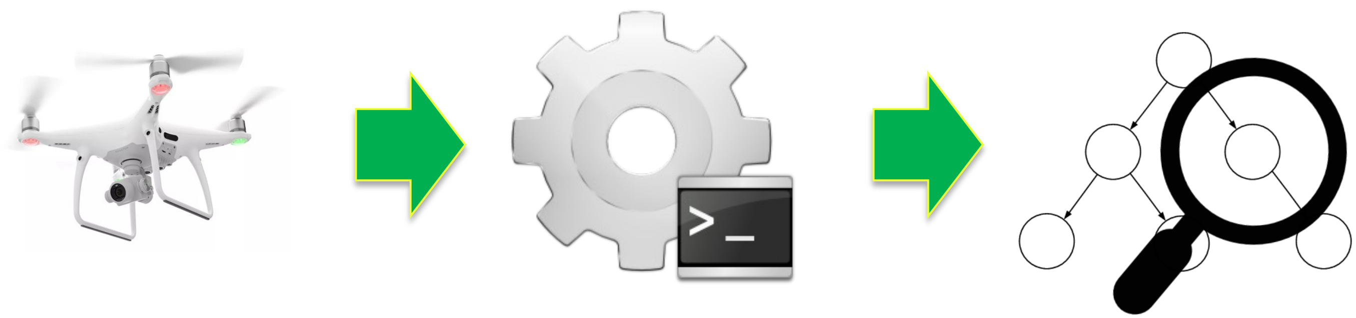
CS 6963/5963
University of Utah

Cyber-physical Systems and IoT Security

Module 2b: Intro to CPS/IoT Program Analysis



Announcements

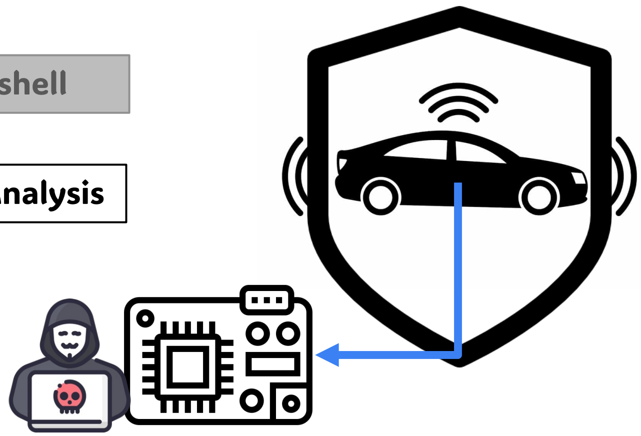
- Quiz 2 next class! (9/14)

- Reminder: Course Project Proposals Due Next Tuesday (9/19)
  - I’ll hang around after class to talk about projects

Topics covered in the first half of this course*

Module 1: Security in a nutshell

Module 2: CPS/IoT Program Analysis

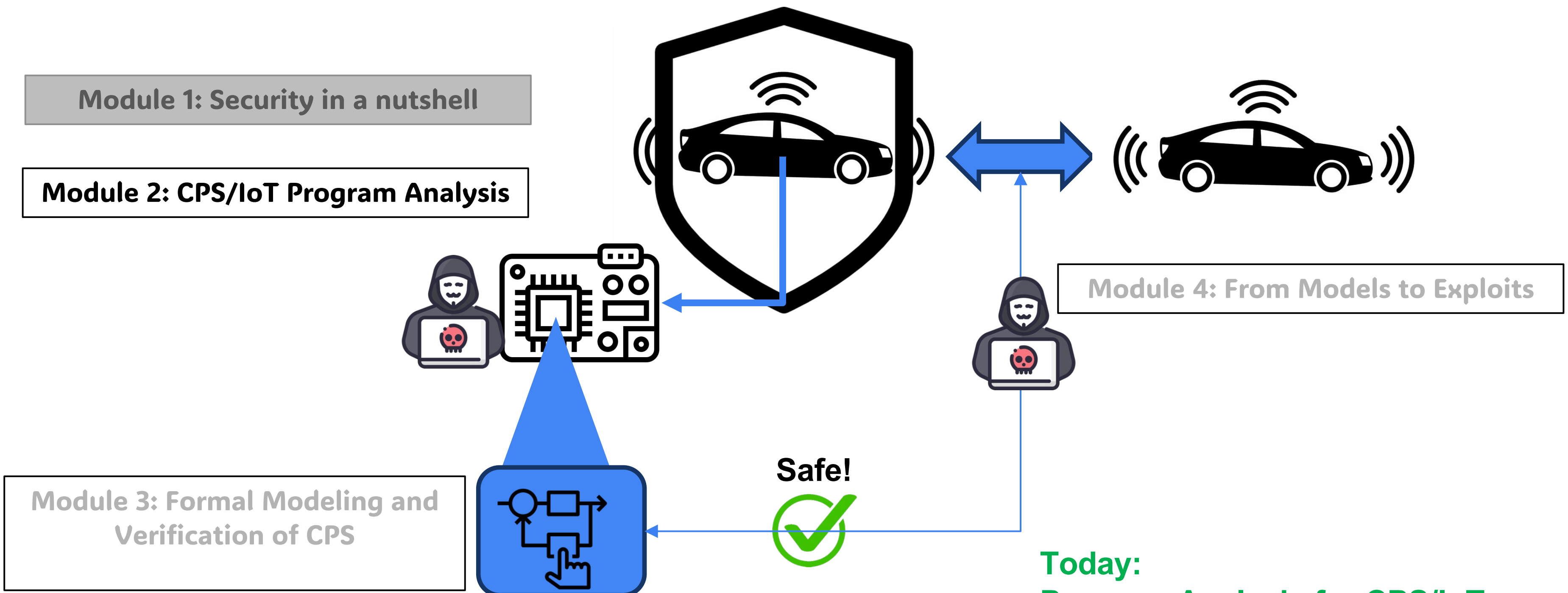


Last Lecture: Embedded Control Flow Security (Exploits)

Today:
Program Analysis for CPS/IoT
(Backbone for Modules 3 + 4)

***(subject to *slightly* change)**

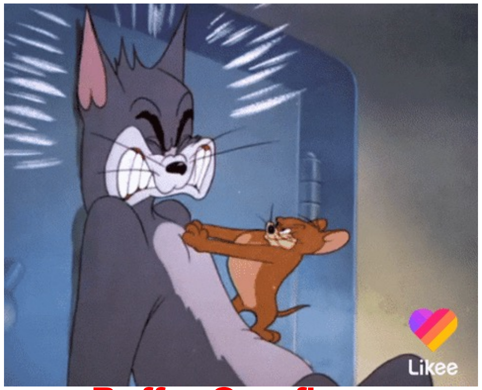
Topics covered in the first half of this course*



Today:
Program Analysis for CPS/IoT
(Backbone for Modules 3 + 4)

Recall: The Cat and Mouse Game of Control Flow Exploits

Attacks



Buffer Overflows

Return-to-Lib-C Attacks

Return-Oriented Programming

⋮

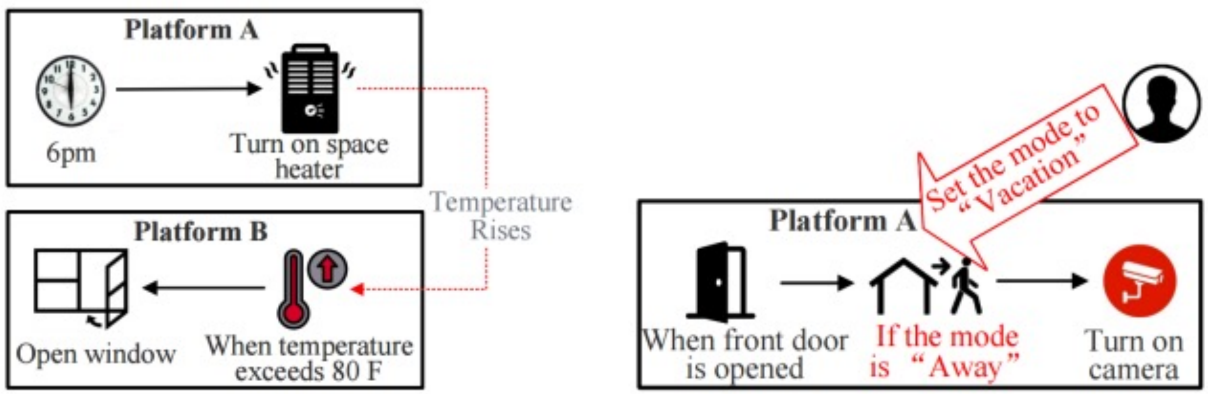
Defenses

ASLR/Stack Canaries

LibSafe

⋮

Beyond Buffer Overflows: Physical Interaction Threats



(a) Cross-App Interaction (CAI) threat

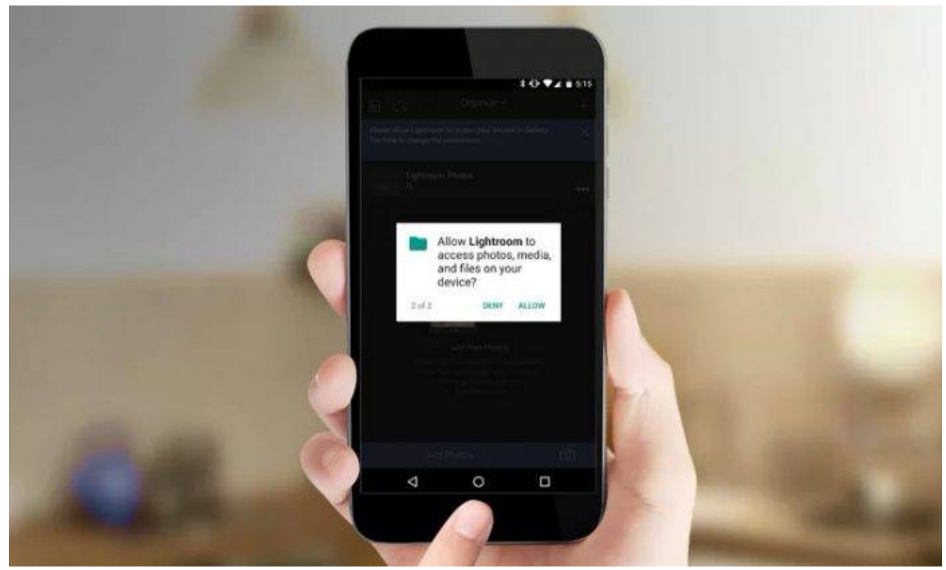
(b) Cross Manual-control and Automation Interaction (CMAI) threat

(From last lecture's class presentation)

https://www.usenix.org/system/files/usenixsecurity23-chi.pdf

Beyond Buffer Overflows: Privacy Violations

```
[capsule]
# Capsule ID and version
com.corp.capsule      1
[policy]
# from - to - action
com.corp.capsule  any  TAG_BLOCK
[contexts]
# time and geolocalized contexts
0  time-frame  OOC_BLOCK  1222333  1422333  20000
  1  geo-loc  OOC_ALLOW_LOG      25.45356      -80.51119
0  1000
[files]
# Files in capsule
/Sdcard/Documents/corp_report.pdf  0
/Sdcard/Documents/corp_report2.pdf  1
[applications]
com.corp.vpn.app
com.corp.reader.app
[connections]
vpn.corp.com
[accounts]
account@corp.com
```

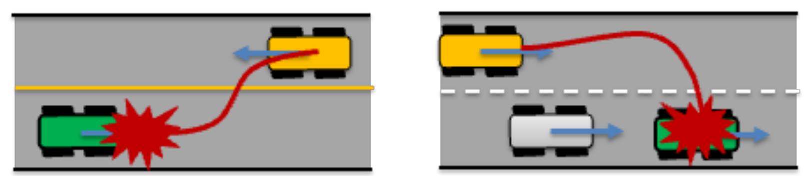


Salles-Loustau et. al, DSN ‘16 https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7579769

Beyond Buffer Overflows: Safety Violations

- Complex stochastic systems interfacing with safety rules
- We can encode safety logic into programs all we want and search for violations…
- Simple requirement of "Never Crash" is not sufficient!

Car crashing onto a Waymo AV in autonomous mode in Chandler, AZ

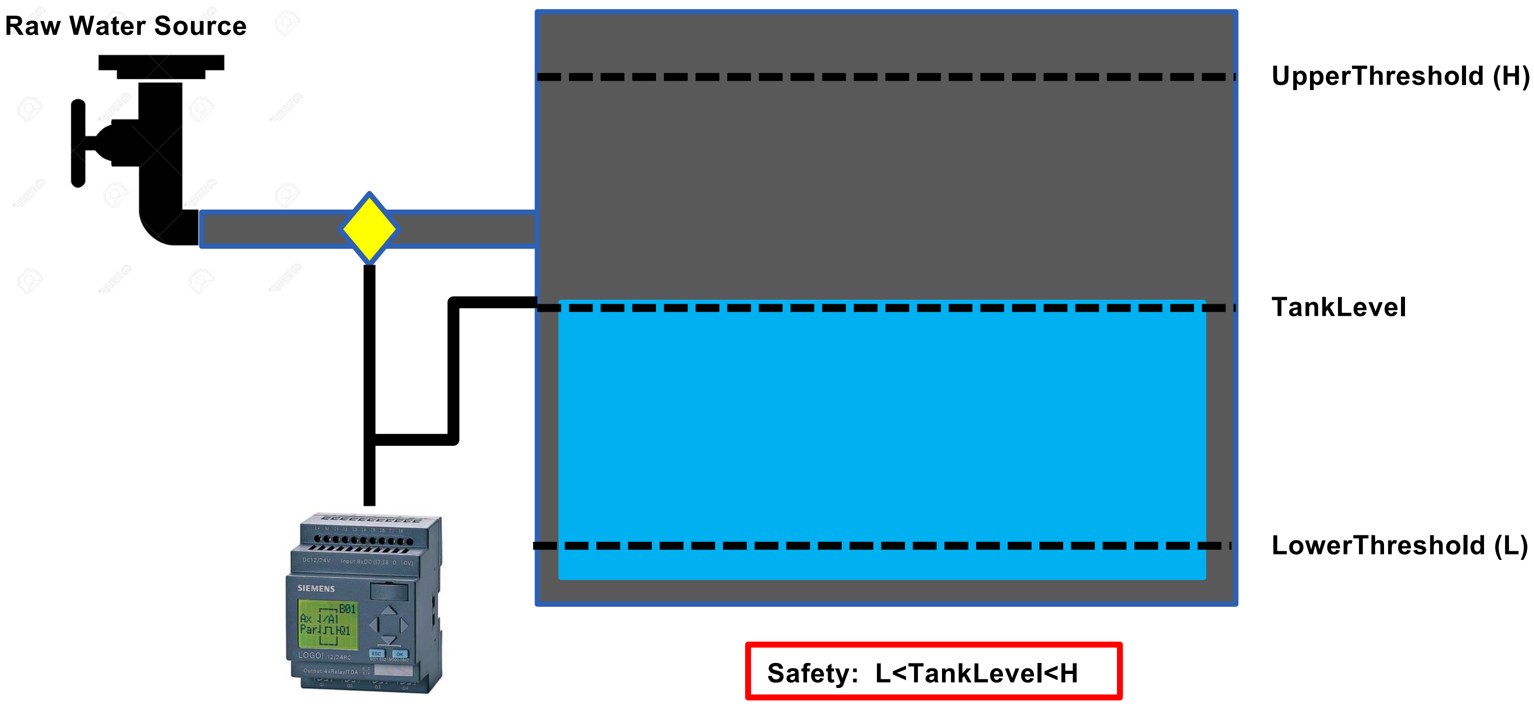


Requirement:
"Never crash" unless what?

Goal: Find Software Failures BEFORE Deployment

- User Safety and Privacy
- Cost Efficiency
- Reputation Management
- Ensure Regulatory Compliance

Example Programs We Analyze: Embedded CPS Controllers



Example Programs We Analyze: Embedded CPS Controllers

Raw W

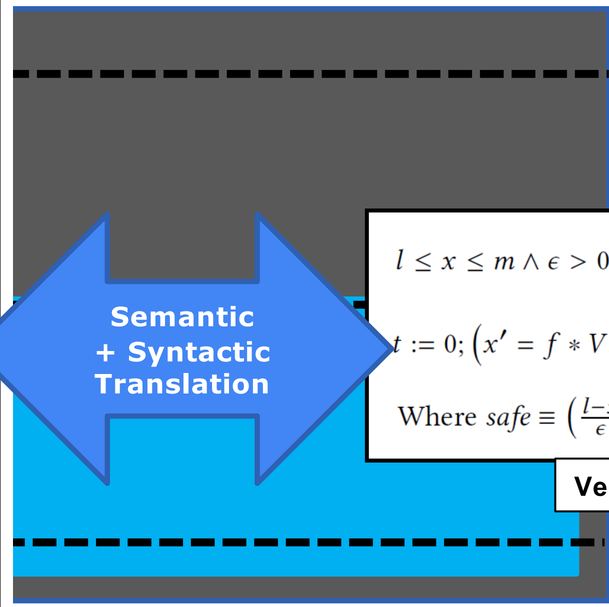
```
PROGRAM prog0
  VAR_INPUT
    f : REAL;
    x : REAL;
  END_VAR

  VAR_OUTPUT
    V : BOOL;
  END_VAR

  IF((f<((L-x)/(Tsample+Tplc)))) THEN
    V:=0; ELSE
  IF((f>((H-x)/(Tsample+Tplc)))) THEN
    V:=0; ELSE
  IF((f>=((L-x)/(Tsample+Tplc)))) THEN
    IF((f<((H-x)/(Tsample+Tplc)))) THEN
      V:=1;
    END_IF;
  END_IF;
  END_IF;
  END_IF;
END_PROGRAM

CONFIGURATION Config0
RESOURCE Res0 ON PLC
TASK Main(INTERVAL:=T#Tsamplems,PRIORITY:=0);
PROGRAM Inst0 WITH Main : prog0;
END_RESOURCE
END_CONFIGURATION
```

PLC Structured Text



$$l \leq x \leq m \wedge \epsilon > 0 \rightarrow \Big[\Big(f := *; ?safe; V := 1; \cup ?\neg safe; V := 0;$$

$$t := 0; \big(x' = f * V, t' = 1 \& x \geq 0 \wedge t \leq \epsilon\big)\Big)^*\Big] l \leq x \leq m$$

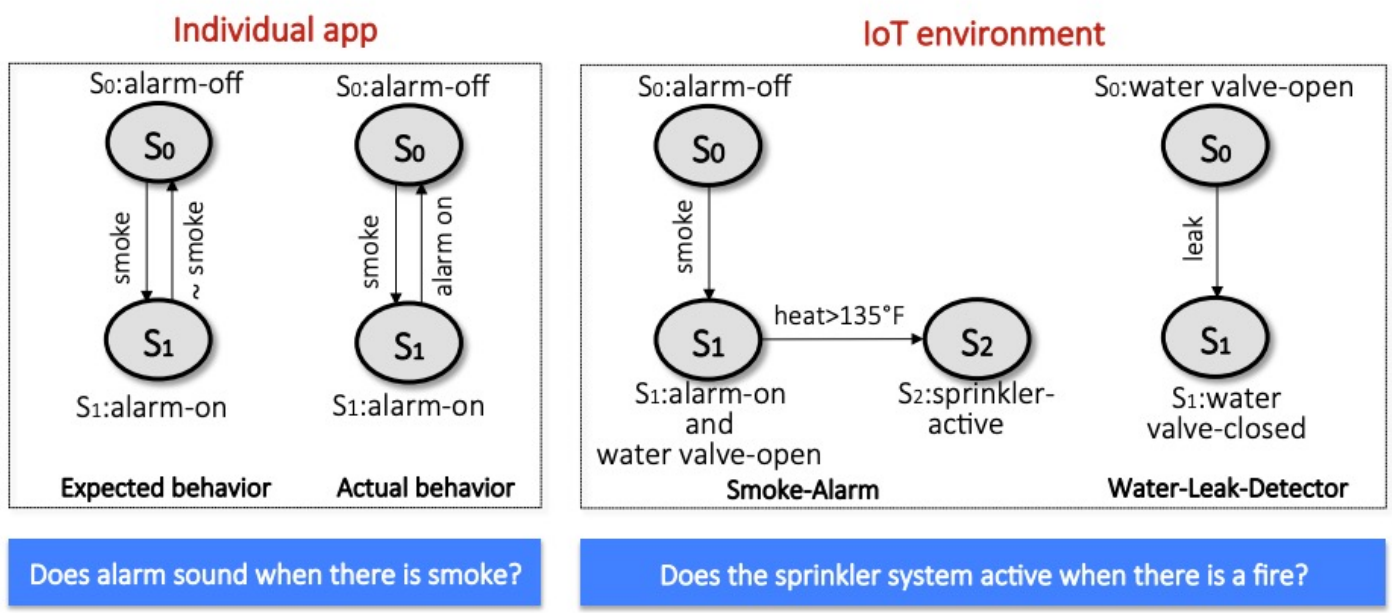
Where $safe \equiv \left(\frac{l-x}{\epsilon} \leq f \leq \frac{m-x}{\epsilon}\right)$.

Verifiable Hybrid Program Logic

Safety: L<TankLevel<H

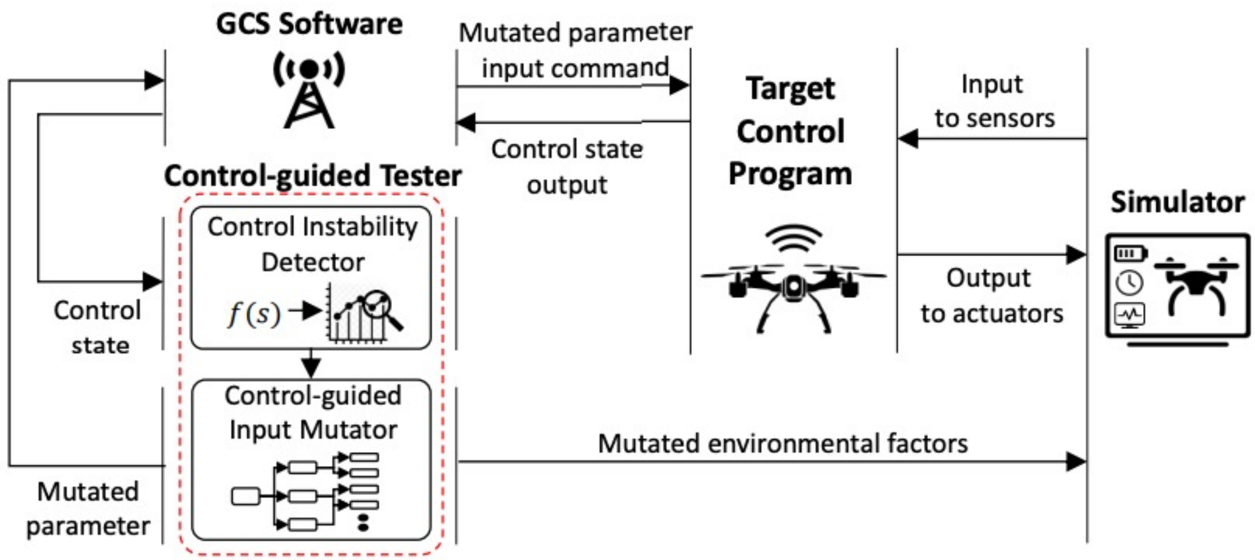
Garcia et. al, ICCPS '19

Types of Programs We Analyze: Commodity IoT Programs



Celik, et. al, USENIX ATC '18

Example Programs We Analyze: Autonomous Vehicle Controllers



RVFuzzer, Kim et. al, USENIX Security '19

Example Programs We Analyze: Learning-enabled CPS Controllers

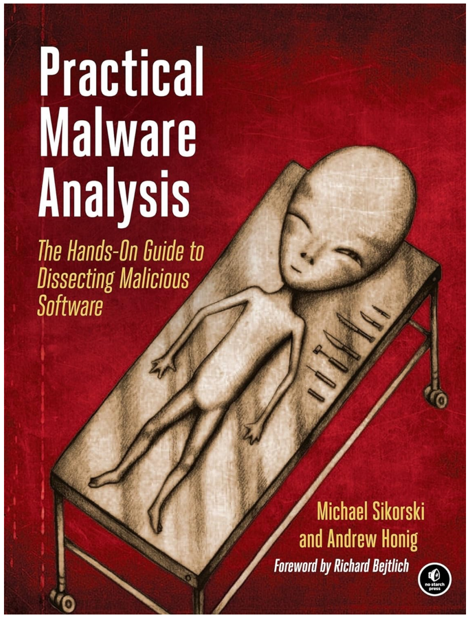
"At every time step, for all the objects (*id*) in the frame, if the object class is car with probability > 0.7, then in the next 5 frames the object (*id*) should still be detected and classified as a car with probability > 0.6"

$$\phi_2 = \Box\left(\begin{array}{c} x.\forall id@x, (C(x,id) = Car \wedge P(x,id) > 0.7) \rightarrow \\ \Box y.\left((x \leq y \wedge y \leq x+5) \rightarrow (C(y,id) = Car \wedge P(y,id) > 0.6)\right)\end{array}\right)$$

Car in adjacent lane (Red Box) becomes undetected for 3 frames (Yellow Boxes)

Example Programs We Don't Analyze in this Course (For Now): Malware

- We care more about **impact** of malware on CPS/IoT
- For the next few classes, we'll focus more on finding bugs/vulnerabilities that a malware would exploit
- **However, lots of the techniques we'll look at are used for malware analysis and reverse engineering**



Further reading:
Awesome book with
awesome malware analysis labs!

Why Not Just Add Tests in Code Manually?

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,              /* plaintext */
                       dataToSignLen,           /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

Suman Jana

Why Not Just Add Tests in Code Manually?

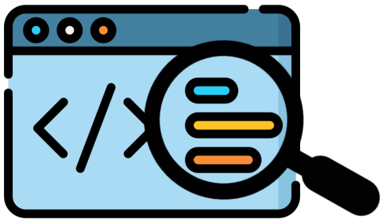
Apple “goto fail” vuln., 2014

- Time consuming
- Error-prone
- Incomplete
- Depends on quality of test cases or inputs
- Provides little in terms of code coverage

Suman Jana

Program Analysis Techniques

<u>Static Analysis</u>



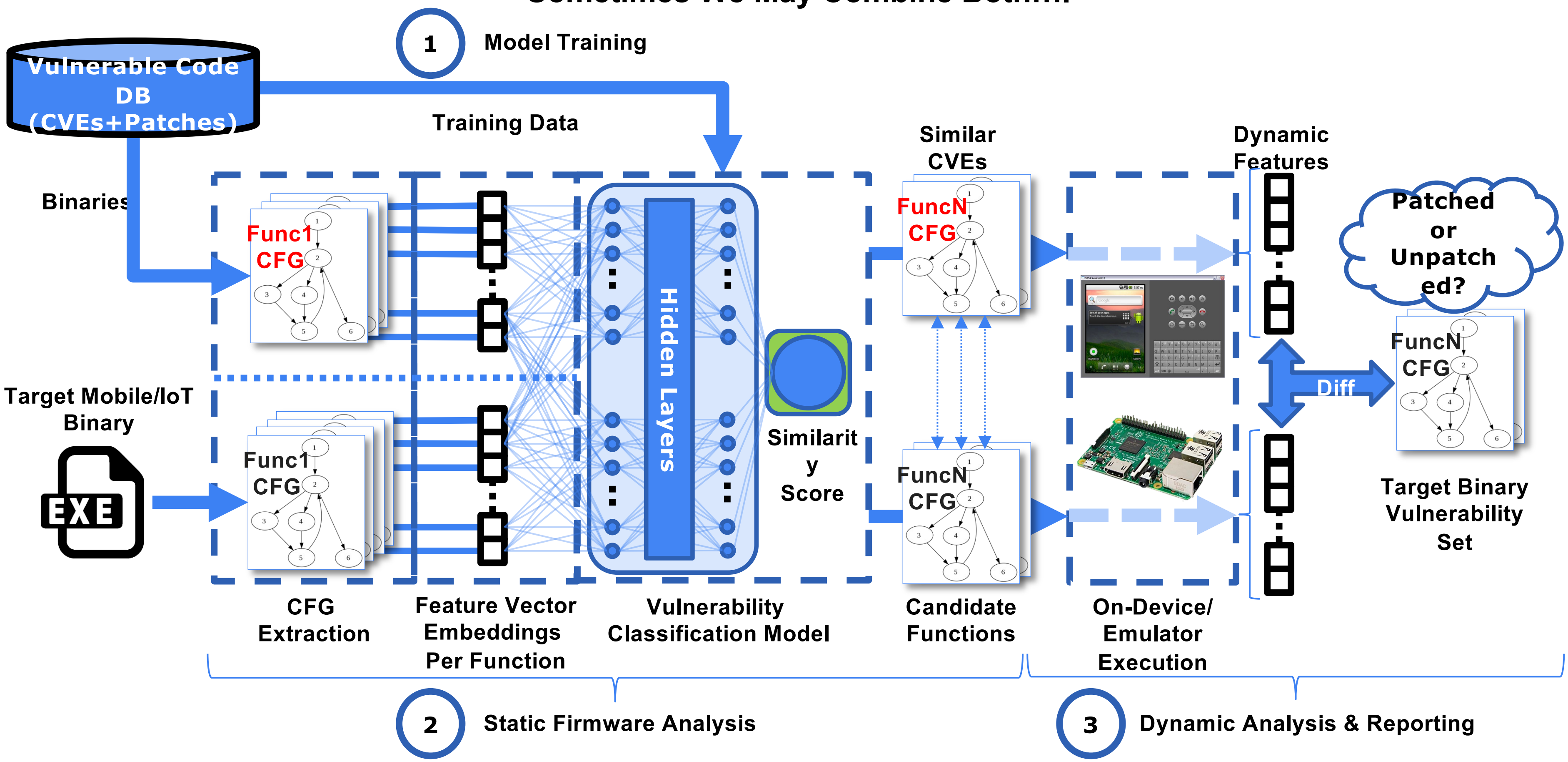
Analyze code ***without*** executing it...

<u>Dynamic Analysis</u>

Analyze program by executing it...

...sometimes with specific inputs

Sometimes We May Combine Both….



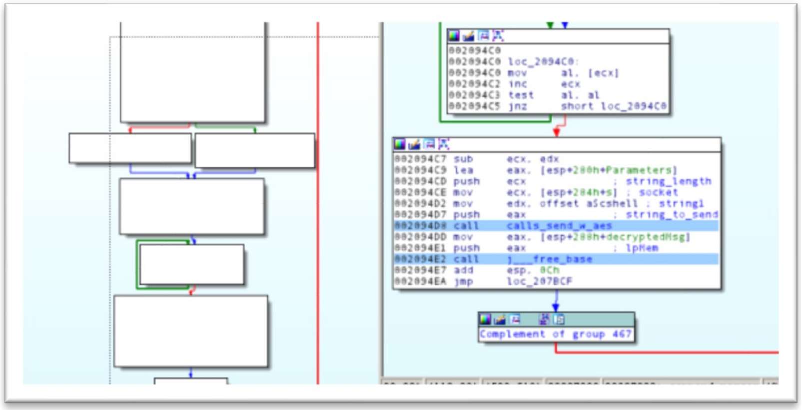
https://par.nsf.gov/servlets/purl/10211514

Common Challenges for Both in CPS/IoT

- **Diverse Hardware and Software Platforms**
  - Difficult to scale
- **Sometimes we may not have access to source code/ground truth**
  - Analyzing 3rd party binaries
  - Lose semantic meaning of different pieces of code
- **Modeling Interactions with Physical World**
  - Very difficult to capture complexity of real-world noise
- **State-space Explosion**
  - Mostly relevant for static techniques, but dynamic techniques require modeling execution environment as well
  - Number of software and physical states can be seemingly infinite for CPS/IoT applications

Static Program Analysis

- Examining code, bytecode, or binary code without execution
- **Pros:**
  - Early detection of bugs
  - **Scalable** for analyzing lots of codebases across platforms (you don't have to execute each one)
  - Comprehensiveness: analyze all code paths
- **Cons:**
  - False positives: lots of bugs may be reported
  - Doesn't scale with code complexity
  - Environment dependencies: e.g., peripheral communication
  - Doesn't include physical interactions



IDA Pro

Common Static Program Analysis Techniques

- **Control Flow Analysis**
  - Analyze the order in which different parts of the program are executed
- **Data Flow Analysis**
  - Analyzes the flow of data through program variables
  - Taint tracking can be used to analyze the flow of ”tainted” data through the program
    - Can also be dynamic!
- **Symbolic Execution**
  - ”Execute” programs symbolically to explore execution paths
  - Can identify conditions under which certain paths are taken
  - Can also be dynamic!

Overlap between techniques!

Dynamic Program Analysis

- Examining code by executing the program
- **Pros:**
  - Capture dynamic behaviors of programs to generate real traces
  - Can do black-box testing (useful when no source-code)
  - Detect runtime errors
  - Can simulate environmental interactions (network, CPS simulators, etc.)
  - Reduce false positives (e.g., only analyze bugs associated with main scan cycle of CPS)
- **Cons:**
  - Incomplete code coverage
  - Incomplete behavioral analysis
  - Requires executing the environment with high fidelity (e.g., emulating processor, simulating real-world interactions, etc.)
    - Difficult to scale to large code bases!
  - Non-deterministic behavior (pertinent to black-box testing)

Common Dynamic Program Analysis Techniques

- **Fuzz Testing**
  - Providing random or semi-random inputs to a program to see if it crashes or behaves unexpectedly
  - Useful for unanticipated scenarios
- **Runtime Verification**
  - Monitor execution of a system to check if it conforms to certain properties or specficiations
    - **More on this in the coming modules!**
- **Taint Analysis**
  - Track the flow of data through a program's execution
    - **More on this in the privacy modules!**
- **Symbolic Execution**
  - Mixing concrete values with symbolic representation (mixing static + dynamic analysis)

For Deeper Dive into Applied Software Security and Fuzzing

- Explore state-of-the-art techniques in discovering software security vulnerabilities

CS 5963/6963: Applied Software Security Testing

This special topics course will dive into today's state-of-the-art techniques for uncovering hidden security vulnerabilities in software. Projects will provide hands-on experience with real-world security tools like AFL++ and AddressSanitizer, culminating in a final project where **you'll team up to hunt down, analyze, and report security bugs in a real application or system of your choice**.

This class is open to graduate students and upper-level undergraduates. It is recommended you have a solid grasp over topics like software security, systems programming, and C/C++.

Professor

Stefan Nagy

https://users.cs.utah.edu/~snagy/courses/cs5963/

Static or Dynamic?

Raw W

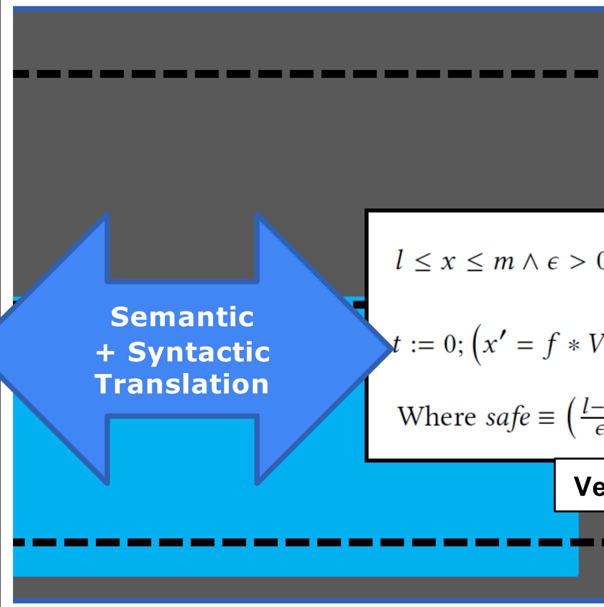
```
PROGRAM prog0
  VAR_INPUT
    f : REAL;
    x : REAL;
  END_VAR

  VAR_OUTPUT
    V : BOOL;
  END_VAR

  IF((f<((L-x)/(Tsample+Tplc)))) THEN
    V:=0; ELSE
  IF((f>((H-x)/(Tsample+Tplc)))) THEN
    V:=0; ELSE
  IF((f>=((L-x)/(Tsample+Tplc)))) THEN
    IF((f<((H-x)/(Tsample+Tplc)))) THEN
      V:=1;
    END_IF;
  END_IF;
  END_IF;
  END_IF;
END_PROGRAM

CONFIGURATION Config0
RESOURCE Res0 ON PLC
TASK Main(INTERVAL:=T#Tsamplems,PRIORITY:=0);
PROGRAM Inst0 WITH Main : prog0;
END_RESOURCE
END_CONFIGURATION
```

PLC Structured Text



Semantic + Syntactic Translation

$$l \le x \le m \wedge \epsilon > 0 \rightarrow \Big[\Big(f := *; ?safe; V := 1; \cup ?\neg safe; V := 0;$$

$$t := 0; \big(x' = f * V, t' = 1 \,\&\, x \ge 0 \wedge t \le \epsilon\big)\Big)^*\Big] l \le x \le m$$

Where $safe \equiv \left(\frac{l-x}{\epsilon} \le f \le \frac{m-x}{\epsilon}\right)$.

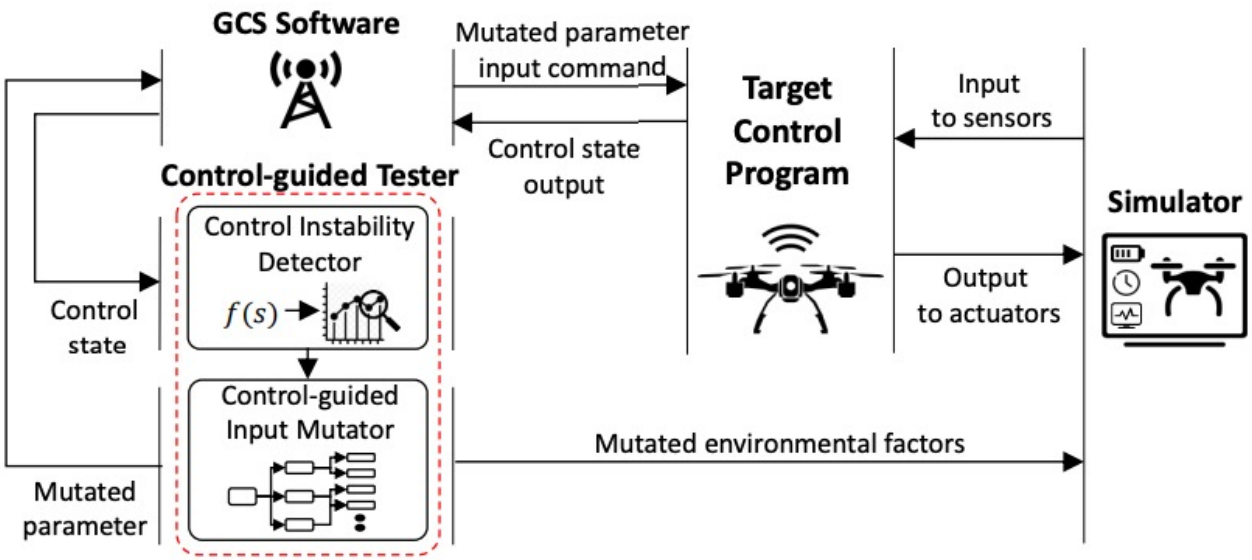
Verifiable Hybrid Program Logic

Safety: L<TankLevel<H

Static

CPS '19

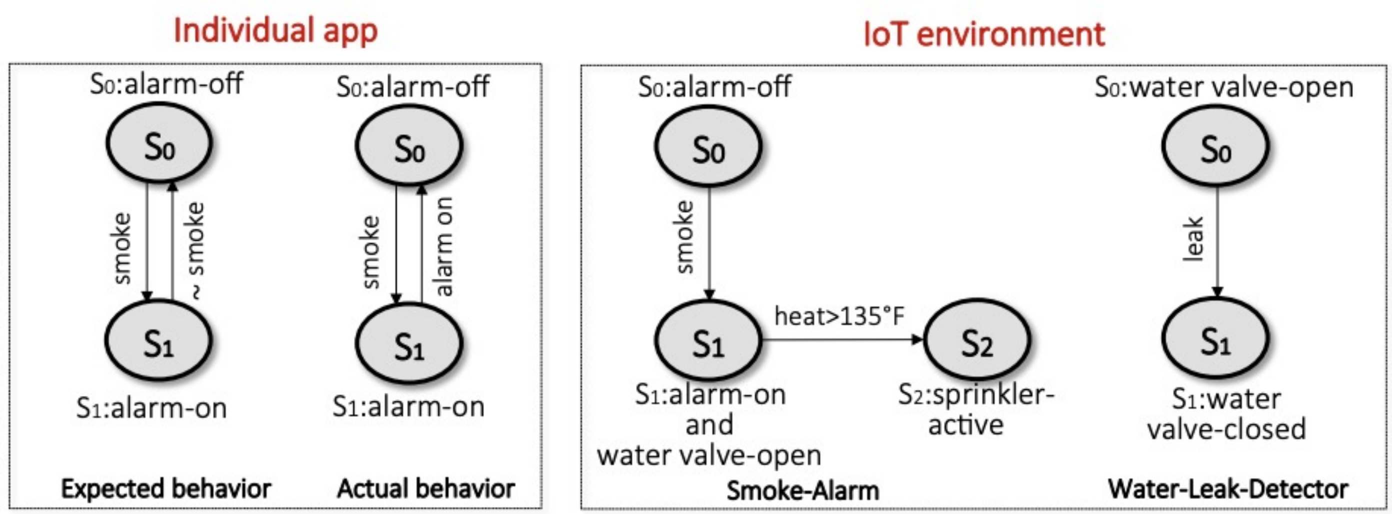
Static or Dynamic?



Dynamic

RVFuzzer, Kim et. al, USENIX Security '19

Static or Dynamic?



Does alarm sound when there is smoke?

Does the sprinkler system active when there is a fire?

Celik, et. al, USENIX ATC '18

Could be both!

Static or Dynamic?

"At every time step, for all the objects (*id*) in the frame, if the object class is car with probability > 0.7, then in the next 5 frames the object (*id*) should still be detected and classified as a car with probability > 0.6"

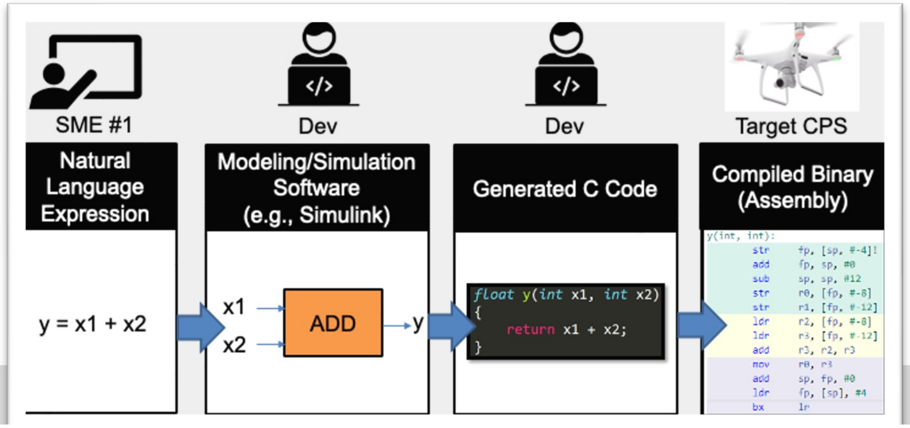
$$\phi_2 = \Box\left(\begin{array}{c} x.\forall id@x, (C(x,id) = Car \wedge P(x,id) > 0.7) \rightarrow \\ \Box y.((x \leq y \wedge y \leq x+5) \rightarrow (C(y,id) = Car \wedge P(y,id) > 0.6)) \end{array} \right)$$

Dynamic

Car in adjacent lane (Red Box) becomes undetected for 3 frames (Yellow Boxes)

CPS Semantics vs. Source Code vs. Binary Analysis

- Semantics are lost in translation
- Program analysis techniques can still be applied at all levels of abstraction
  - However, correctness can be lost in translation
  - Requires **semantic reverse engineering** at each level
  - **But how are CPS semantics encoded as bugs?**



Background: SAT

Given a propositional formula in conjunctive normal form (CNF), find if there exists an assignment to Boolean variables that makes the formula true:

literals

clauses

$\omega_1 = (b \vee c)$

$\omega_2 = (\neg a \vee \neg d)$

$\omega_3 = (\neg b \vee d)$

$\varphi = \omega_1 \wedge \omega_2 \wedge \omega_3$

$A = \{a=0, b=1, c=0, d=1\}$

SATisfying assignment!

Background: SMT (Satisfiability Modulo Theory)

- An SMT instance is a generalization of a Boolean SAT instance
- Various sets of variables are replaced by predicates from a variety of underlying theories.

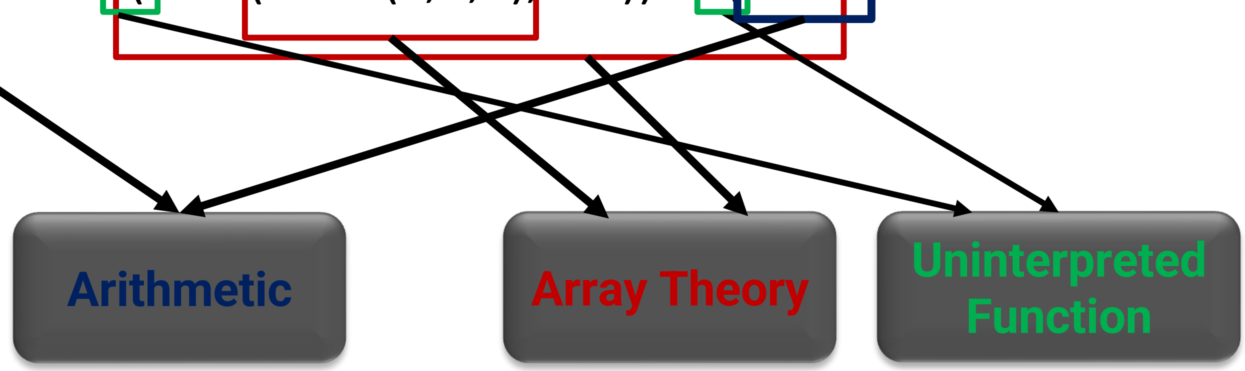
Input: a **first-order** formula φ over background theory (Arithmetic, Arrays, Bit-vectors, Algebraic Datatypes)

Output: is φ satisfiable?

- does φ have a model?
- Is there a refutation of φ = proof of $\neg\varphi$?

Background: SMT

- b + 2 = c and f(read(write(a,b,3), c-2)) ≠ f(c-b+1)



Example SMT Solving

- b + 2 = c and f(read(write(a,b,3), c-2)) ≠ f(c-b+1)

[Substituting c by b+2]

- b + 2 = c and f(read(write(a,b,3), b+2-2)) ≠ f(b+2-b+1)

[Arithmetic simplification]

- b + 2 = c and f(read(write(a,b,3), b)) ≠ f(3)

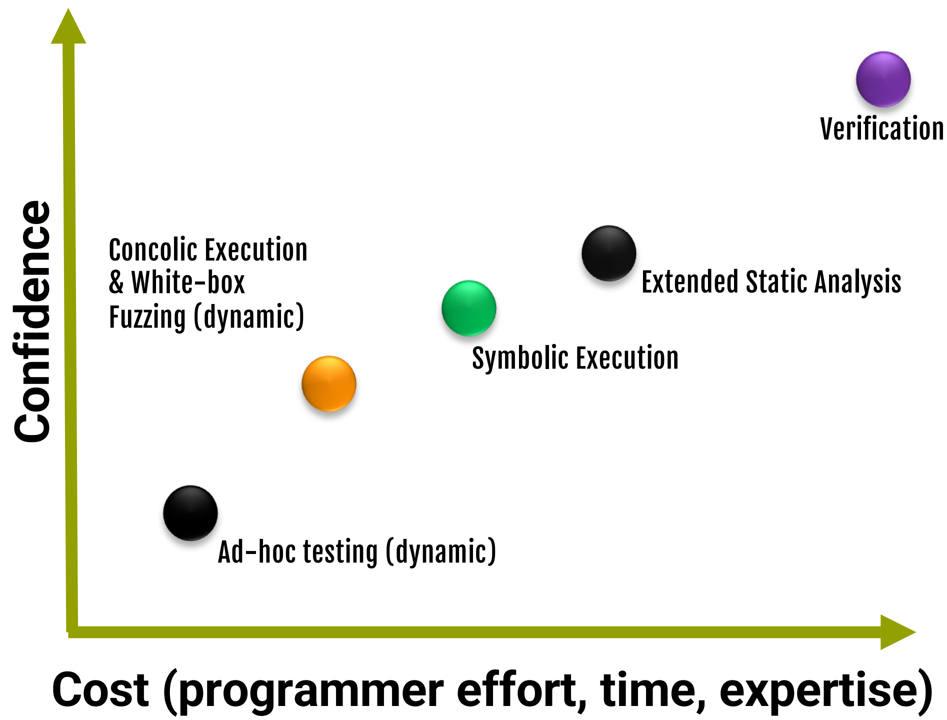
[Applying array theory axiom]

forall a,i,v:read(write(a

- b+2 = c and f(3) ≠ f

read : array × index → element
write : array × index × element → array

Program Validation Approaches



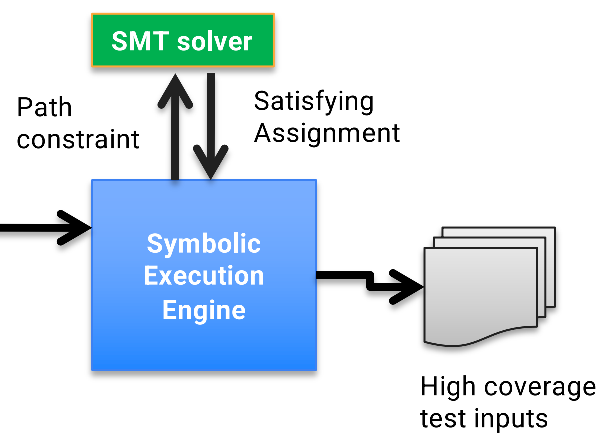
Automatic Test Generation
Symbolic & Concolic Execution

- How do we automatically generate test inputs that induce the program to go in different paths?

- **Intuition**:
  - Divide the whole possible input space of the program into equivalent classes of input.
  - For each equivalence class, all inputs in that equivalence class will induce the same program path.
  - Test one input from each equivalence class.

A Brief Intro Symbolic Execution

```
Void func(int x, int y){
    int z = 2 * y;
    if(z == x){
        if (x > y + 10)
            ERROR
    }
}
int main(){
    int x = sym_input();
    int y = sym_input();
    func(x, y);
    return 0;}
```



Symbolic Execution

Slides in collaboration with Suman Jana

Symbolic Execution

- Blurring the lines between static and dynamic analysis
- Execute program with symbolic valued inputs (**Goal: good path coverage**)
- One path constraint abstractly represents all inputs that induces the program execution to go down a specific path
- Solve the path constraint to obtain one representative input that exercises the program to go down that specific path
- **Symbolic execution implementations**: KLEE, angr, Java PathFinder, etc.

More on Symbolic Execution

- Instead of concrete state, the program maintains **symbolic states**, each of which maps variables to symbolic values

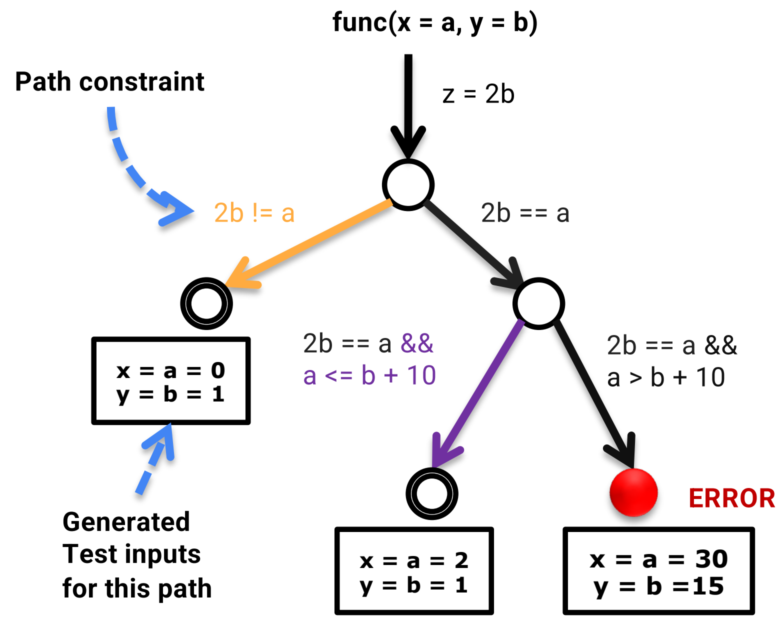
- **Path condition** is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
  - We'll discuss "quantifier-free" formulas later

- All paths in the program form its **execution tree**, in which some paths are feasible and some are infeasible

Symbolic Execution

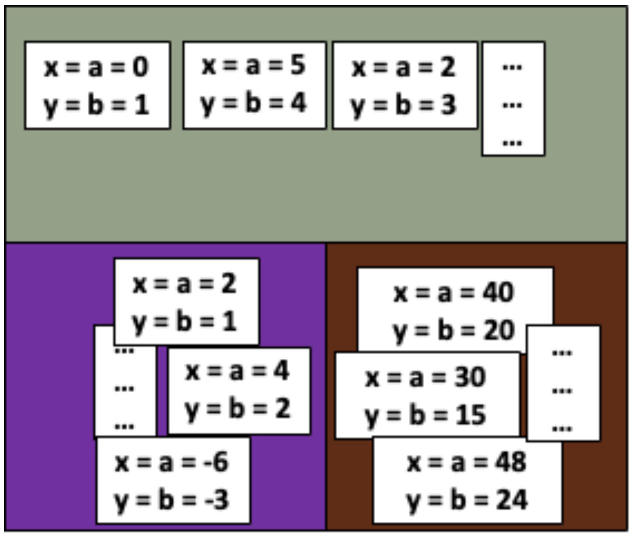
```
Void func(int x, int y){
    int z = 2 * y;
    if(z == x){
        if (x > y + 10)
            ERROR
    }
}
int main(){
    int x = sym_input();
    int y = sym_input();
    func(x, y);
    return 0;
}
```

How does symbolic execution work?



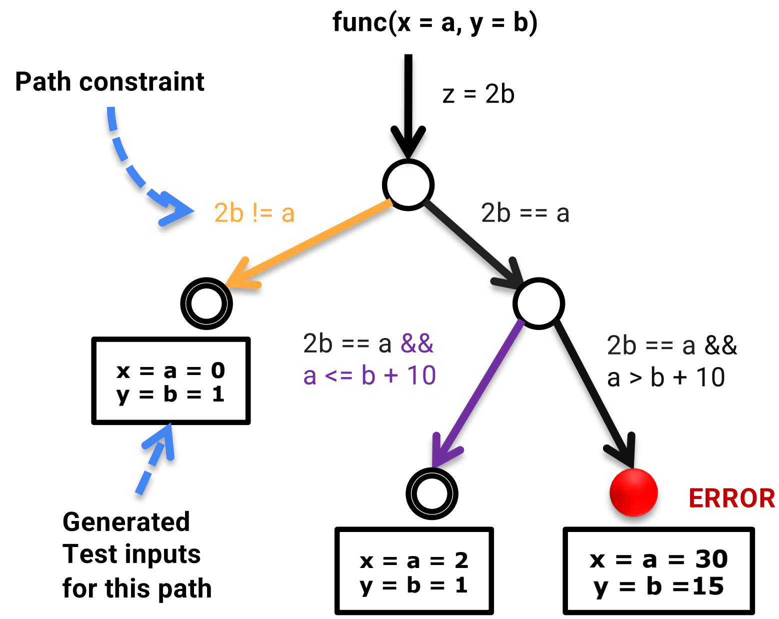
Note: Require inputs to be marked as symbolic

Symbolic Execution



Path constraints represent equivalence classes of inputs

How does symbolic execution work?

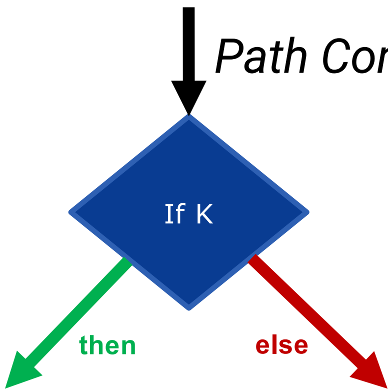


Note: Require inputs to be marked as symbolic

SMT Queries

- Counterexample queries (generate a test case)
- Branch queries (whether a branch is valid)

Path Constraints = $\{C_1, C_2, \ldots, C_n\}$; SAT



<u>Use queries to determine validity of a branch</u>

else path is impossible: $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge \neg K$ is UNSAT

then path is impossible: $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge K$ is UNSAT

How does Symbolic Execution Find bugs?

- It is possible to extend symbolic execution to help us catch bugs
- **How**: Dedicated checkers
  - **Divide by zero example** --- y = x / z where x and z are symbolic variables and assume current PC is ***f***
  - Even though we only fork in branches we will now fork in the division operator
  - One branch in which z = 0 and another where z !=0
  - We will get two paths with the following constraints:

    $\boxed{z = 0 \;\&\&\; f}$, $\boxed{z \mathrel{!=} 0 \;\&\&\; f}$
  - Solving the constraint $\boxed{z = 0 \;\&\&\; f}$ will give us concrete input values that will trigger the divide by zero error.

How does Symbolic Execution Find bugs?

- It is possible to extend symbolic execution to he… …gs
- **How**: Dedicated checkers
  - **Divide by zero example** --- y = x / z where x … …es and assume current PC is ***f***
  - Even though we only fork in bran… …e division operator
  - One branch in which z = 0 an…
  - We will get two paths w… …nts:

z = 0 && ***f***,   z != 0 &&…

  - Solving th… …ve us concrete input values that will trigger the divide …

Write a dedicated checker for each kind of bug (e.g., buffer overflow, integer overflow, integer underflow)

Classic Symbolic Execution --- Practical Issues

- **Loops and recursions** --- infinite execution tree
- **Path explosion** --- exponentially many paths
- **Heap modeling** --- symbolic data structures and pointers
- **SMT solver limitations** --- dealing with complex path constraints
- **Environment modeling** --- dealing with native/system/library calls/file operations/network events
- **Coverage Problem** --- may not reach deep into the execution tree, specially when encountering loops.

Solution: Concolic Execution

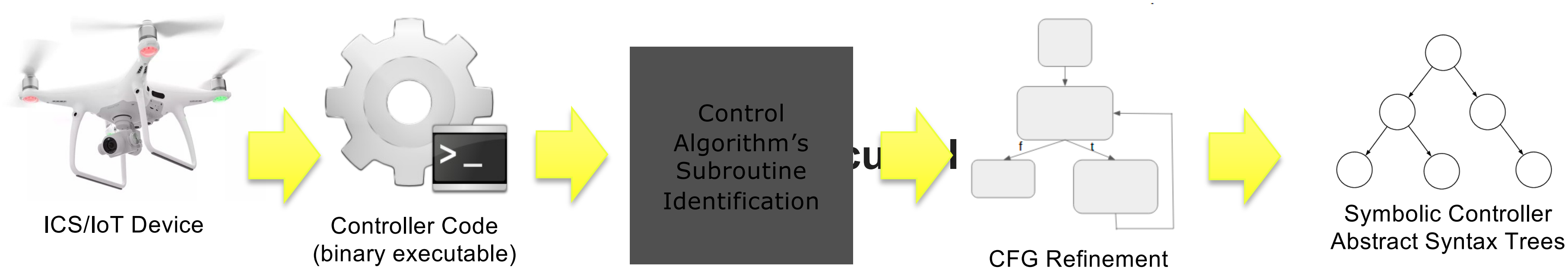
Concolic = **Con**crete + Symb**olic**

Combining Classical Testing with Automatic Program Analysis

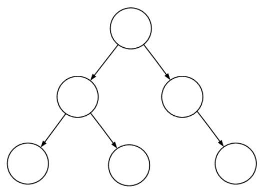
Also called **dynamic symbolic execution**

- The intention is to visit deep into the program execution tree
- Program is simultaneously executed with concrete and symbolic inputs
- Start off the execution with a random input
- Specially useful in cases of remote procedure call
- **Concolic execution implementations**: SAGE (Microsoft), CREST

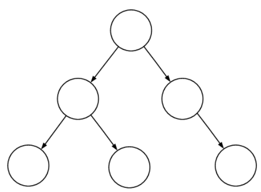
Use Case:
Symbolic Execution for Recovering CPS Semantics



Use Case:
Symbolic Execution for Recovering CPS Semantics

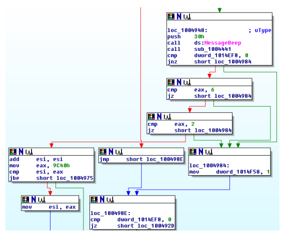


Symbolic Controller
Abstract Syntax Trees



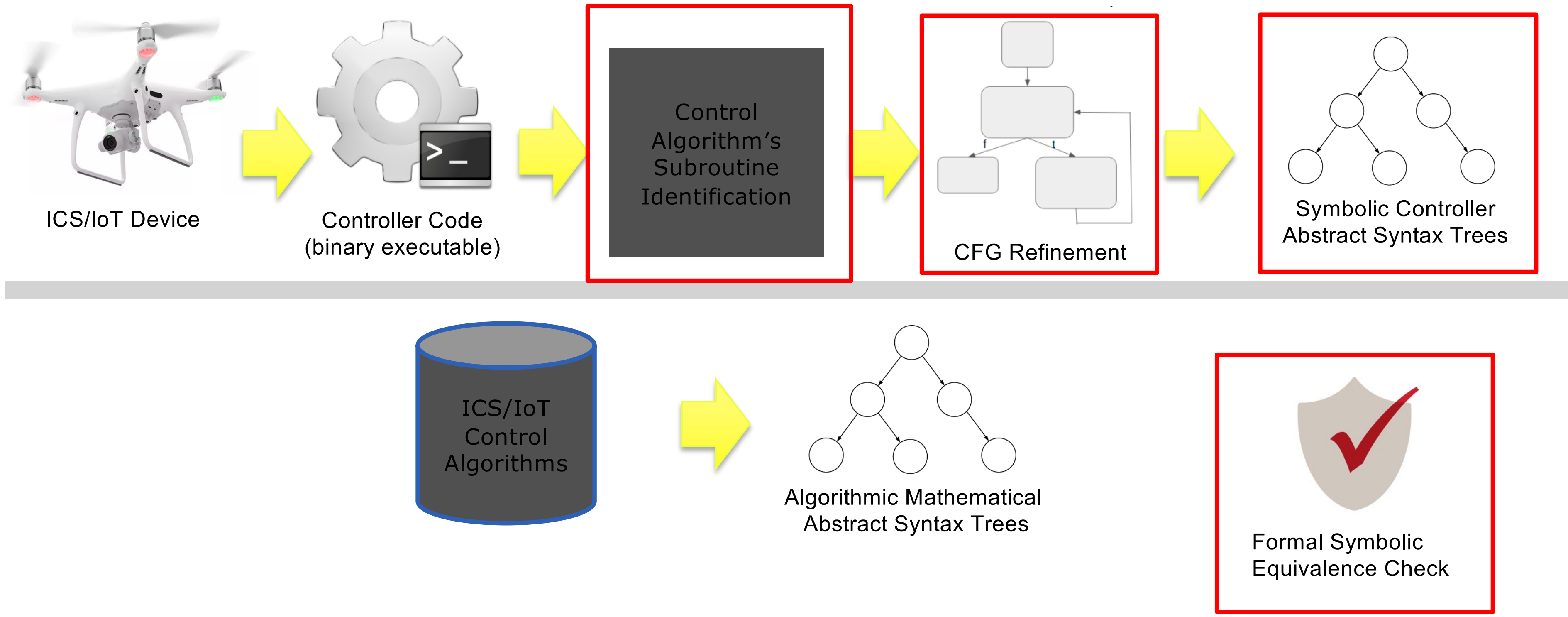
Algorithmic Mathematical
Abstract Syntax Trees

Formal Symbolic
Equivalence Check



Annotated
Disassembly

Use Case:
Symbolic Execution for Recovering CPS Semantics



Recap: Program Analysis Techniques for CPS/IoT

- Lots of static and dynamic program analysis techniques can be applied to CPS/IoT to automate testing

- Bugs can be encoded as constraints to your program analysis tool
  - **But how do we encode semantics of cyber-physical interactions?**
    - **Next module!**

- Next class: Starting Module 3: Formal Modeling and Verification
  - We'll bring back some program analysis techniques as needed

Questions?