Unused Services	🟥	🟨
Network Addressing	🟨	🟧

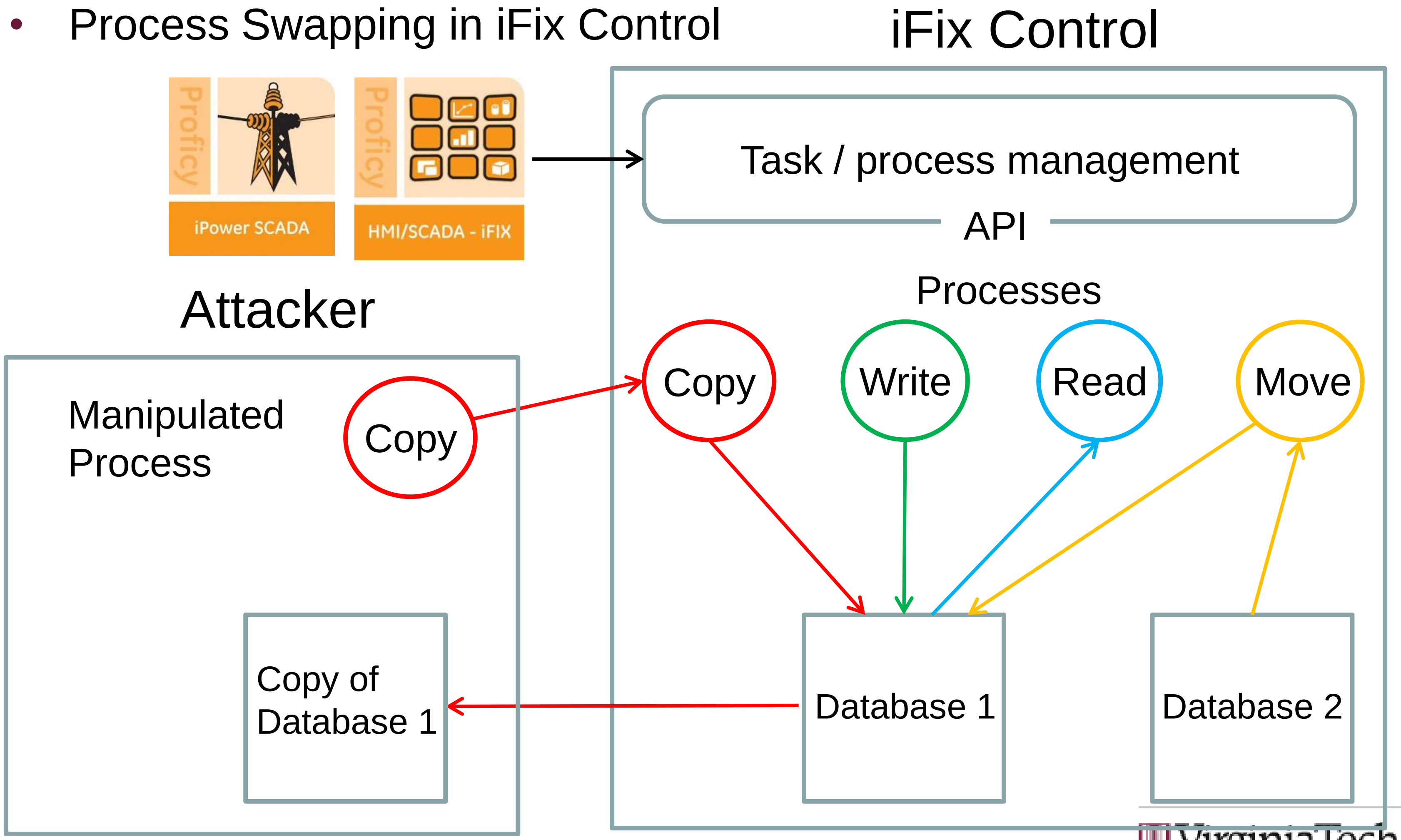
Category	Ease of Attack	Severity of Impact
Scripting and Interface Programming	🟨	🟧
Unpatched Components	🟥	🟥
Webservers and Clients	🟧	🟧
Perimeter Protection	🟨	🟥
Enumeration	🟥	🟨

**Source from NSTB (National SCADA Test Bed) Report 2006*

VirginiaTech
Invent the Future

More Sophisticated Attacks

- Process Swapping in iFix Control

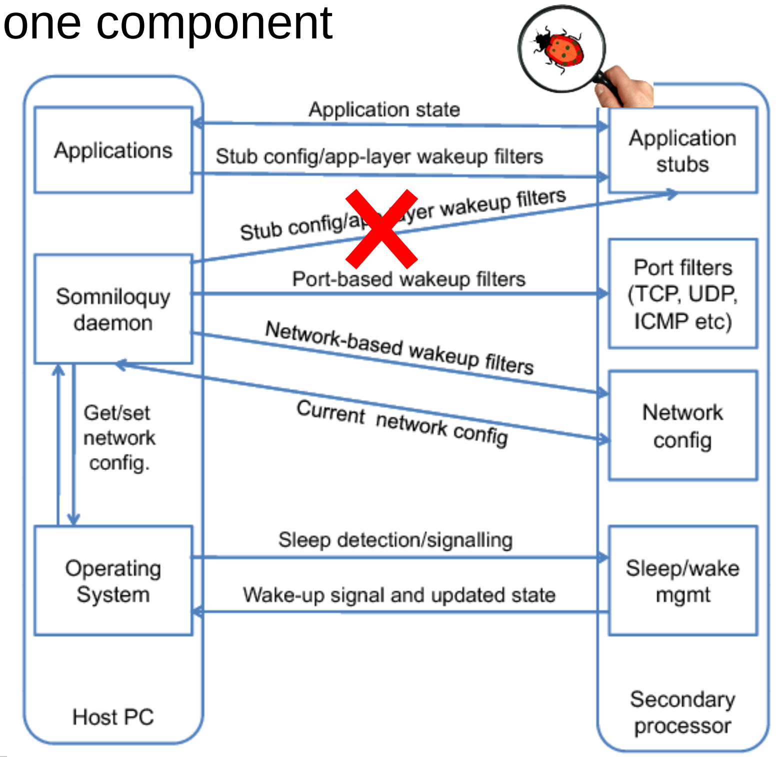


2: Model-driven, behavior learning for SCADA software vulnerability assessment

Objective

- Existing challenges in software vulnerability detection industry.
  - Not easy to test or statically analyze the pre-existing software components with vulnerabilities.
  - Not a scalable testing method
  - Vulnerabilities that might be germane to interactions between the components that are not localized in one component

- We propose to utilize machine learning techniques based on *learning automata* to discover/learn software component *interface behaviors* and then utilize them to *synthesis specific wrappers* to thwart any interactions that could enable an attacker to exploit any vulnerabilities.



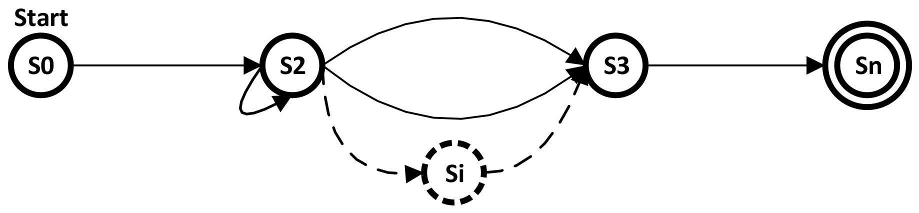
Applications
Somniloquy daemon
Get/set network config.
Operating System
Host PC
Application state
Stub config/app-layer wakeup filters
Stub config/app-layer wakeup filters
Port-based wakeup filters
Network-based wakeup filters
Current network config
Sleep detection/signalling
Wake-up signal and updated state
Application stubs
Port filters (TCP, UDP, ICMP etc)
Network config
Sleep/wake mgmt
Secondary processor

Methodology

- *Abstract models* for components with essential *interface behaviors* captured.
- In some cases, *static analysis techniques cannot be directly applied* for example, some components are often available only as executables.
- The solution for this issue is building *hypothesized finite state models* for the component and rendering the black box into grey box.
- Main functions of the hypothesized automaton:
  - Model the behavioral interface
  - Capture vulnerabilities that may exist in the original component.
  - Create composed larger system models
- Once we detect the vulnerabilities, model-based *wrapper generation* methods would allow the component to only react to safe input scenarios.

Active Learning Automata

- Active learning automata is a process that alters between a *“testing" phase* for completing the transitions relation of the model aggregated from the observed behavior, and an equivalence checking phase

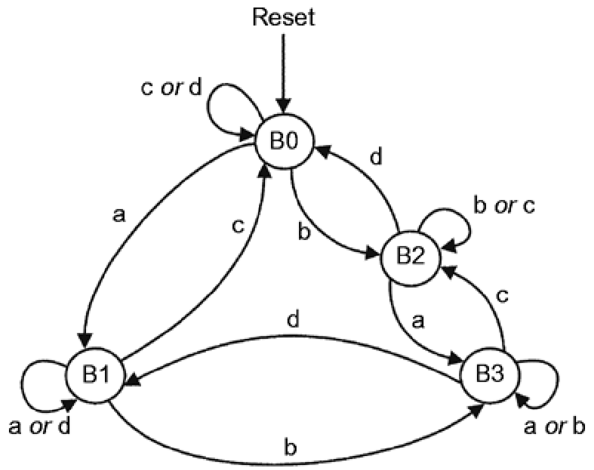


- The *first phase* incrementally constructs a model of the system
- The *second phase* is used to check if the constructed model so far captures all the possible behaviors of the original executable.
- If the checking results are “No”, it provides a counter example, which is *a trace of a behavior* that shows the distinction between the learned automata so far, and the original system.
- The learned automata us is treated as a ‘*hypothesis testing*’ which is done via methods of testing.

VirginiaTech Invent the Future

Implemented Learning Automata

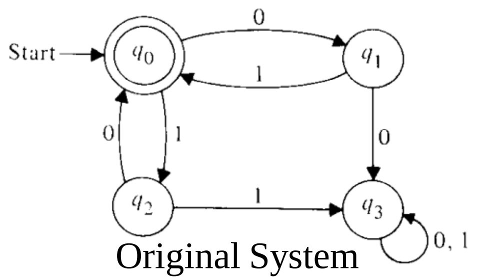
- Recent work by Steffen, el, 2011, they introduced *mealy machine models* as the target model in order to capture reactivity, and to learn some of the state transitions that are specific to input processing.
- One critical question is *how detailed* the learned mealy machine state structure should be depends on *how far* into the interface behavior one wants to capture.
- If we can build *enough depth in the state transitions* on specific inputs, one can capture problems such as buffer overflow.



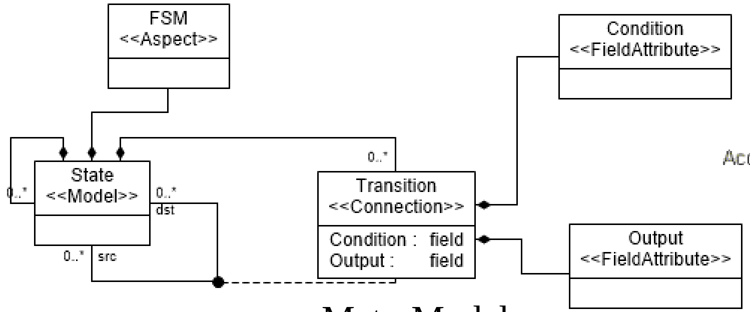
VirginiaTech
Invent the Future

Meta-model Creation & wrapper synthesis

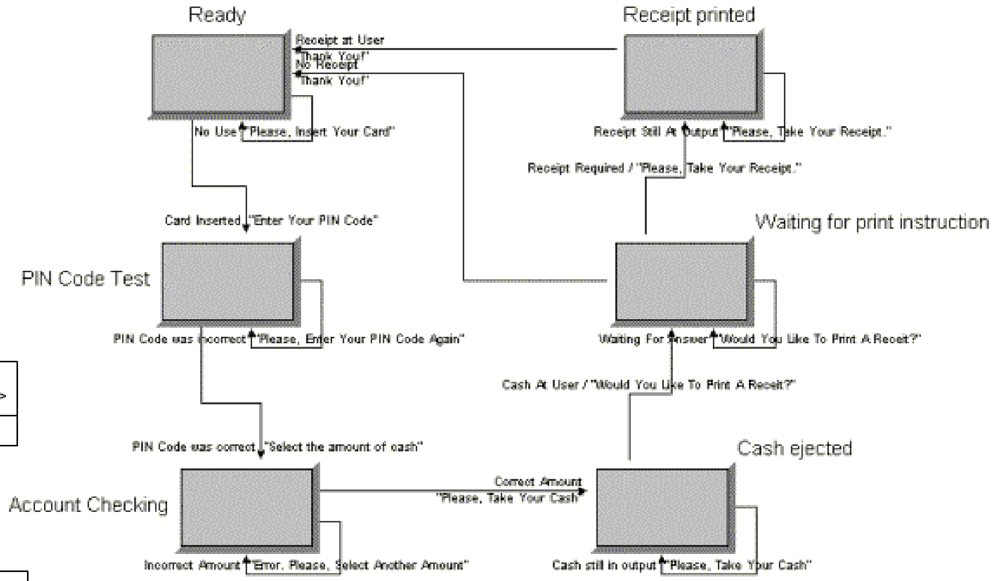
- Once a hypothesis has been considered to pass the test, we would create *meta-model* of the component in our existing meta-modeling based tool, and *generate wrapper* that would disallow inputs that could trigger unintended behavior, and there by thwarting any attacks based on that specific vulnerability.



Original System



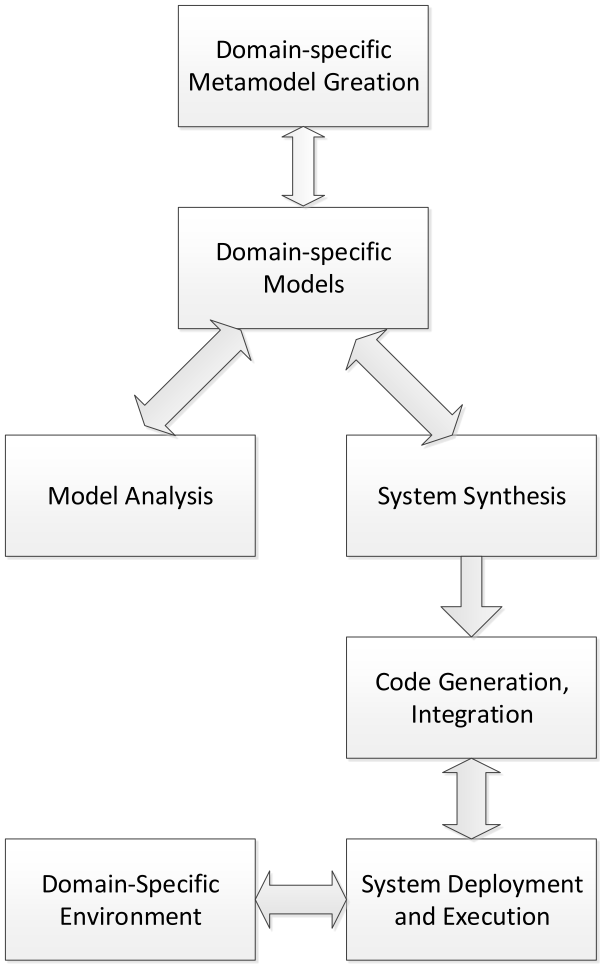
Meta-Model


Ready
Receipt printed
Receipt at User
"Thank You!"
No Receipt
"Thank You!"
No Use "Please, Insert Your Card"
Receipt Still At Output "Please, Take Your Receipt."
Receipt Required / "Please, Take Your Receipt."
Card Inserted "Enter Your PIN Code"
Waiting for print instruction
PIN Code Test
PIN Code was incorrect "Please, Enter Your PIN Code Again"
Waiting For Answer "Would You Like To Print A Receipt?"
Cash At User / "Would You Like To Print A Receipt?"
PIN Code was correct "Select the amount of cash"
Cash ejected
Correct Amount
"Please, Take Your Cash"
Account Checking
Incorrect Amount "Error, Please, Select Another Amount"
Cash still in output "Please, Take Your Cash"
System Model

VirginiaTech
Invent the Future

Model-Driven Engineering

- *Modeling* of the system and its environment from multiple, interacting aspects;
- *Automated synthesis* of design models to accelerate the modeling process;
- *Analysis* of the system using analysis/simulation tools, which may necessitate model transformations;
- *Generation* of the executable system using generator tools (that map models into implementation domain artifacts)


Domain-specific Metamodel Greation
Domain-specific Models
Model Analysis
System Synthesis
Code Generation, Integration
Domain-Specific Environment
System Deployment and Execution

VirginiaTech
Invent the Future

Defining a DSML

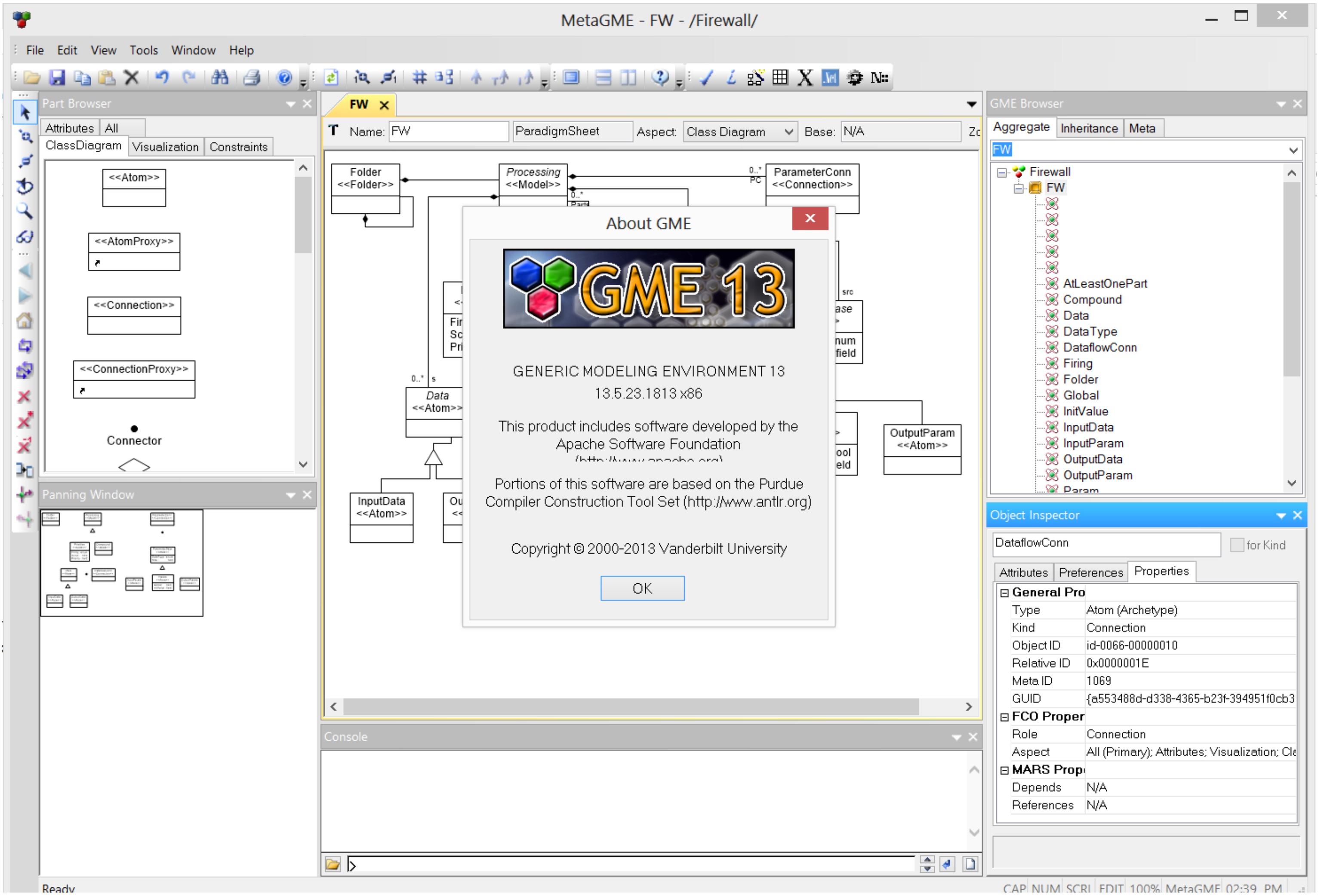
$L = < C, A, S, M_S, M_C >$

Language component	Sign	Explanation
Concrete Syntax	C	Notation for representing models ❑ Using the elements of MetaGME: UML extended with stereotypes and OCL
Abstract Syntax	A	Ontology ❑ The (important) concepts of a domain and their relations
Parsing	Mc	Mapping the concepts of A to the concepts of C ❑ Creating the concepts in A by using the concepts available in MetaGME
Semantic Domain	S	Semantics describes meaning of the models ❑ There can be multiple valid semantics
Semantic Mapping	Ms	Mapping the concepts of A to S ❑ Usually means writing an interpreter using GME's BON/MON interfaces to traverse the models

* According to the definition of GME

VirginiaTech Invent the Future

Generic Modeling Environment (GME)



About GME

GME 13

GENERIC MODELING ENVIRONMENT 13
13.5.23.1813 x86

This product includes software developed by the Apache Software Foundation (http://www.apache.org)

Portions of this software are based on the Purdue Compiler Construction Tool Set (http://www.antlr.org)

Copyright © 2000-2013 Vanderbilt University

OK

DataflowConn for Kind

General Pro	
Type	Atom (Archetype)
Kind	Connection
Object ID	id-0066-00000010
Relative ID	0x0000001E
Meta ID	1069
GUID	{a553488d-d338-4365-b23f-394951f0cb3
FCO Proper	
Role	Connection
Aspect	All (Primary); Attributes; Visualization; Cl
MARS Prop	
Depends	N/A
References	N/A

VirginiaTech Invent the Future

Meta-Model based Software Synthesis

Understand the Domain-specific Systems

⇓

Design the Meta-model

⇓

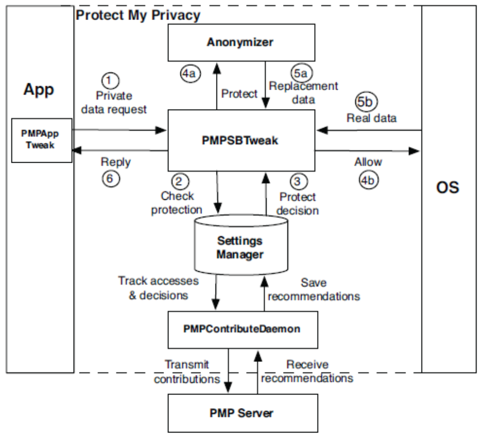
Create the Application Model

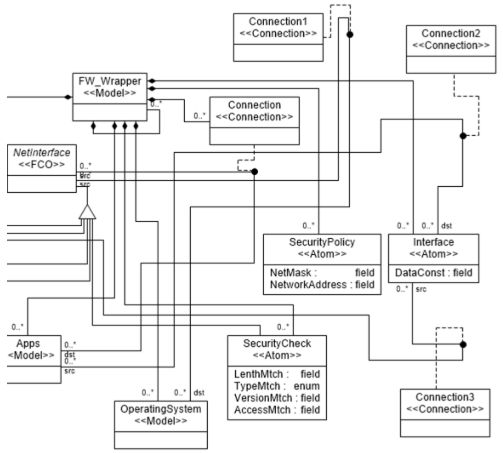
⇓

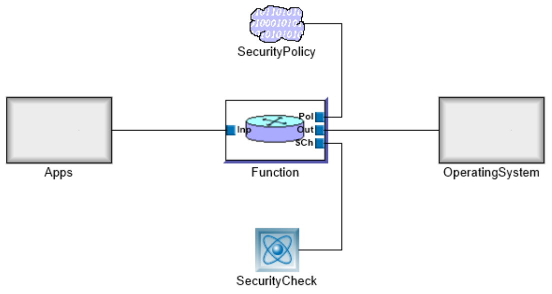
Interpreter Design

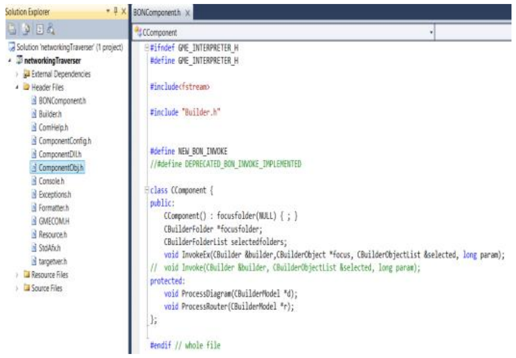
⇓

Define Constraints with Interpreter

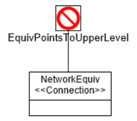


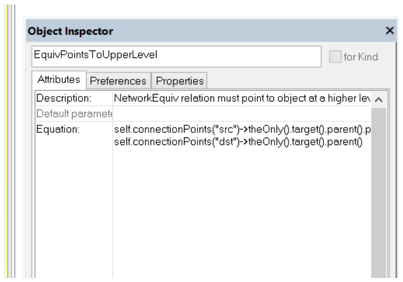






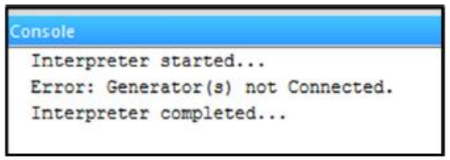
```
#ifndef GME_INTERPRETER_H
#define GME_INTERPRETER_H

#include<fstream>

#include "Builder.h"


#define NEW_BON_INVOKE
//#define DEPRECATED_BON_INVOKE_IMPLEMENTED

class CComponent {
public:
    CComponent() : focusfolder(NULL) { ; }
    CBuilderFolder *focusfolder;
    CBuilderFolderList selectedfolders;
    void InvokeEx(CBuilder &builder,CBuilderObject *focus, CBuilderObjectList &selected, long param);
//  void Invoke(CBuilder &builder, CBuilderObjectList &selected, long param);
protected:
    void ProcessDiagram(CBuilderModel *d);
    void ProcessRouter(CBuilderModel *r);
};

#endif // whole file
```





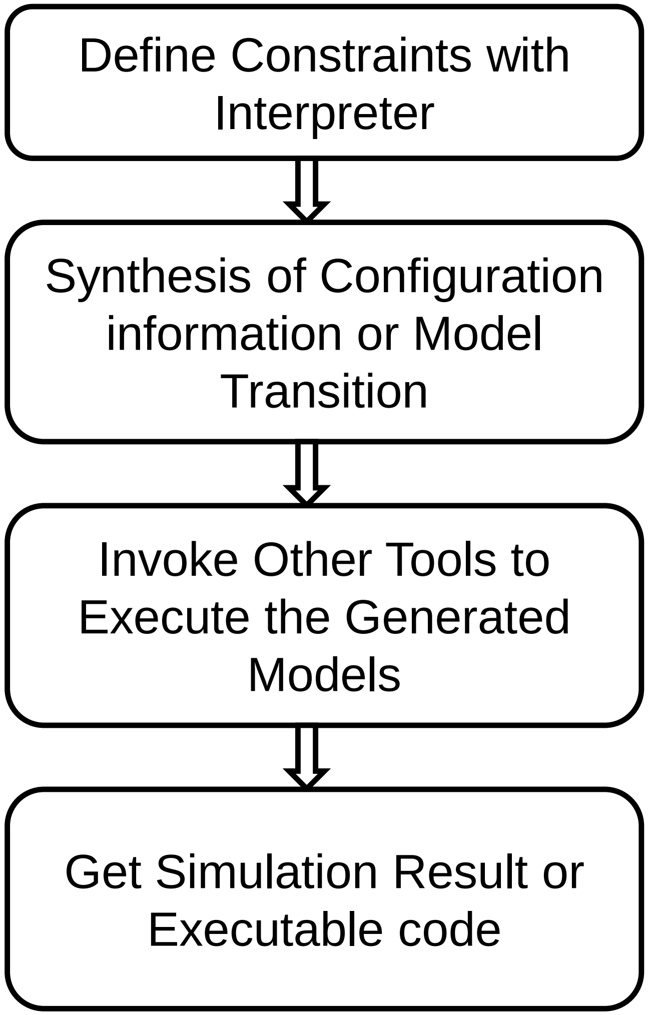
EquivPointsToUpperLevel

Description:	NetworkEquiv relation must point to object at a higher lev
Default paramete	
Equation:	self.connectionPoints("src")->theOnly().target().parent().p self.connectionPoints("dst")->theOnly().target().parent()

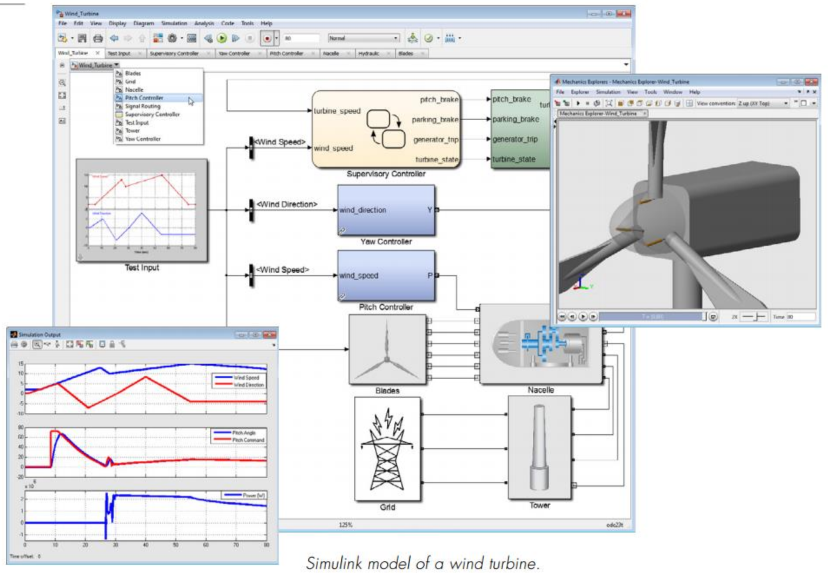


```
Interpreter started...
Error: Generator(s) not Connected.
Interpreter completed...
```

Meta-Model based Software Synthesis


Define Constraints with Interpreter
Synthesis of Configuration information or Model Transition
Invoke Other Tools to Execute the Generated Models
Get Simulation Result or Executable code

Model synthesis in the compositional modeling framework can be formulated as a search problem: given a set of *{M1, M2, ... , Mk}*, and a set of composition operators, how to select an *Md* = *Mi*||*Mj*…||*Mt* design such that a set of *{P1d, P2d, …, Pkd}* properties for *Md* are satisfied



Simulink model of a wind turbine.

Summary

- Develop *active automata learning methods* targeted to construct models that specifically models the behavior of the component in response to inputs that are hypothesized to be problematic.
- Building *hypothesized finite state models* would provide visibility into the component.
- The proposed hypothesized finite state models can model the *behavioral interface*, and *capture vulnerabilities* that may exist in the original component.
- Once vulnerabilities are detected, , *wrapper generation* using our algorithms would allow the component to only react to safe input scenarios, hence mitigating some of the vulnerabilities

VirginiaTech
Invent the Future

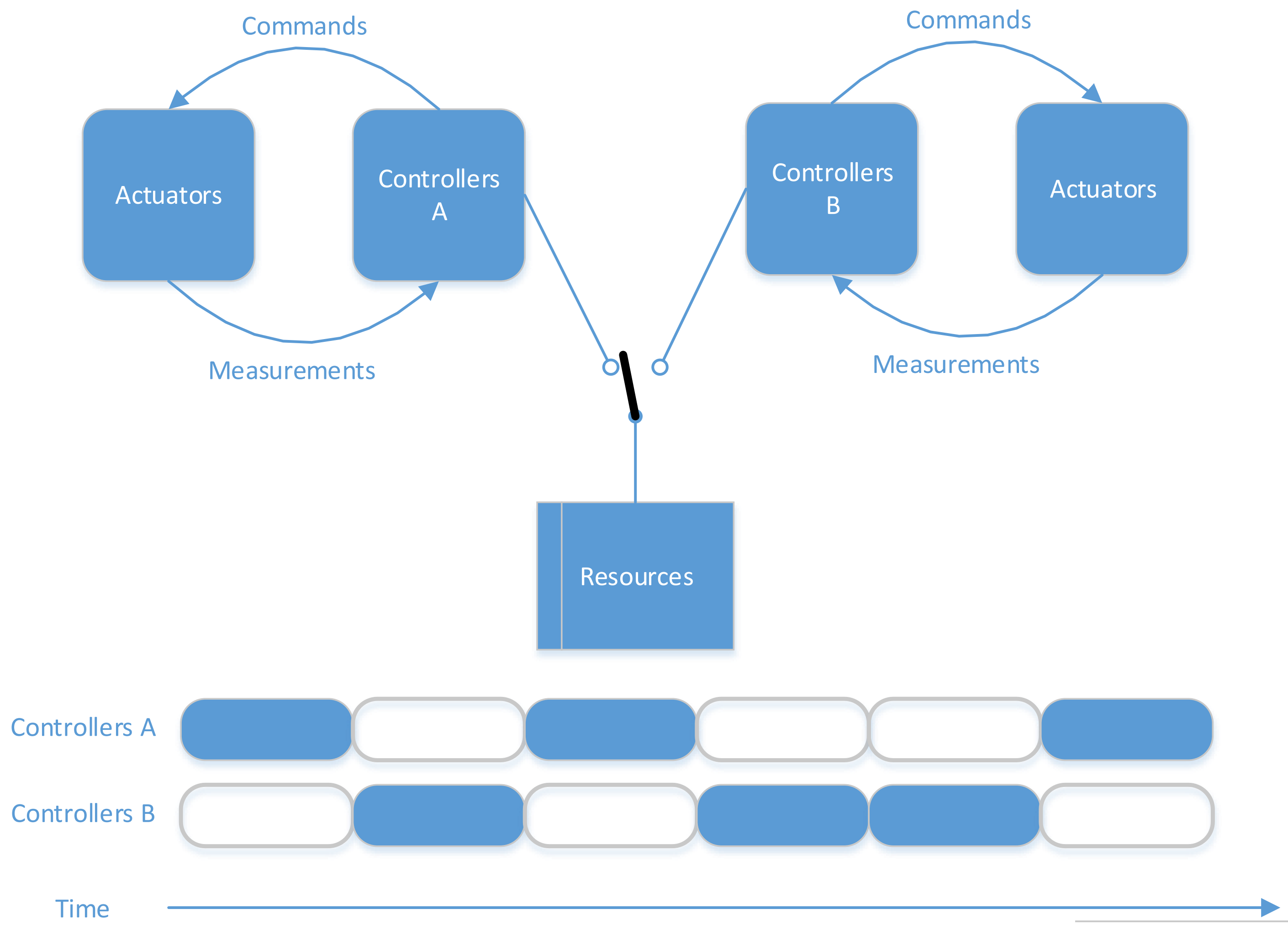
4:Static Analysis Techniques for SCADA Components for ICS Security

Objective

- *Code replacement attack* by internal employees or phishing attacks or other means of code injection has been known to be a vector for cyber-attacks. Technical method: buffer overflow, SQL injection, etc.
- We are propose to use statically analyzing techniques to investigate the legitimate control program, and constructing an *omega-regular language* based *timing signature*, which can then be periodically checked on the running components to distinguish a replaced component from the original component.
- The time signature based omegar-regular language was originally proposed by Alur, et, al in the context of real-time communication scheduling.
- In this project, we plan to take the idea further into the arena of guarding against cyber-attacks on real-time industrial systems.

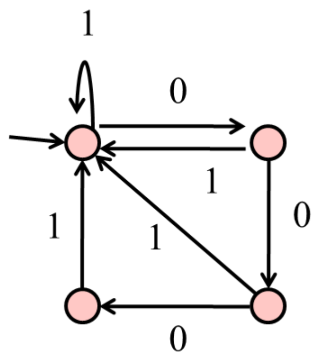
VirginiaTech
Invent the Future

Time-triggered Resource Scheduling



Automata Based Interfaces

- Generalization of the periodic interface
- Automaton (regular language) for each component
- Specifying allowed patterns of resource allocations
- Running components can be represented by timing signature based automaton

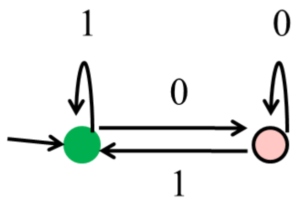


Spec: Component must get at least one slot in a window of 4 slots

0: Slot not allocated to the component
1: Slot allocated to the component

Automata Based Interfaces

- Infinite schedules specified using Buchi automata



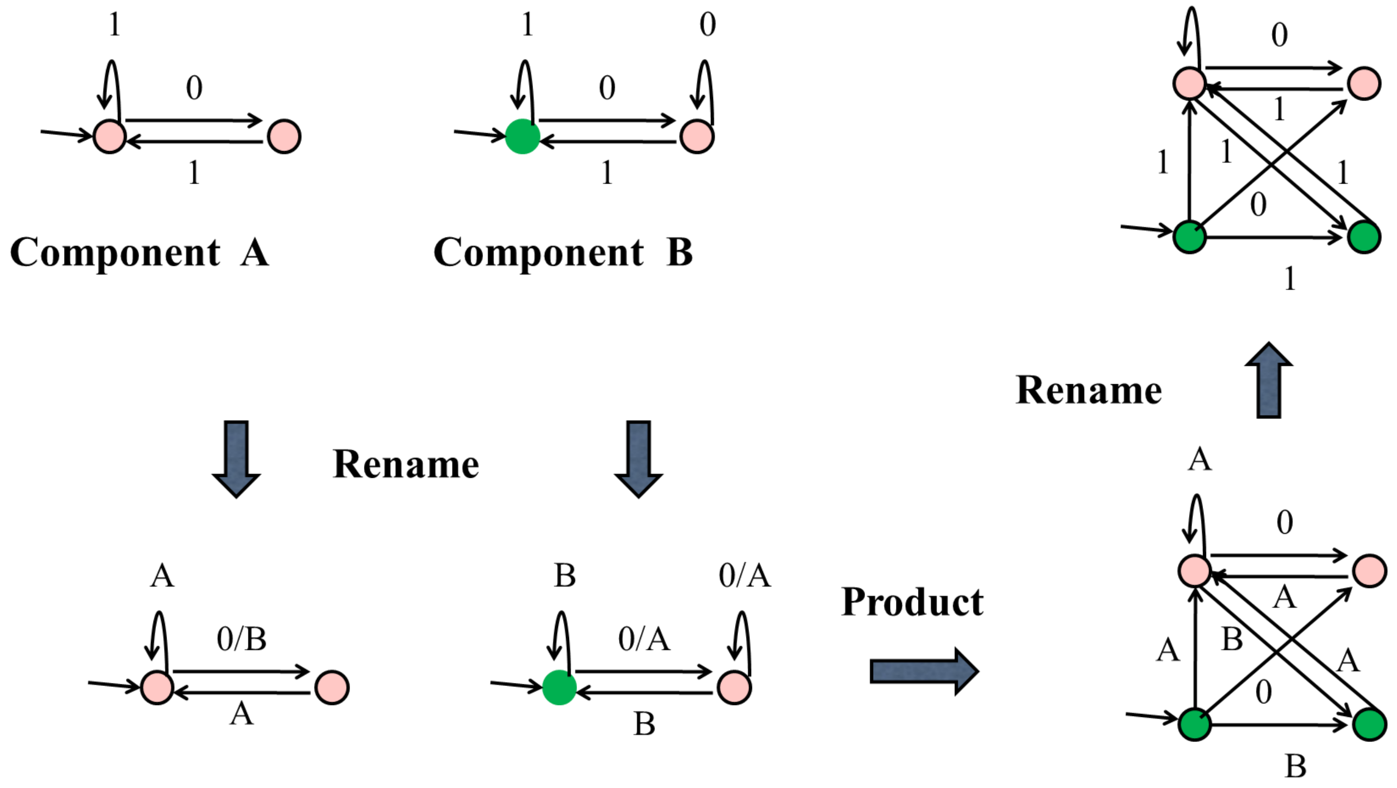
Spec: Component must get infinitely many slots

- Example specs:
- Component must get at least 2 slots in a window of 5
- Eventually component must get every alternate slot
- Periodic: (k 0’s . 1) *

- Omega-regular languages is the infinite word version of regular languages, that is recognized by Buchi automata.

Composing Specs

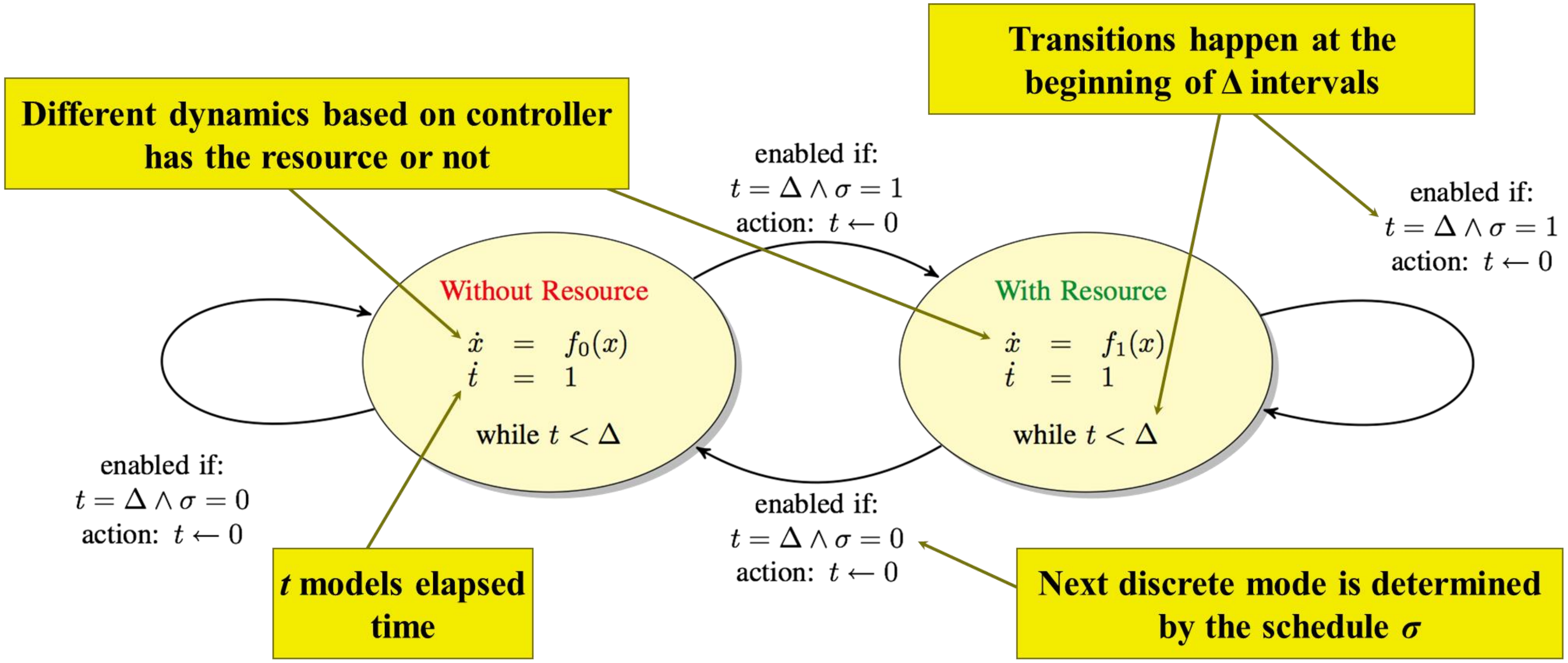
- Composition : Rename followed by intersection (product)



Schedulability Test: Check if composition of all specs is nonempty

* From “Interfaces for control components” by Rajeev Alur, University of Pennsylvania. Invited talk at Virginia Tech, Nov 2011

Analyzing stability with resource scheduling



Different dynamics based on controller has the resource or not
Transitions happen at the beginning of Δ intervals
enabled if: $t = \Delta \wedge \sigma = 1$ action: $t \leftarrow 0$
enabled if: $t = \Delta \wedge \sigma = 1$ action: $t \leftarrow 0$
Without Resource
$\dot{x} = f_0(x)$
$\dot{t} = 1$
while $t < \Delta$
With Resource
$\dot{x} = f_1(x)$
$\dot{t} = 1$
while $t < \Delta$
enabled if: $t = \Delta \wedge \sigma = 0$ action: $t \leftarrow 0$
enabled if: $t = \Delta \wedge \sigma = 0$ action: $t \leftarrow 0$
t models elapsed time
Next discrete mode is determined by the schedule σ

Challenge: Compute set of schedules σ for which system is stable

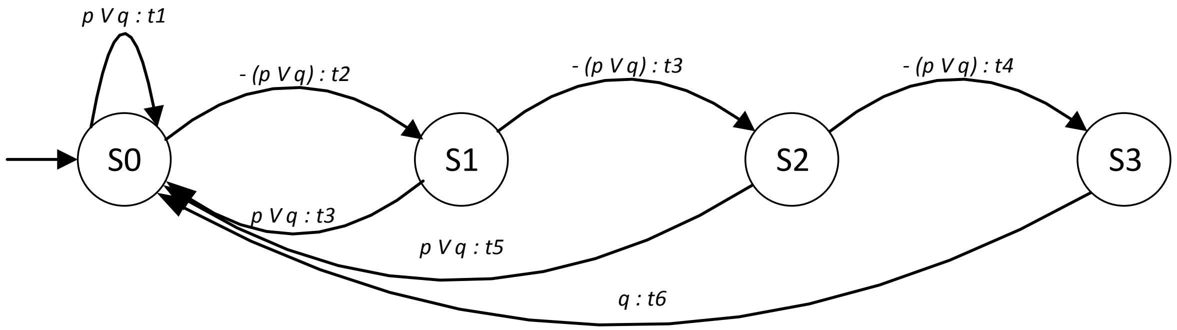
Methodology

- The original code that runs on a SCADA network as one of the control loops could be characterized by *statically analyzing* the source code.
- Then, periodically, the running executing code could be calibrated against that characterization.
- Two challenge problems:
    - What is the *right characterization* of the code?
    - How to calibrate against the known characterization if you do *not have the source code* of the executing entity?
- We claim to use the *omega regular language* to describe the scheduling patterns of the investigated control systems.
- The omega regular language is constituted by using Buchi automata which describe the scheduling patterns as *infinitely sequences*.

VirginiaTech
Invent the Future

Differences with existing methods

- Instead of using characterization of various permissible access patterns of the shared bus for scheduling analysis, we propose to use it as a *timing signature*.



- Normally, an *injected replacement component* will *differ* in the *timing behavior* in order to disrupt the real-time control system.
- It is of extremely *high probability* that the replacement component's timing signature *will not match* the original even though most common scheduling sequences may match in order to create the initial illusion that nothing has been replaced.

Implementing procedure

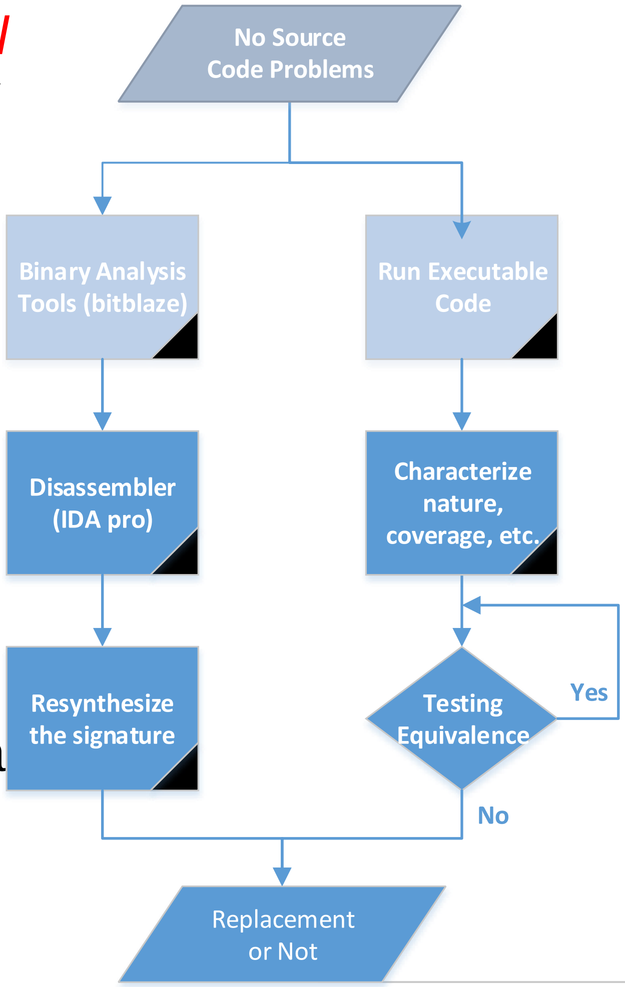
- First, implement the analysis technique and *extract such signatures*.
- Then *inject various changes* to the code, and *re-extract* the signature, and show *inequivalence*.

- However, there is one problem still remains?
- The replacement code is written and compiled by the attacker, and hence cannot be statically analyzed from the source.
- We propose two approaches:

VirginiaTech
Invent the Future

Methods to solve no source code issues

- First, we might use a *binary analysis tool* such as bitblaze by UC Berkeley (Binary Analysis for Computer Security) to create an assembly version of the executable, and then use *a disassembler* such as IDA pro, and then *resynthesize the signature*.
- Second, *testing equivalence*:
- In this approach, since we already know the timing signature, we might synthesize some crucial test sequences *to test the executable*, and find if we can disprove that the running executable is a replacement.
- We have to characterize the nature, coverage, and distinguishing power of such tests



Research Procedures

- Implement the *extraction algorithm* to obtain the omega-regular language for timing signature of control components
- *Change the source code to inject differences* that would be needed to make the control component affect the overall SCADA network
- *Re-extract the timing signature*, and *show in-equivalence* against the original
- The timing signatures are indeed useful to detect replacements or at least put *high probability value* to the suspicion that a replacement has occurred.
- In addition even *in the absence of the source code* of the replacement, we can provide some *test synthesis techniques* to find distinguishable *scheduling paths*

VirginiaTech
Invent the Future

Summary

- *Static analysis* for timing signature to thwart *code-replacement attacks* in SCADA system.
- *Statically* analyze the legitimate control program at *run-time*
- Construct an *omega-regular language based timing signature*
- The proposed omega-regular language can be periodically checked on the *running components* to distinguish a replaced component from the original component.
- Establish the omega regular timing signature as a *standard signature or key* to be identified with *all real-time control software components* on a SCADA system.
- Demonstrate a *testing methodology* to check if a given component whose source code may not be readily available, and check if it conforms to the timing signature given in omega regular form.

Thank You!

{yideng56, shukla}@vt.edu