A11L.UAS.95_A58: Illustrate the Need for UAS Cybersecurity Oversight & Risk Management

Appendix B: Task 3 Scenario Summaries and Lessons Learned

January 2, 2025

Modeling UAS w/ Chase

Garrett Mills <glm@ku.edu>

The UAS

- Linux controller:
    - UxAS - flight plan & decisions
    - ArduPilot - directs flight hardware
    - Stat - reports state to UxAS/Ground
- Flight hardware:
    - Sensors - current state of UAS
    - Hardware - physical flight controls
- Ground Station/Network

Model consists of:

- Architecture of UAS
- Rules for corruption
- Initial points of corruption

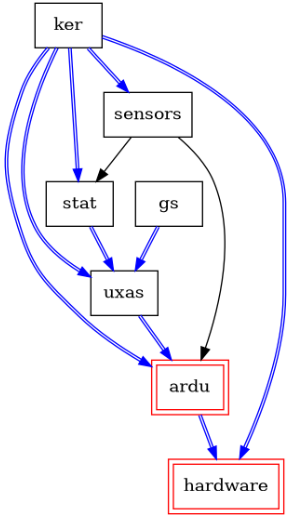
```
% Architecture of UAS
controls(network, uxas).
controls(uxas, ardu).
controls(ardu, hardware).

informs(sensors, ardu).
informs(stat, uxas).

% Rules for corruption
cor(C1) & informs(C1, C2) => misinformed(C2).
cor(C1) & controls(C1, C2) => puppet(C2).

% Initial points of corruption
cor(network).
```

```
% Architecture of UAS
controls(network, uxas).
controls(uxas, ardu).
controls(ardu, hardware).

informs(sensors, ardu).
informs(stat, uxas).

% Rules for corruption
cor(C1) & informs(C1, C2) => misinformed(C2).
cor(C1) & controls(C1, C2) => puppet(C2).

% Initial points of corruption
cor(network).
```

```
% Architecture of UAS
controls(network, uxas).
controls(uxas, ardu).
controls(ardu, hardware).

informs(sensors, ardu).
informs(stat, uxas).

% Rules for corruption
cor(C1) & informs(C1, C2) => misinformed(C2).
cor(C1) & controls(C1, C2) => puppet(C2).

% Initial points of corruption
cor(network).
```

```
% Architecture of UAS
controls(network, uxas).
controls(uxas, ardu).
controls(ardu, hardware).

informs(sensors, ardu).
informs(stat, uxas).

% Rules for corruption
cor(C1) & informs(C1, C2) => misinformed(C2).
cor(C1) & controls(C1, C2) => puppet(C2).

% Initial points of corruption
cor(network).
```

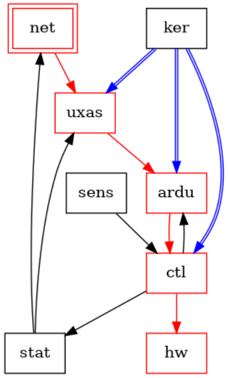
Ex.1: Simple Corruption



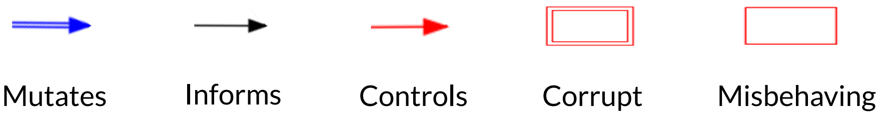
- ArduPilot is corrupt
- A component "mutates" another when it can entirely compromise it
- A component "informs" another when it provides read-only information

Mutates Informs Corrupt

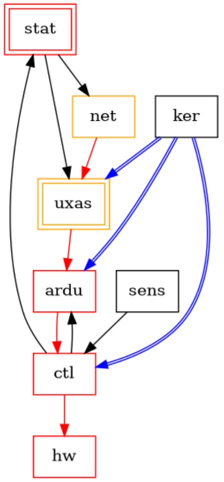
Ex.2: Control- & Data-Flow Separation



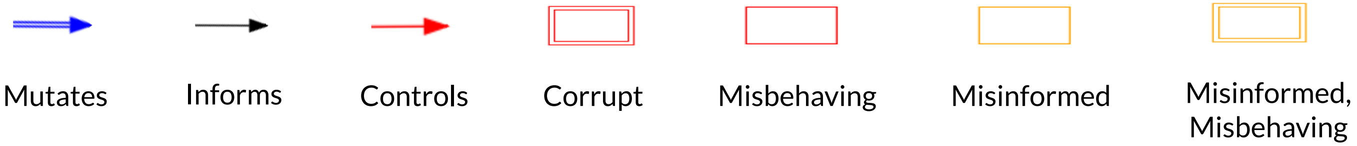
- Network is corrupt
- Introduced embedded controller
- "Control" lets a corrupt component direct the actions of another without corrupting it
  - Distinct from "mutates"



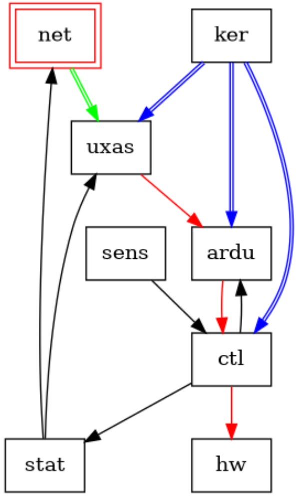
Ex.3: Misinformation & Misbehavior



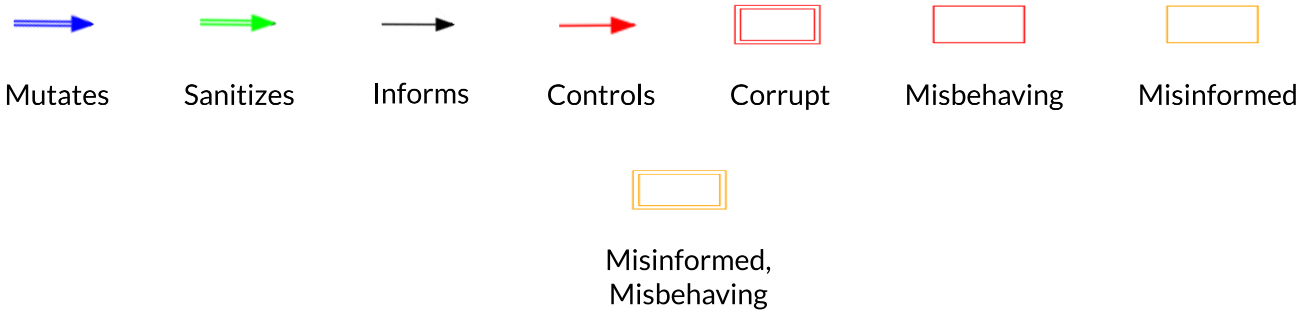
- Stat reporter is corrupt
- Bad information travels along data flows
- If component A is controlled by a misinformed component B, then A will misbehave.
- If A is both misinformed *and* misbehaving, then it is fully compromised.



Ex.4: Sanitization



- Network is corrupt
- UxAS is protected by an authentication check
  - e.g. Ground station can send messages, but 3rd-party actor cannot



Ongoing Work

- Auto-generating models (corruption points, structures)
- Automated analysis (scoring heuristics)
- Analysis of UxAS in Coq

Analysis of UxAS in Coq

```
Inductive uxas_state :=
| UxasState
    (atom: nat)
    (messages: list uxas_message)
    (arvs: arvs_state)
    (task: task_state)
    (ras_agg: ras_aggregator_state)
    (ras_col: ras_collector_state)
    (atbbs: atbbs_state)
    (pbs: pbs_state)
.
```

- Documentation defines core task pipeline as collection of transition systems
- Model in Coq and compose into a single system

THE UNIVERSITY OF KANSAS

Institute for Information Sciences

Adam Petz, Garrett Mills, Perry Alexander

11-17-22

A58 Monthly TIM: November, 2022 (UAS Static Analysis updates)

Outline

1. Overview of CHASE model finder
2. Overview of UxAS + architecture
3. Initial findings: Modeling UxAS architecture + attacks
4. Initial findings: Modeling UxAS message sequences

CHASE model finder (Overview)

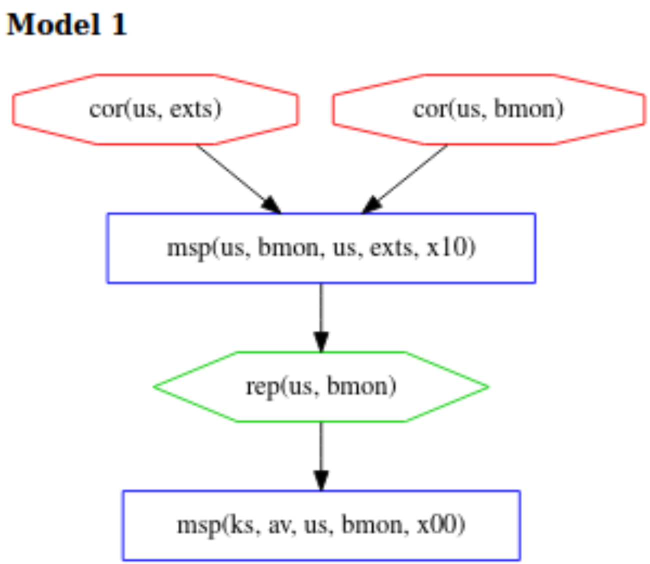
- CHASE[1]
  - Model finder for first-order logic with equality
  - Open source: https://github.com/ramsdell/chase
- Model specifications
  - Written in *Finitary Geometric Form*
  - A_1 & A_2 & … & A_m => C_1 | C_2 | … | C_n .
  - Each A_i (Antecedent – Left of "=>"): Atomic Formula
  - Each C_j (Consequent – Right of "=>"): Conjunction of Atomic Formulas ($B_{j,1}$ & $B_{j,2}$ & … & $B_{j,p}$)
- Custom Predicates
  - $P(c_1, c_2, \ldots, c_n)$
  - $f(c_1, c_2, \ldots, c_m) = c_0$
  - **Example:**

```
author(X) & paper(Y) & assigned(X, Y).
author(X) & paper(Y) => read_score(X, Y) | conflict(X, Y).
assigned(X, Y) & author(X) & paper(Y) => read_score(X, Y).
assigned(X, Y) & conflict(X, Y) => false.
```

[1] Ramsdell, J. D. ***Chase: A model finder for finitary geometric logic***. https://github.com/ramsdell/chase, 2020.

CHASE model finder (Example)

```
[ bound = 500, limit = 5000, input_order ]

% Assume adversary avoids detection at our main measurement
% event. Others can be added.
l(V) = msp(us, M, us, exts, X)
 => corrupt_at(us, exts, V).

% Assumptions about system dependencies.
depends(ks, C, ks, av) => false.
depends(us, C, us, bmon) => false.
depends(us, C, us, exts) => false.

% Axioms defining "deep" components
% We don't want to see models with deep corruptions
l(V) = cor(ks, M) => false.

% Axiom defining which components cannot be recently corrupted
prec(V, V1) & l(V1) = cor(P,C) & ms_evt(V)
 => false.

m4_include(`ex1b.gli')m4_dnl

m4_include(`ex1b_dist.gli')m4_dnl

m4_include(`thy.gli')m4_dnl
```



[2] C. Parran et. al, **Trust Analysis of Copland Phrases (Tutorial)**, copland-lang.org, 2022.

CHASE model finder (Example)

Corruption events

Repair events

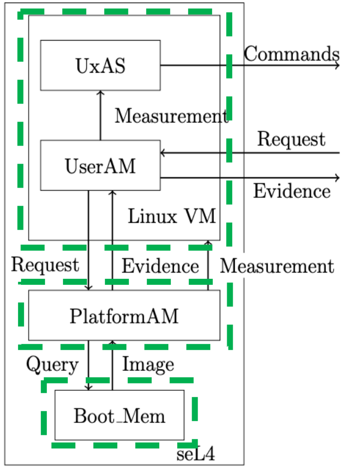


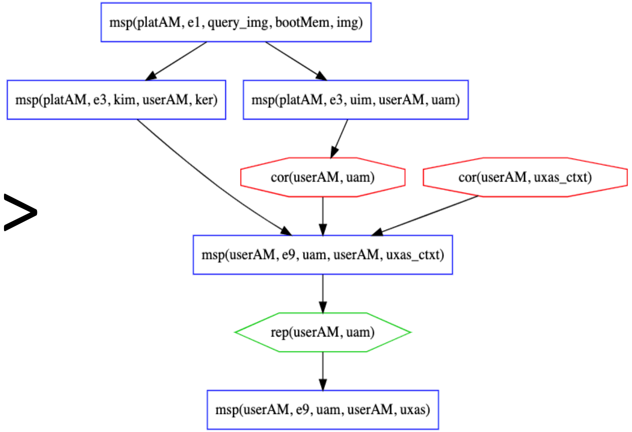
Figure 6.2. Hardened Ground Station Architecture.

==>

```
% platAM components only corrupted via a corrupt boot image
l(E) = cor(platAM, C) => phi(bootMem, img, E).

%% img in bootMem cannot be corrupted
phi(bootMem, img, E) => false.

% user AM (uam) only corrupted via a corrupt kernel
l(E) = cor(userAM, uam) => phi(userAM, ker, E).
```

```
% In addition to ker, uxas depends also on uxas_ctxt
ctxt(userAM, C, uxas) => C = ker | C = uxas_ctxt.

% Ignore attacks that corrupt ker
l(E) = cor(userAM, ker) => false.
```

==>

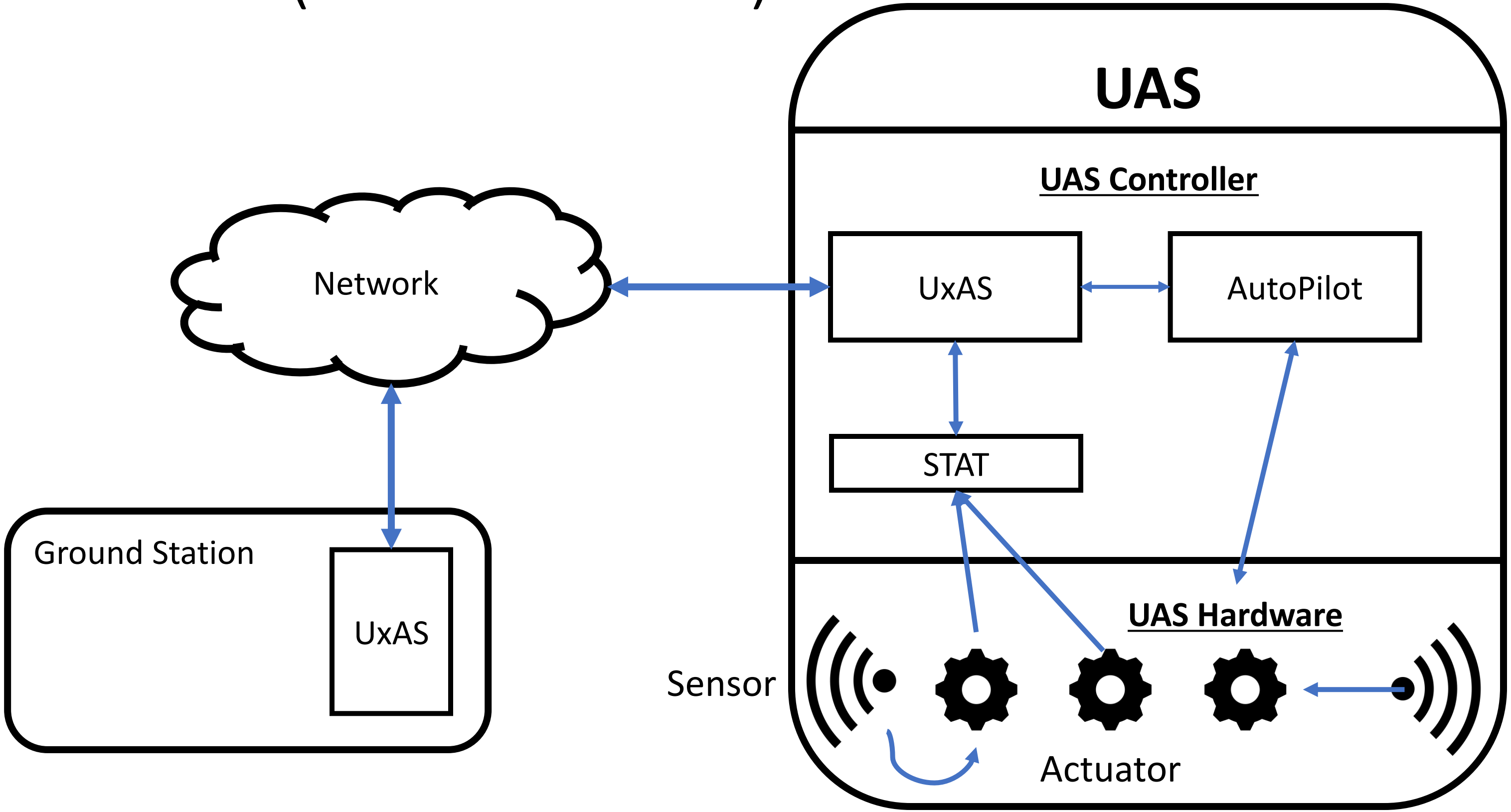


[3] Petz, A., G. Jurgensen, and P. Alexander, ***Design and Formal Verification of a Copland-based Attestation Protocol***, ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'21), Virtual, Nov 20-22, 2021.

UxAS (Overview)

- OpenUxAS[4]
  - "Software architecture … to enable autonomous capabilities on-board unmanned systems"
  - Open source: https://github.com/afrl-rq/OpenUxAS
  - Developed under AFRL's ICE-T program
- Core software
  - Implemented in C++
  - LMCP (Lightweight Message Construction Protocol): Message structure + serialization
  - ZeroMQ: Data bus for publish/subscribe message passing between services
- Applications
  - Collaboration algorithms (i.e. route planning) on-board UAVs
  - Core functionality of Unmanned Ground Sensors (UGS)

UxAS (Architecture)



References

- [1] Ramsdell, J. D., ***Chase: A model finder for finitary geometric logic***. https://github.com/ramsdell/chase, 2020.
- [2] Parran, C., I. Kretz, Ramsdell, J., and P. Rowe, **Trust Analysis of Copland Phrases (Tutorial)**, copland-lang.org, 2022.
- [3] Petz, A., G. Jurgensen, and P. Alexander, ***Design and Formal Verification of a Copland-based Attestation Protocol***, ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'21), Virtual, Nov 20-22, 2021.
- [4] UxAS Developers, **UxAS User's Manual**, https://github.com/afrl-rq/OpenUxAS/tree/develop/doc/reference/UserManual, 2022.

Practical Software Defense for GPS Spoofing on a Hobby UAV

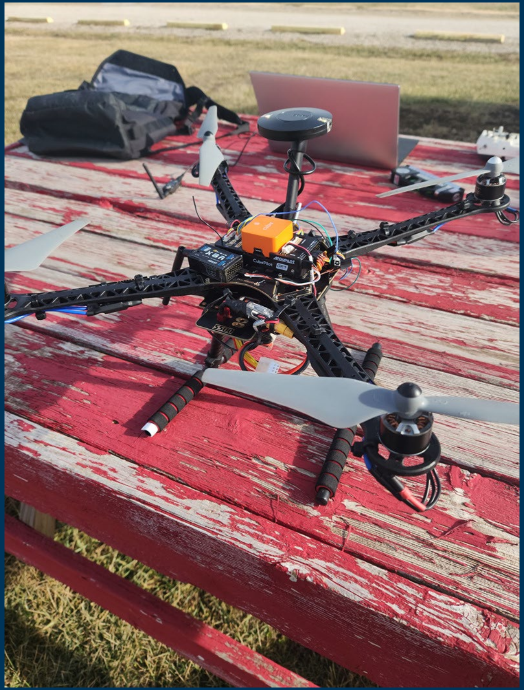
Bailey Srimoungchanh
The University of Kansas

J. Garrett Morris
The University of Iowa

Drew Davidson
The University of Kansas

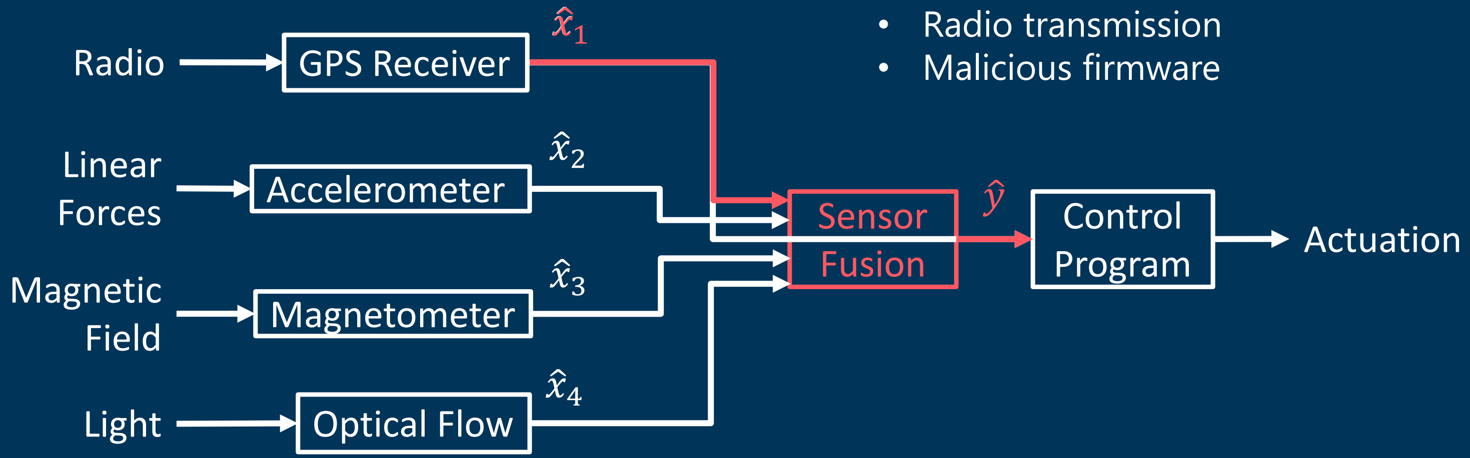
This Research

- GPS spoofing detection
- No need for pre-trained models
- Detects even subtle deviations
- Low false positive rate
- Fast time to detect attack



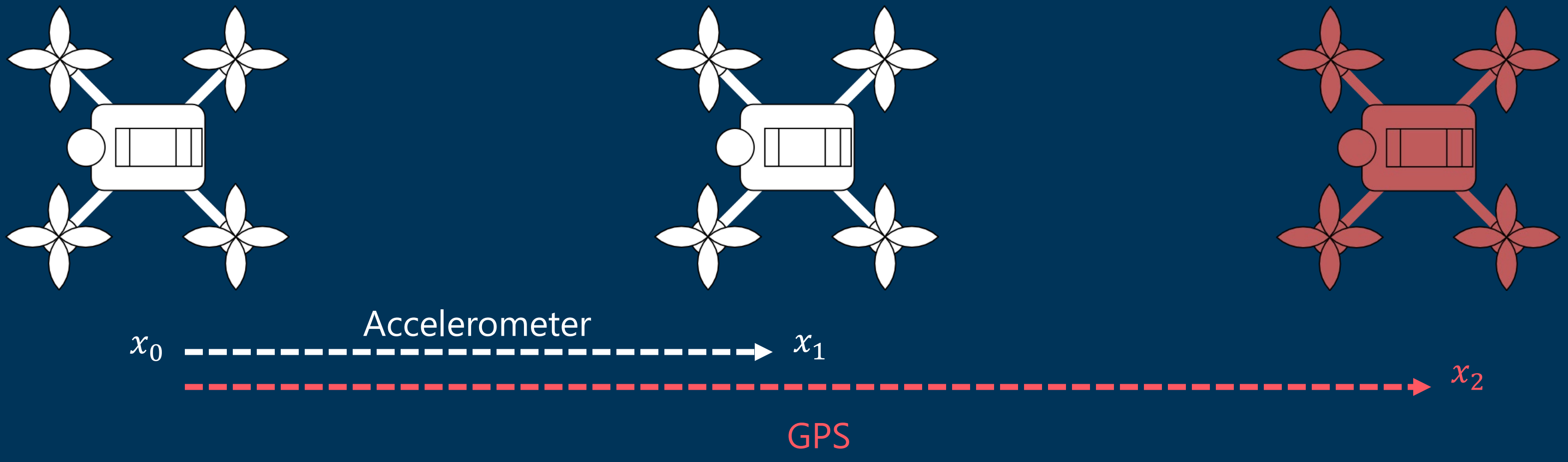
Sensor Spoofing

- Goal
  - Implicitly control the behavior of a system and cause it to behave irregularly
  - Appreciable effect on the behavior of the system
- Capabilities
  - Knowledge of the system components and software
    - Subvert predictive models
  - Complete control and knowledge of GPS receiver
    - Radio transmission
    - Malicious firmware



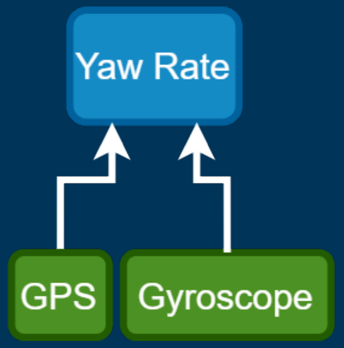
Key Insight

- Observations by the GPS need to confirm with observations by other sensors



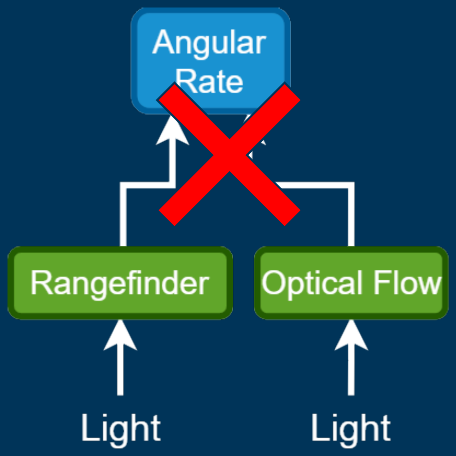
Defense Implementation

- Detect when 2 sensors are no longer confirming within some margin of error
- Modified ArduPilot
- Evaluated on Quadcopter

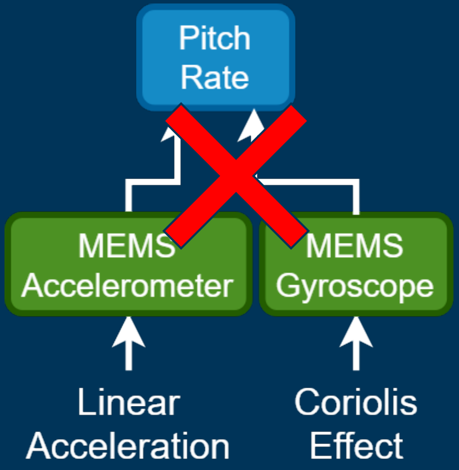


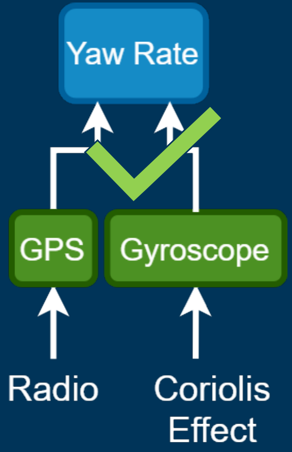
Challenge 1: Orthogonal Sensors

Requirement 1:
Measure different phenomena



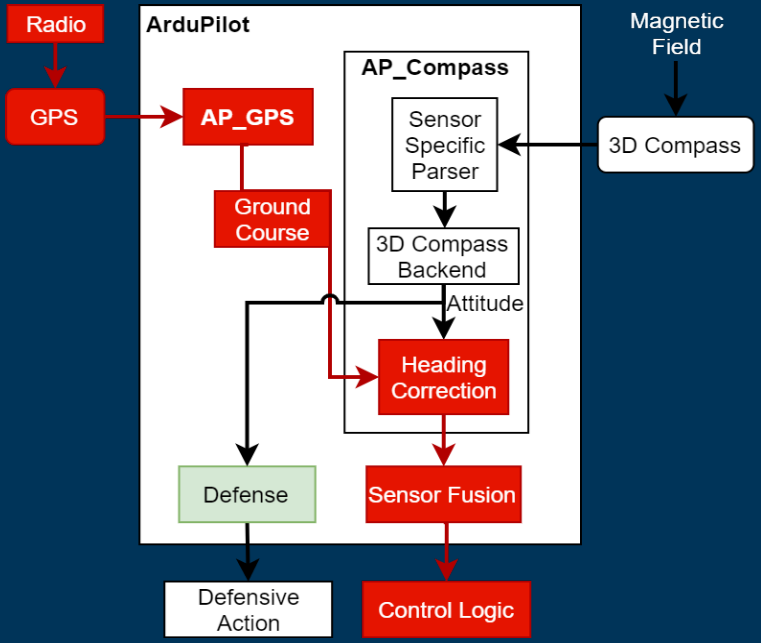
Requirement 2:
Have different physical attack surfaces





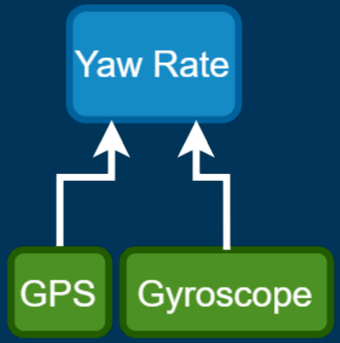
Challenge 2: Disentangle Sensors

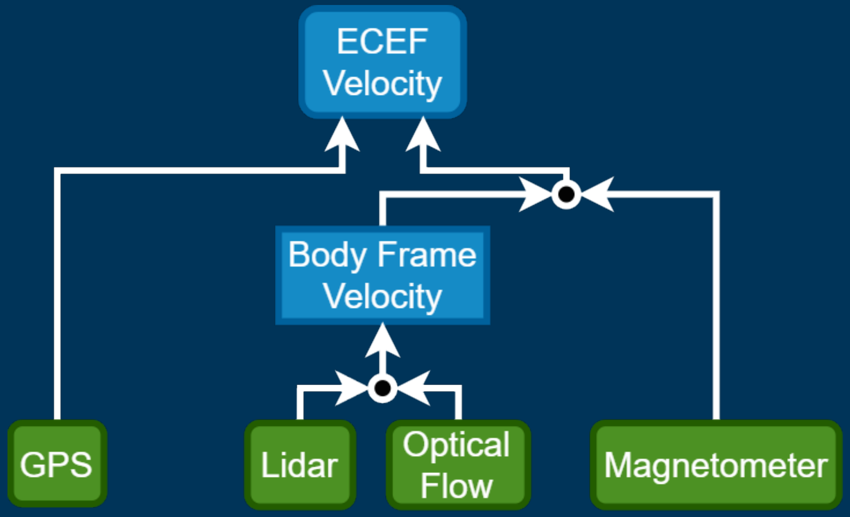
- Gyroscope
  - Measure angular rate
- Optical Flow
  - Derive velocity
- Entanglement
  - Optical Flow rotates into GPS frame
  - Uses rotation matrix from Compass
  - Rotation matrix influenced by GPS



Challenge 3: Operating Limitations

- Yaw rate from GPS
  - Too slow
- Altitude from rangefinder
  - Too high
- Body rate from optical flow
  - Too dark



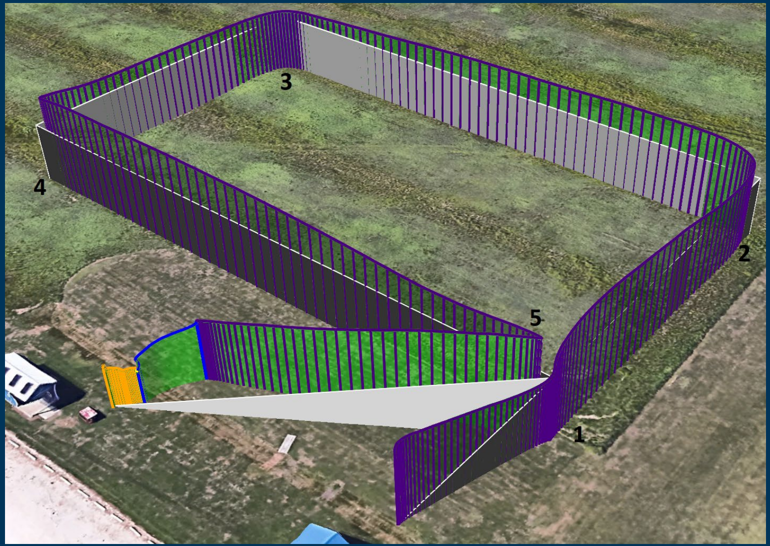


Evaluation Questions

1. Does this technique have a low false positive rate?
2. Does this technique detect attacks within our threat model?
3. Does the technique address a credible attack undetected by current approaches?

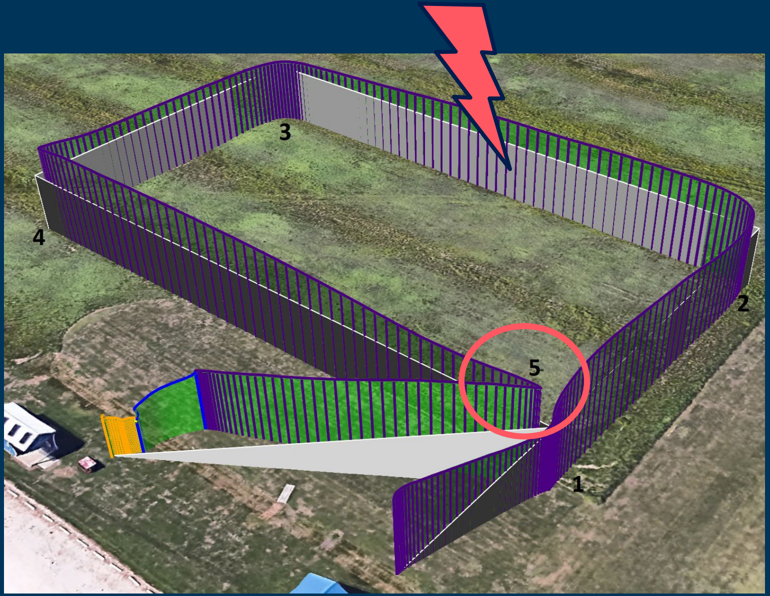
Benign Flights

- 100mx200m rectangle
- Maintain altitude of 10m
- Maintain speed of 10m/s
- 5 total flights
- Goal
  - Collect sensor data



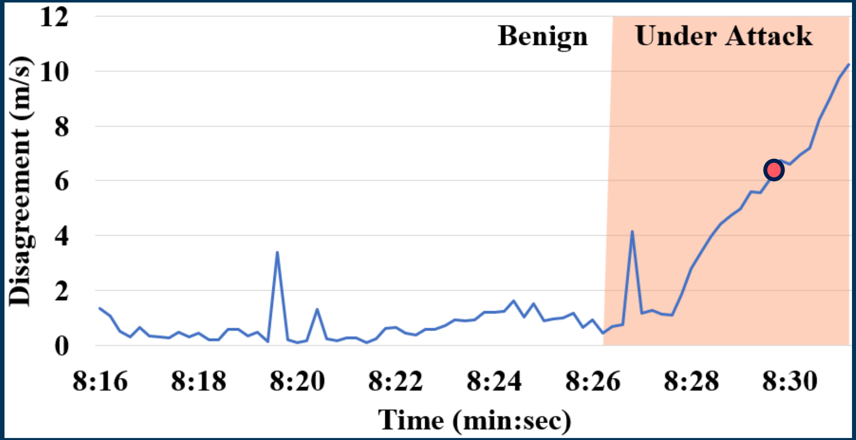
Adversarial Flights

- F-Subtle (4)
  - Same mission as Benign Flights
  - Spoofed 2.5m at a rate of $0.1 m/s^2$ from real location
- L-Overt
  - Spoofed 2m from real location in a single timestep and held there
- L-Subtle
  - Spoofed 2m at a rate of $1 m/s^2$ from real location and held there



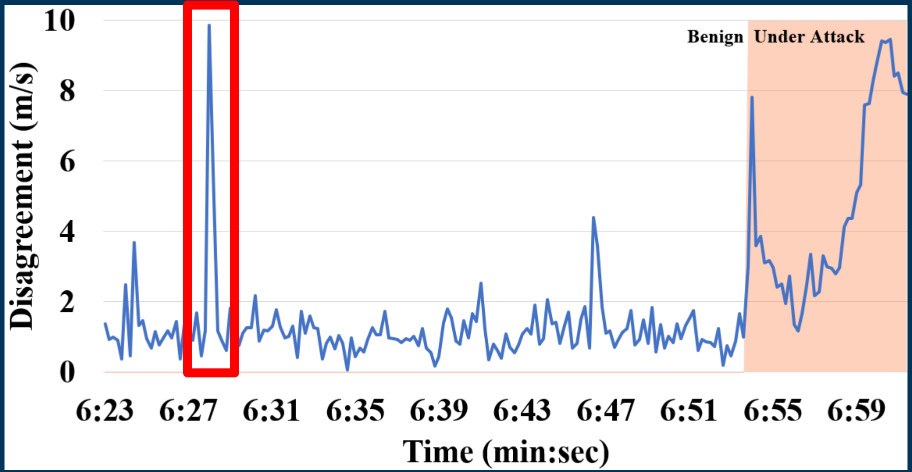
Optical Flow Performance

- Average TTD of 2.04s
- Average Displacement of 7.81m
- False Alarm in 1 flight



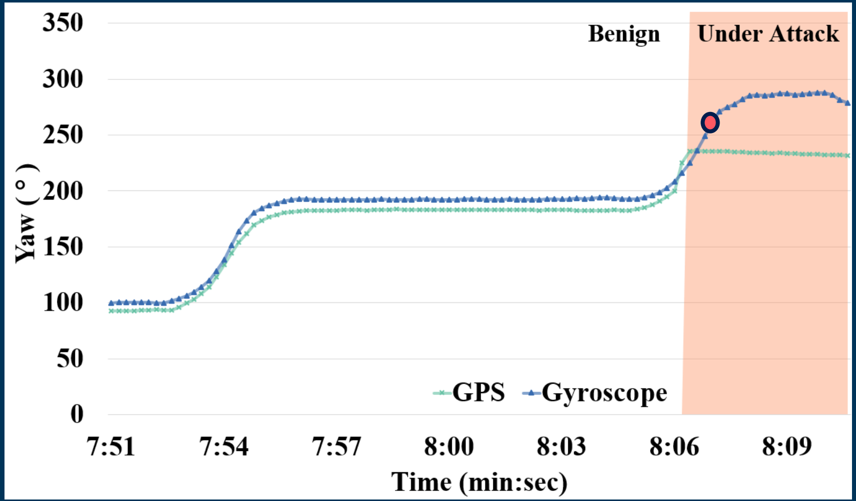
False Alarm

- Single timestep
- Instability due to noise or environmental conditions
- Can be smoothed with filtering at the cost of delaying detection



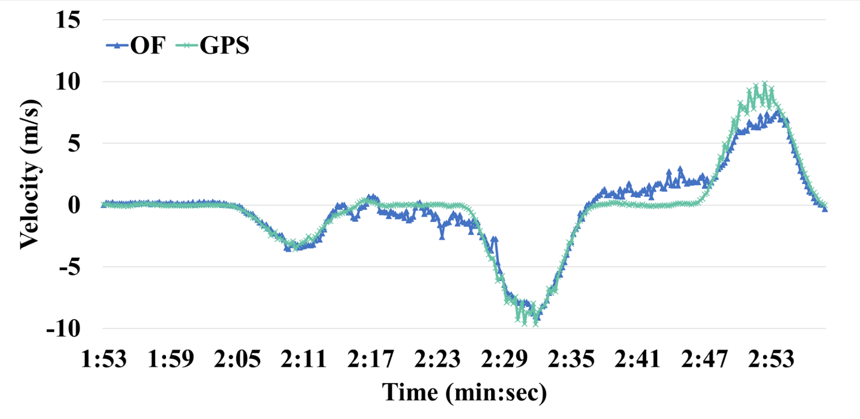
Gyroscope Performance

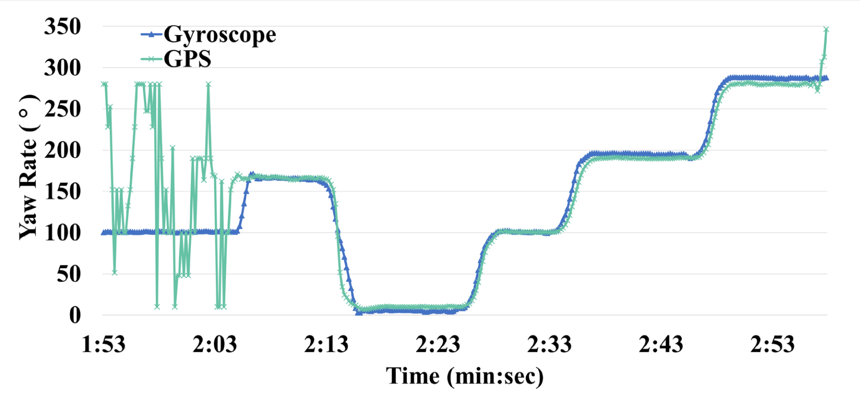
- No Loiter data
  - Limitation of GPS
- Average TTD of 1.66s
- Average Displacement of 7.61m



Composable Defense

- Average TTD of 1.29s
- Average Displacement of 3.64m



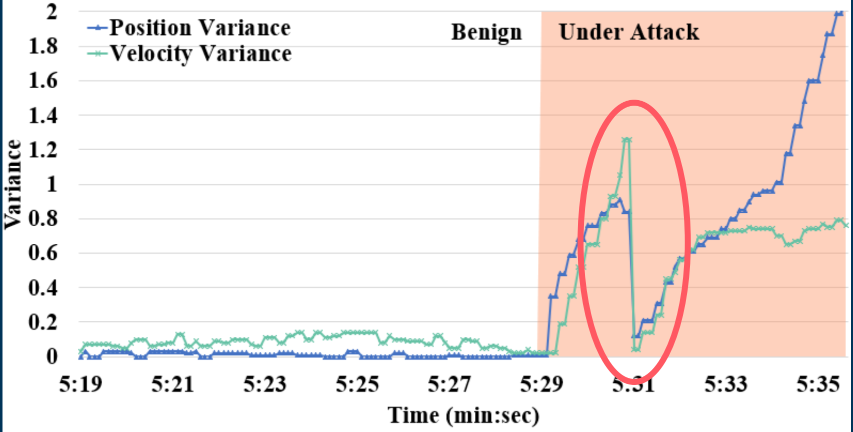


Comparison to Existing Defenses

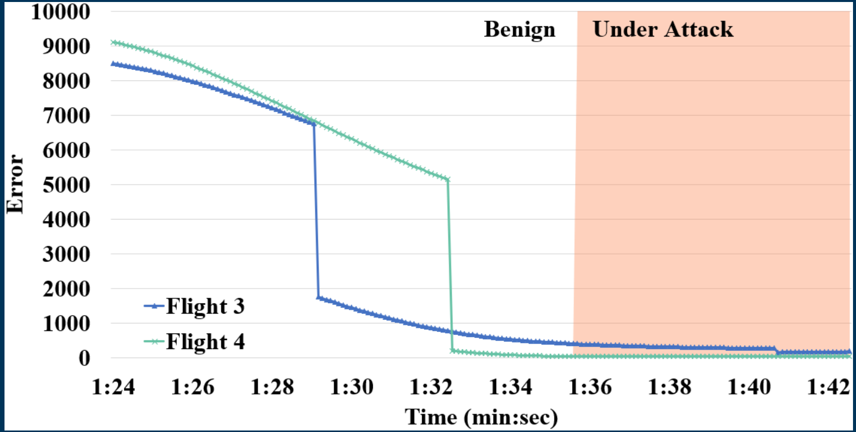
Are orthogonal sensors necessary in the context of other defenses?

Are they sufficient?

ArduCopter Health Checks
F-Subtle 1



System Identification
F-Subtle 3 and F-Subtle 4



Future Work

- Develop a formal notion of good sensor candidates and build tooling that can automatically identify entanglement
  - Identification of sensor pairs and discovering entanglement was a manual process
- Generalize our approach to more than just GPS spoofing detection

Conclusion

- We show how orthogonal sensors are effective and can overcome the limitations of sensor fusion
- Implement a novel defense that detects GPS spoofing with either a Gyroscope or an Optical Flow sensor
- Evaluate our defense with live flight tests
  – 0.001% False Positives
  – 1.29s Average Time-To-Detection

Thanks for listening!

Data and Implementation files can be found in the OSF Repository:
https://osf.io/qj97w/?view_only=721a3b784e004465a0f8bbd548da09c6

QR Code to the repository:

Acknowledgements

Jayhawk Model Masters for providing us a safe testing site

Flight Research Lab at The University of Kansas for data collection

Questions?

Presentation on Ardupilot SITL

Name: Sadia Afrin Ananna

Ph.D. Student in Electrical and Computer Engineering,

Drexel University

Supervised By: Dr. Steven Weber

What is SITL?

- SITL(Software In The Loop) is a build of the autopilot code using C++ compiler.
- SITL simulator allows us to run plane, copter or rover without any hardware.
- ArduPilot is a portable autopilot that can run on a very wide variety of platforms. Our PC is just another platform that Ardupilot can be built and run on.
- SITL takes the advantage of the fact and so it allows us to run ArduPilot on our PC directly without any special hardware.

ArduPilot

- ArduPilot is an open source, unmanned vehicle AutoPilot Software Suite. It enables the creation and use of trusted, autonomous, unmanned vehicle systems.
- Since ArduPilot is an open-source project, it is constantly evolving based on rapid feedback from a large number of users.
- Being coupled with ground control software, unmanned vehicles running ArduPilot can have advanced functionality
- ArduPilot has a wide range of vehicle simulators built in. Also, it can interface to several external simulators.

ArduPilot(Contd.)

- Although ArduPilot does not manufacture any hardware, ArduPilot firmware works on a wide variety of different hardware to control unmanned vehicles of all types.

  i. Copter
  ii. Plane
  iii. Fixed-wing aircrafts
  iv. Rover
  v. Multi-rotor drones
  vi. Submarines
  vii. Antenna trackers

Fig.1: Different type of unmanned vehicles that ArduPilot firmware can work on.

ArduPilot Hardware and Firmware

- Hardware: It is the peripheral sensors, controllers and output devices that acts as the vehicle's eyes, ears, brain and arms. It runs on a variety of hardware platforms such as Navio2, Pixhawk, Parrot Bebop etc.

- Firmware: It is the code running on the controller. The firmware can be chosen to match the vehicle and mission: Copter, Plane, Rover, Sub, or Antenna Tracker.

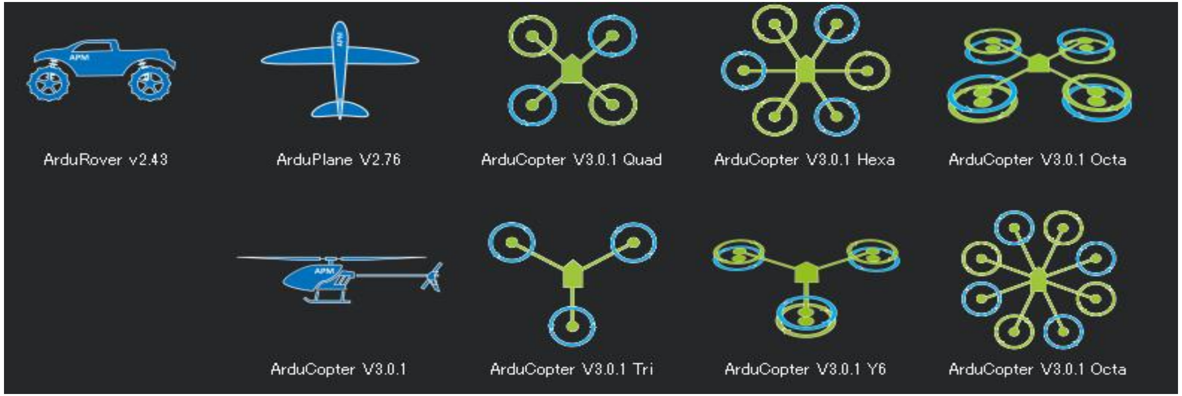


Fig.2: Different type of autopilots.

Ground Control Station

- Software: It is the interface to the controller. Also called a Ground Control Station (GCS), the software can run on PC’s or mobile devices.
- Ground Control Station runs on a ground-based computer, that communicates with the UAV via wireless telemetry.
- It displays real-time data on the UAVs performance and position and can serve as a “virtual cockpit”, showing many of the same instruments.
- A GCS can also be used to control a UAV in flight, uploading new mission commands and setting parameters.
- It is often also used to monitor the live video streams from a UAV’s cameras.

Ground Control Station(Contd.)

- There are at least ten different ground control stations. On desktop, there is:
  - Mission Planner,
  - APM Planner 2,
  - MAVProxy,
  - QGroundControl and
  - UgCS

- For Tablet/Smartphone there is :
  - Tower (DroidPlanner 3),
  - MAVPilot,
  - AndroPilot and
  - SidePilot
- The decision to select a particular GCS often depends on your vehicle and preferred computing platform.

Ground Control Station(Contd.)

- Mission Planner is a full-featured GCS supported by ArduPilot. It offers point-and-click interaction with your hardware, custom scripting, and simulation.

Fig.3: Mission Planner Ground Control Station.

Background of ArduPilot

- In year 2007, Jordi Munoz and Chris Anderson wrote an Arduino program (which he called “ArduCopter”) to stabilize an RC helicopter.
- In 2009 Munoz and Anderson released Ardupilot 1.0 (flight controller software) along with a hardware board it could run on.
- The years 2011 and 2012 witnessed an explosive growth in the autopilot functionality and codebase size, thanks in large part to new participation from Andrew Tridgell and Pat Hickey. Tridge's contributions included automatic testing and simulation capabilities for Ardupilot, along with PyMavlink and Mavproxy.
- Between 2013 and 2014 ArduPilot evolved to run on a range of hardware platforms and operating system.
- In late 2014, the DroneCode was formed and in Fall 2015 again, with a swarm of 50 planes running ArduPilot simultaneously flown. Within this time period, ArduPilot's code base was significantly refactored, and the code evolution continues.

Intended Scope

- The basic goal of the software is to provide control of the vehicle. It can be done either autonomously, or via pilot input through radio control transmitter. It can also be done through ground control station.
- ArduPilot offers a wide range of features and capabilities including:
    - Autonomous flights.
    - Telemetry.
    - Sensor integration
    - GPS-base navigation
    - Customization.

ArduPilot use cases

- Aerial photogrammetry
- Aerial photography and filmmaking.
- Remote sensing
- Search and rescue
- Robotic applications
- Academic research
- Package delivery

Integration with software packages

- Ground control stations(GCS): ArduPilot can integrate with various GCS software, including Mission Planner, MAVProxy, QGroundControl etc.

- Simulation: ArduPilot can integrate with simulation software such as ArduPilot-SITL.

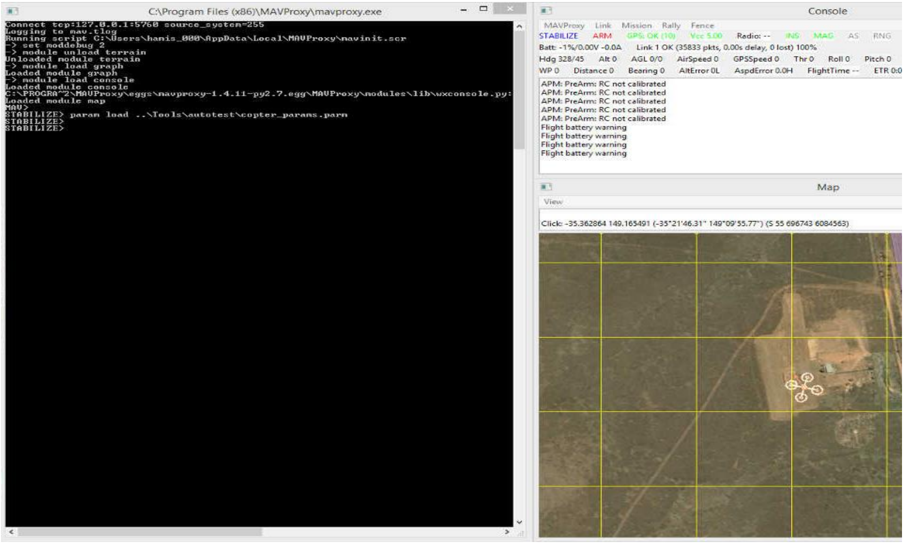


Fig. 4: MAVProxy command prompt,console and map.

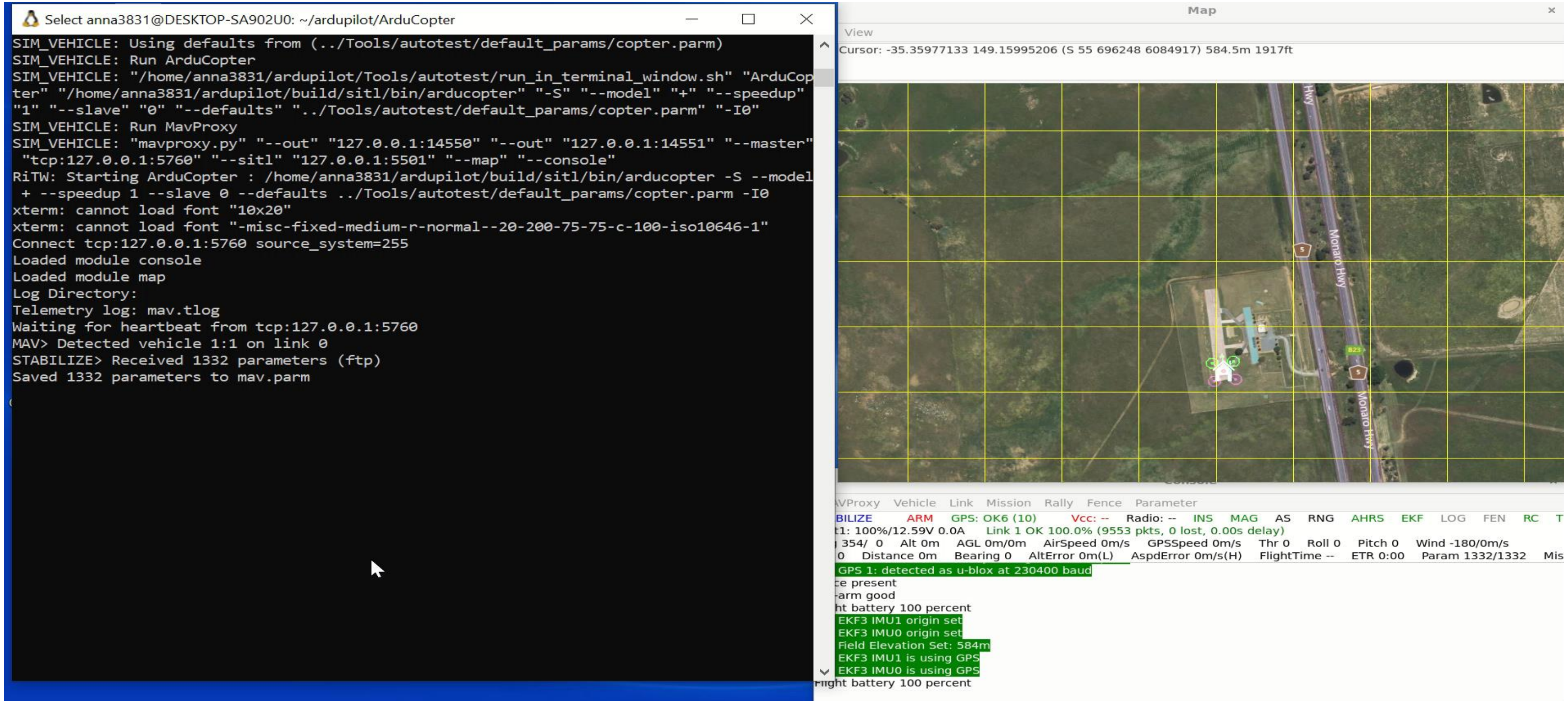
Cyber-security Threats and Counter Measures

Security Objective	Threats	Mitigations
Confidentiality	Eavesdropping	Data link encryption
	Identity spoofing	
	Hijacking	
Integreity	Man-in-the-middle	Hash Authentication MAC
	Message modification	
	Replay attack	
Availability	Jamming	Authentication
	Routing attack	
	Flooding	

Future plan

- Mission Planner Simulation allows us to see the expected behavior for vehicles in missions, or with a joystick attached, be able to fly/drive the vehicle simulation as if with RC.
- Mission Planner supports swarming or formation-flying with multiple drones or UAVs (Unmanned Aerial Vehicles).
- This concept can be useful to design and implement tests to attack UAS (Unmanned Aerial Vehicle).

Demonstration



Thanks

Logical Bugs in Drones and Swarms (2)

A survey of recent papers

Presented by Akshith for A58

Oregon State University

Papers:

Part 1 **No code! Just behavior.**

a. [SwarmFlawFinder: Discovering and Exploiting Logic Flaws of Swarm Algorithms, Jung et. al.](#)
IEEE Symposium on Security & Privacy May 2022

Part 2 **Yes code! Code analysis.**

b. [PGFuzz: Policy-Guided Fuzzing for Robotic Vehicles, Kim et. al.](#)
The Network and Distributed Systems Security Symposium, Feb 2021

c. [PGPatch: Policy-Guided Logic Bug Patching for Robotic Vehicles, Kim et. al.](#)
IEEE Symposium on Security & Privacy May 2022

Motivation

- **1.8%** are memory corruption bugs
- **98.2%** of bugs are logic bugs
  - **97.3%** logic bugs lead to physical damage

Threat Model

We know what the mission and algorithm is !

No sensor spoofing, No malware in the system !

Basically looking for **design flaws** in the algorithm / software implementation.

Not memory corruption bugs

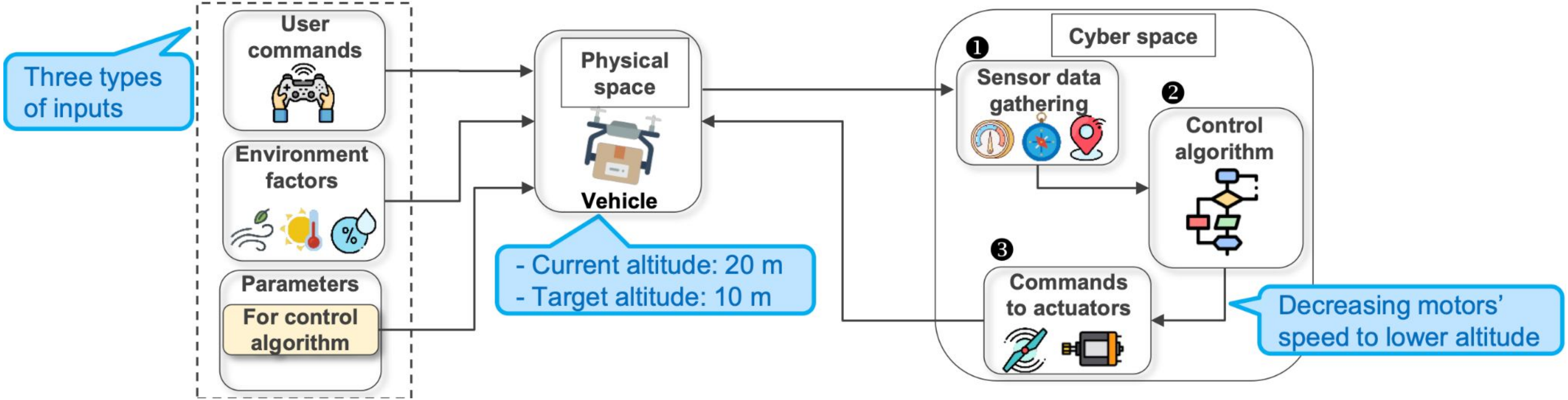
Only logical bugs

PGFuzz: Policy-Guided Fuzzing for Robotic Vehicles

Hyungsub Kim, Muslum Ozgur Ozmen, Z. Berkay Celik, Antonio Bianchi, and Dongyan Xu
Purdue University

2021 NDSS

Components of a Drone/Robotic Vehicle



Fuzzing a Drone: Traditional Fuzzers

Fail-safe mode must be triggered when the engine temperature is higher than 100 C° (212 F°)

```
// Developers forget to convert F° to C° scale
If (temperature >= 100) {
   Fail-Safe -> execute();
}
```

Fail-safe is triggered under 100 F° (37 C°).

Can traditional fuzzers (AFL, libFuzzer) discover such a design flaw? No
- Mutation: Code coverage
- Bug oracle: Memory access violation

PGFuzz

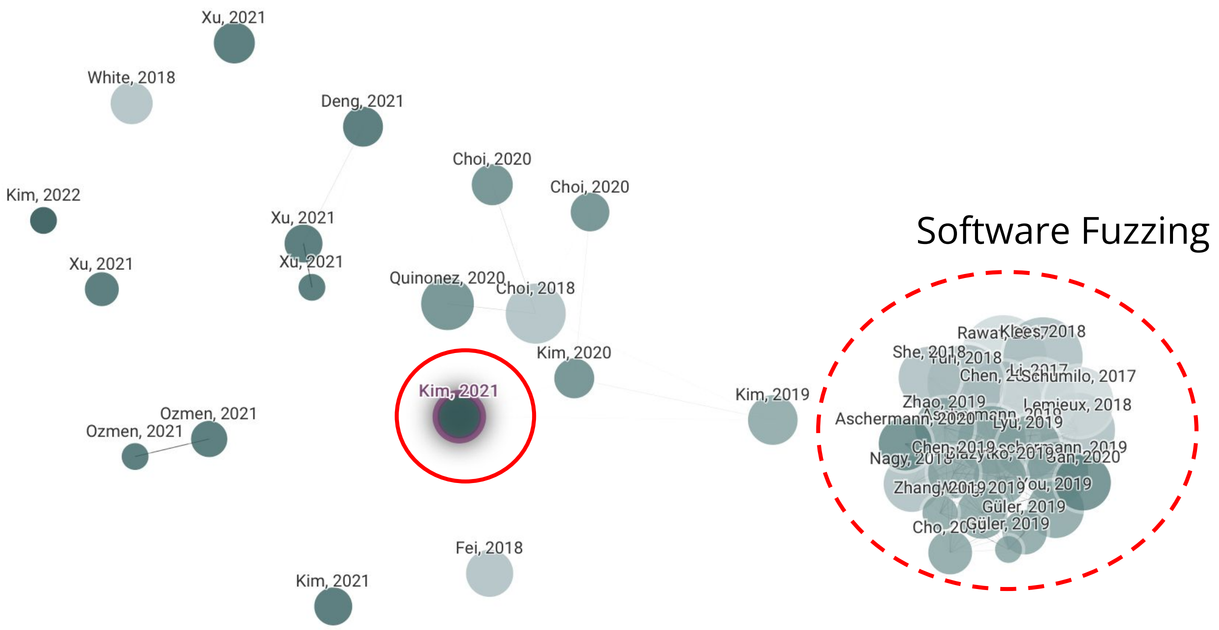


Image: Connected Papers

Fuzzing a Drone: Existing Drone Fuzzers

Can fuzzers specialized for RVs discover the design flaw? (RVFUZZER)

```
// Developers forget to convert F° to C° scale
If (temperature >= 100) {
   Fail-Safe -> execute();
}
```

Fail-safe is triggered under 100 F° (37 C°).

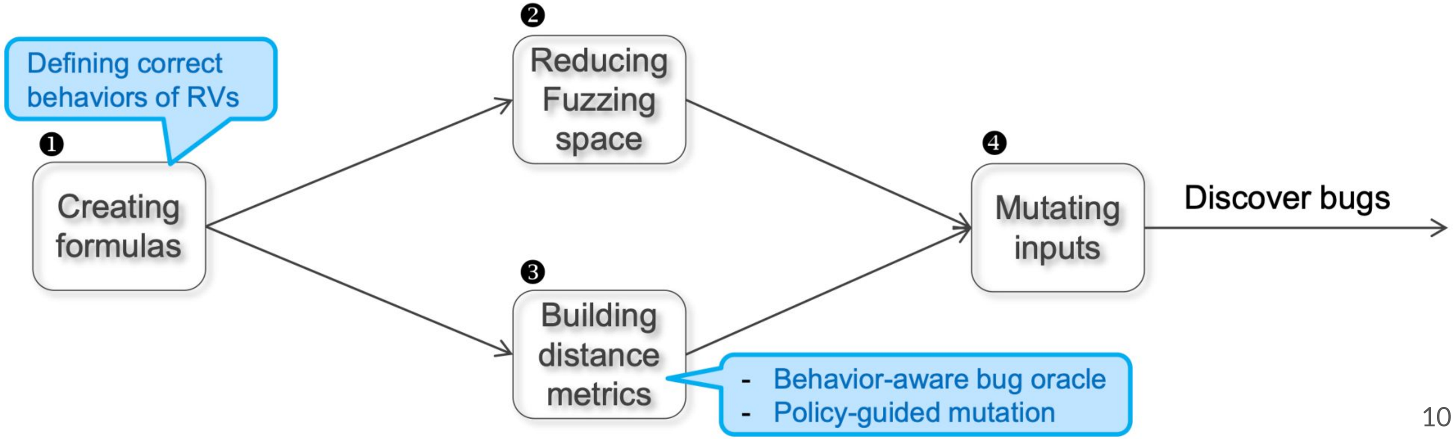
What about fuzzers for RVs? No
- Mutation & Bug oracle: unstable attitude

Fuzzing a Drone: PGFuzz

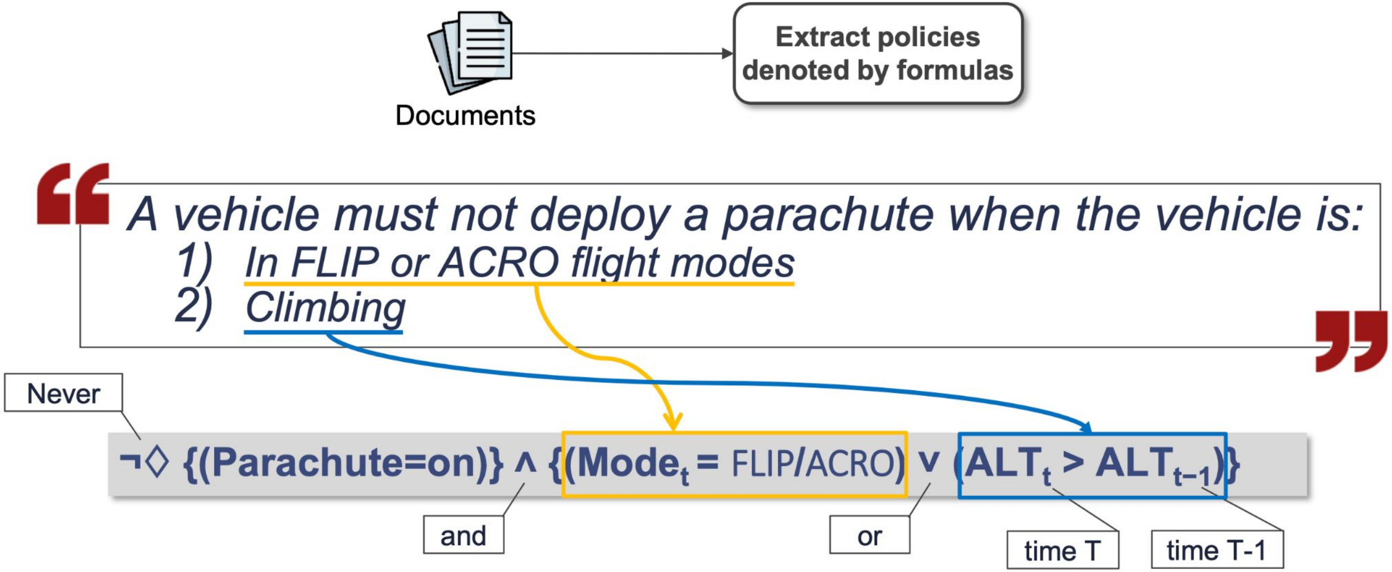
Existing methods DO NOT:

1. Know the RV's correct behaviors
2. Consider entire input space

But PGFuzz …



PGFuzz: Defining policies in formulas



PGFuzz: Finding inputs for mutation

(Reducing fuzzing space)

Huge fuzzing space

- 1,140 configuration parameters
- 58 user commands
- 168 environmental factors

Only mutating inputs relevant to the policy

PGFuzz: Finding inputs for mutation

(Reducing fuzzing space)

Policy consists of terms (physical states) - Decompose the formula into terms (states)

Mutate inputs related to the terms

Term

$$\neg\Diamond\ \{(\text{Parachute}=\text{on})\} \wedge \{(\text{Mode}_t = \text{FLIP/ACRO}) \vee \underline{(\text{ALT}_t > \text{ALT}_{t-1})}\}$$

Proposition

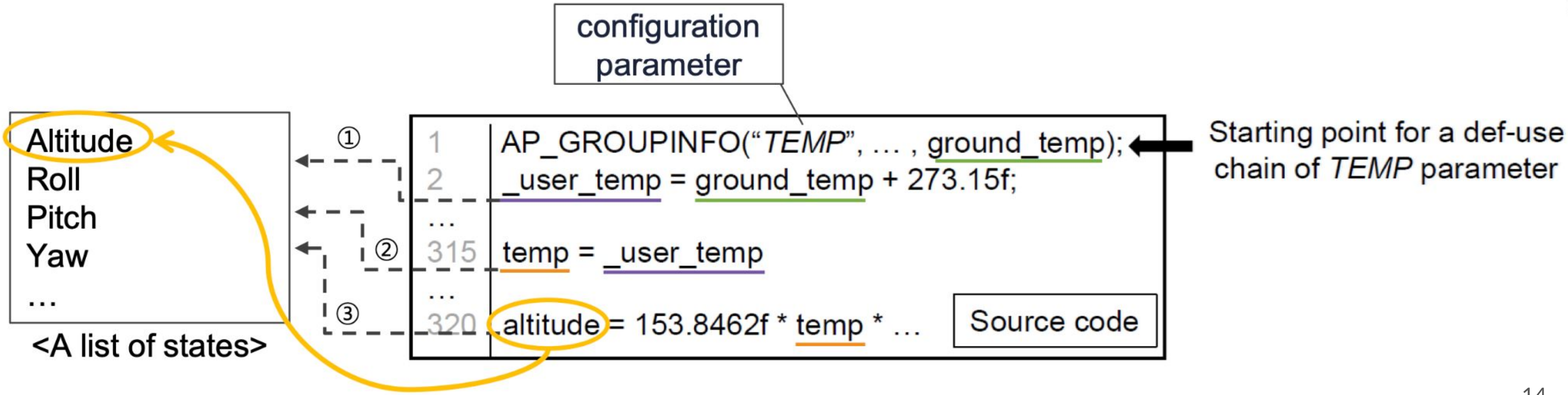
Policy	Related terms			
Parachute	Parachute	Flight mode	Altitude	…

<Policy-term map>

PGFuzz: Mapping parameters to each term

(Reducing fuzzing space)

Static analysis to identify which states are affected by each parameter.

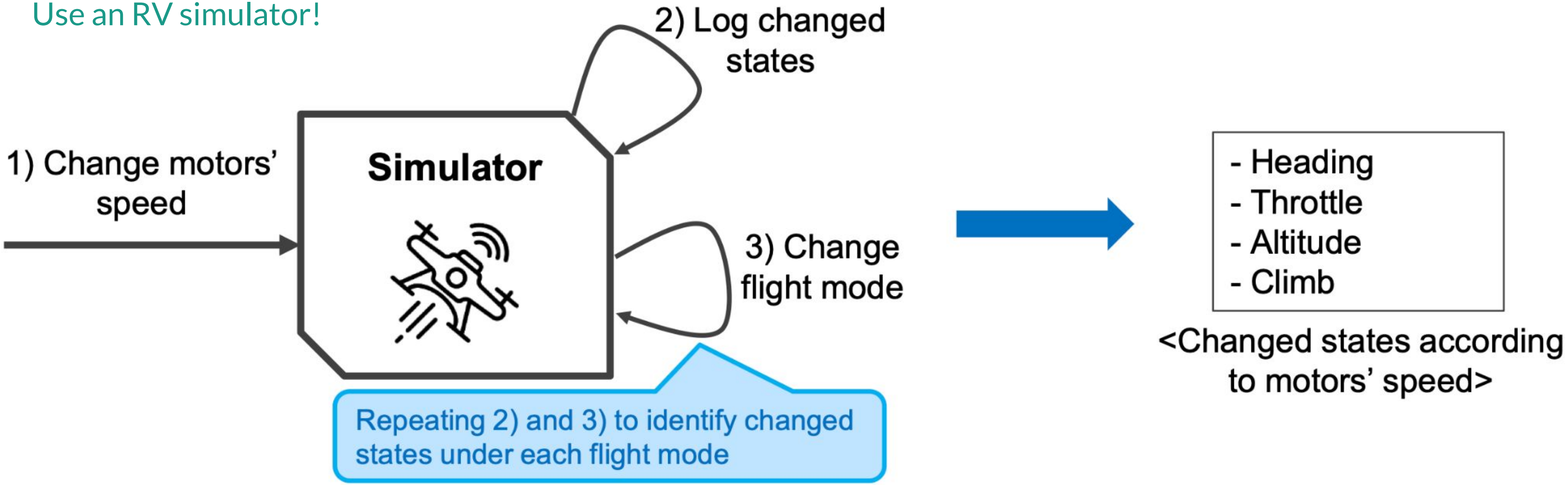


PGFuzz: Mapping other types of inputs to each term

(Reducing fuzzing space)

How to map environmental factors and user commands to each term from source code?

Use an RV simulator!



PGFuzz: Two types of distances to mutate inputs

(Building distance metrics)

Propositional distance: To efficiently mutate inputs. Quantifies how close a proposition to the policy violation

Positive value:
If the proposition is true
Negative value:
If the proposition is false

If the term is numeric, we use normalized difference.

$$\neg\Diamond \{(\text{Parachute=on})\} \wedge \{(\text{Mode}_t = \text{FLIP/ACRO}) \vee (\text{ALT}_t > \text{ALT}_{t-1})\}$$

$$P_1 = \begin{cases} 1 & \text{If parachute = on} \\ -1 & \text{If parachute = off} \end{cases}$$

$$P_2 = \begin{cases} 1 & \text{If mode = FLIP/ACRO} \\ -1 & \text{If mode} \neq \text{FLIP/ACRO} \end{cases}$$

$$P_3 = \frac{\text{ALT}_t - \text{ALT}_{t-1}}{\text{ALT}_t}$$

PGFuzz: Two types of distances to mutate inputs

(Building distance metrics)

Global distance: to detecting a policy violation

$$\neg\Diamond \{(\underline{Parachute=on})\} \wedge \{(\underline{Mode_t = FLIP/ACRO}) \vee (\underline{ALT_t > ALT_{t-1}})\}$$

$$-1 \times [Min\{P_1, Max(P_2, P_3)\}] \begin{cases} \text{Positive value} & \text{if there is no policy violation} \\ \text{Negative value} & \text{if the RV violates the policy} \end{cases}$$

PGFuzz: Example

(Building distance metrics)

$$P_1 = \begin{cases} 1 & \text{If parachute = on} \\ -1 & \text{If parachute = off} \end{cases}$$

$$P_3 = \frac{ALT_t - ALT_{t-1}}{ALT_t}$$

$$P_2 = \begin{cases} 1 & \text{If mode = FLIP/ACRO} \\ -1 & \text{If mode} \neq \text{FLIP/ACRO} \end{cases}$$

$$\boxed{-1 \text{ X } [\text{Min}\{P_1, \text{Max}(P_2, P_3)\}]}$$

Time (T)	Parachute (on/off)	FLIP/ACRO mode (T/F)	Altitude (m)	P_1	P_2	P_3	Global distance	Next input for Time T+1
1	off	false	90	-1	-1	0	1	Motor speed = 1,800[1)]
2	off	false	100	-1	-1	0.1	1	Motor speed = 1,800
3	off	false	110	-1	-1	0.09	1	Parachute = on
4	on	false	112	1	-1	0.02	-0.02	Policy violation!

Vehicle must not increase its altitude

Evaluation

RV control software

- **ArduPilot, PX4, and Paparazzi**

56 extracted policies

- Fuzzing 48 hours per each control software

Found **156 bugs**

- **Violating 14 policies** in the three-control software

PGPatch: Policy-Guided Logic Bug Patching for Robotic Vehicles

Hyungsub Kim, Muslum Ozgur Ozmen, Z. Berkay Celik, Antonio Bianchi, and Dongyan Xu
Purdue University

2022 IEEE S&P

Previous work : PGFUZZ

- Discovered 156 logic bugs using linear temporal logic formula
- Correct behavior vs Incorrect behavior defined using LTL

```
// Get a time delay to trigger position fail-safe
param_get(param_find("COM_POS_FS_DELAY"), &val);
// Force the valid range of the parameter
posctl_nav_loss_delay = math::constrain(val * sec_to_usec,
POSVEL_PROBATION_MIN, POSVEL_PROBATION_MAX);
```

Listing 2: GPS Fail-Safe Bug [41].

```
void Copter::failsafe_battery_event(void) {
  if (ap.land_complete)
    // Stop motors
  else if (g.failsafe_battery_enabled == FS_BATT_RTL
    && home_distance > wp_nav.get_wp_radius())
    // Switch to RTL
  else // Switch to LAND
```

Listing 4: Battery Fail-Safe Bug [13].

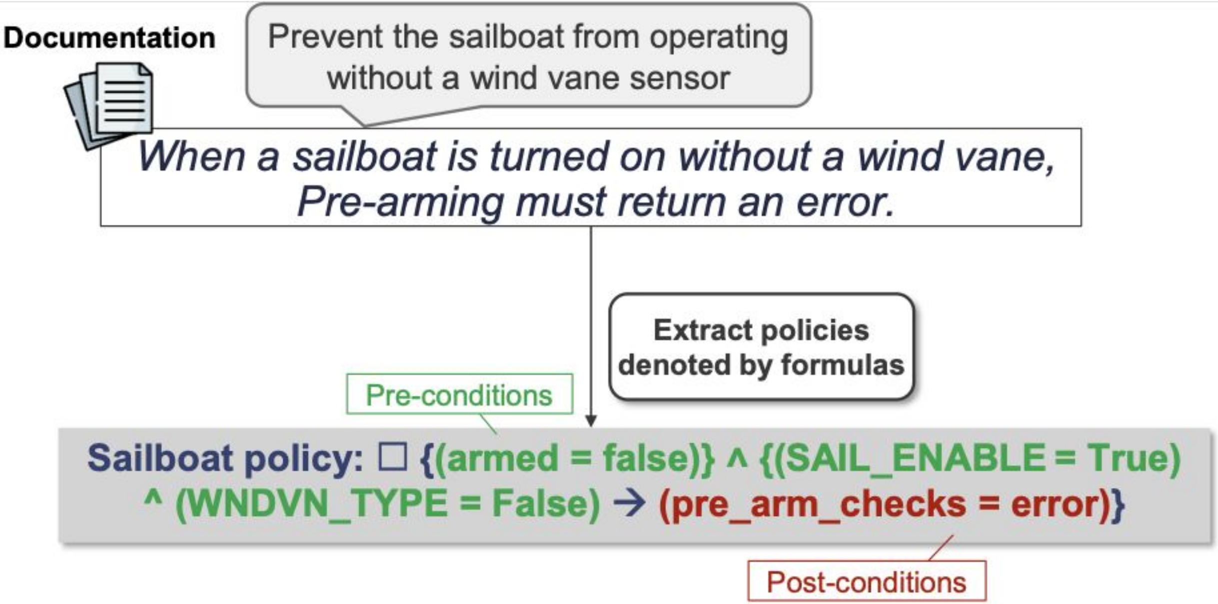
```
bool AP_Arming_Rover::pre_arm_checks() {
  if (rover.g2.sailboat.sail_enabled()
    && !rover.g2.windvane.enabled()) {
      printf("Sailing enabled with no WindVane");
      return false;
```

Listing 3: Sailboat Pre-Arming Bug [1].

```
void FlightTaskAutoMapper::_prepareLandSetpoints() {
  _constraints.tilt = _param_mpc_tiltmax_lnd.get();
...
bool FlightTaskManualAltitude::activate() {
  _constraints.tilt = _param_mpc_man_tilt_max.get();
```

Listing 5: Tilt Angle Bug [77].

Linear Temporal Logic



Main Idea: Can we fix these automatically?

Pre-conditions

Sailboat policy: □ {(armed = false)} ^ {(SAIL_ENABLE = True) ^ (WNDVN_TYPE = False) → (pre_arm_checks = error)}

Post-conditions

The RV software initially did not implement this policy, causing potential safety violations

```
bool AP_Arming_Rover::pre_arm_checks() {
  if (rover.g2.sailboat.sail_enabled()
    && !rover.g2.windvane.enabled()) {
      printf("Sailing enabled with no WindVane");
      return false;
```

Can we automatically fix these logical errors?

PGPatch

Program Repair

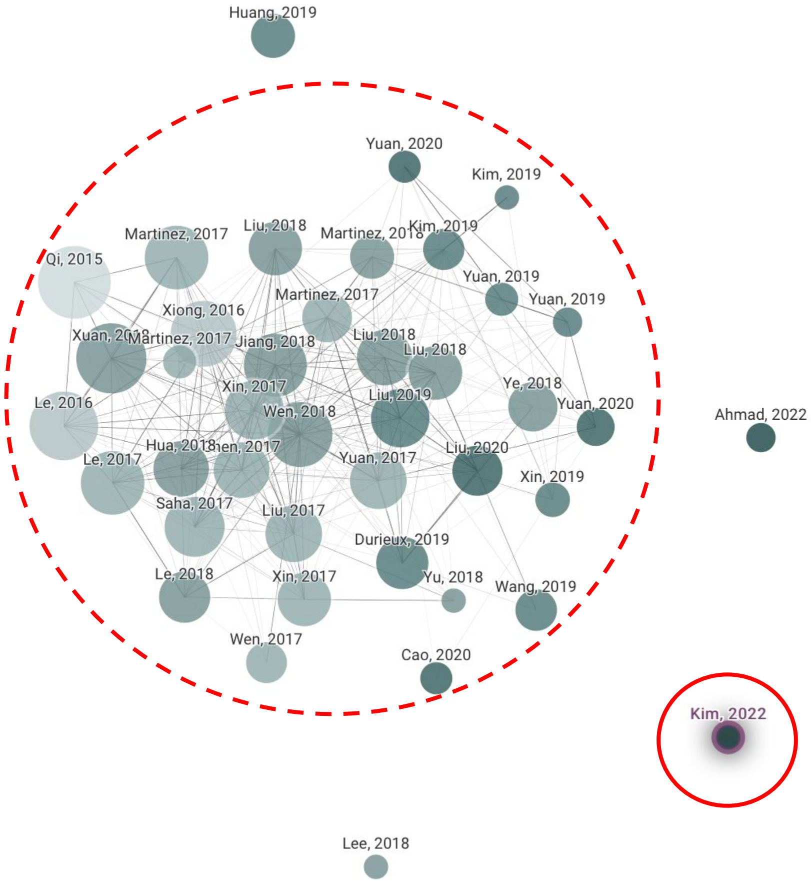


Image: Connected Papers

Limitations of existing tools

1. Largely focus on fixing memory corruption bugs
2. Need a complex set of test cases
3. Use constraint solver - Poor support for floating point operations

PROGRAM-REPAIR.ORG Home Bibliography Tools Benchmarks Pages Statistics Workshop Mailing list Repository

+ Add tool — publication, — repository

C/C++

- AllRepair — mutation-based repair tool for C programs equipped with assertions in the code
- Angelix — automated program repair tool based on symbolic analysis
- CPR — detecting and discarding over-fitting patches via systematic co-exploration of the patch space and input space
- CoderAssist — system for feedback generation
- DeepFix — tool for fixing common programming errors based on deep learning
- ErrDoc — tool that is able to detect, categorize and fix error handling bugs for C programs
- FAngelix — Faster Angelix that performs a guided search via MCMC sampling
- FixMorph — automated patch backporting tool for syntactically similar programs, i.e. across different versions
- GenProg — automated program repair tool based on genetic programming
- Kali — generate-and-validate patch generation system that only deletes functionality
- LeakFix — safe memory-leak fixing tool for C programs
- MemFix — static analysis-based repair tool for memory deallocation errors for C programs
- MintHint — program repair tool that generates repair hints to assist the programmer
- NEM — automated repair of heap-manipulating programs using deductive synthesis
- PatchWeave — automated patch transplantation for semantically equivalent programs
- Prophet — automated program repair that learns from correct patches
- RSRepair — GenProg modification that uses random search
- SPR — automated program repair tool with condition synthesis
- SearchRepair — automated program repair that uses semantic code search over large repositories of candidate code bases to produce high-quality bug patches
- SemFix — automated program repair tool based on symbolic analysis

Eiffel

- AutoFix — automatic program repair of object-oriented programs with contracts

Java

- ACS — automated program repair tool with accurate condition synthesis
- ARJA — multi-objective genetic programming for automated repair of Java
- AVATAR — fixing Java bugs by the fix patterns of static analysis violations (FindBugs violations)
- Astor — automatic software repair framework for Java (incl. GenProg, Kali and mutation repair for Java)
- CapGen — context-aware patch generation technique
- ConFix — automated patch generation with context-based change application
- GenPat — inferring program transformation from historical bug fixes via big code
- Genesis — system that automatically infers sets of code transforms for automatic patch generation
- HistoricalFix — automated program repair tool that leverages bug fix history
- JAID — an APR technique that uses detailed state abstractions to guide both fault localization and fix generation

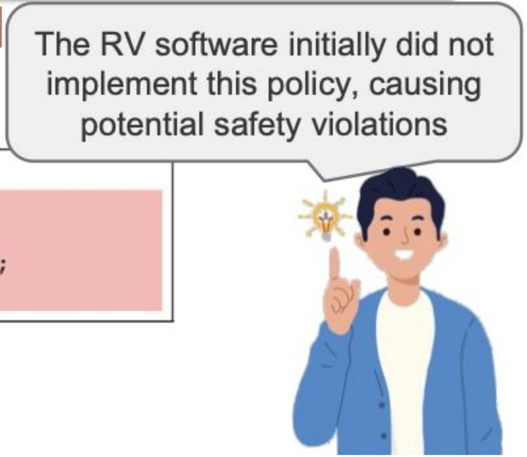
Can we reuse the LTL formulas to fix them automatically?

Pre-conditions

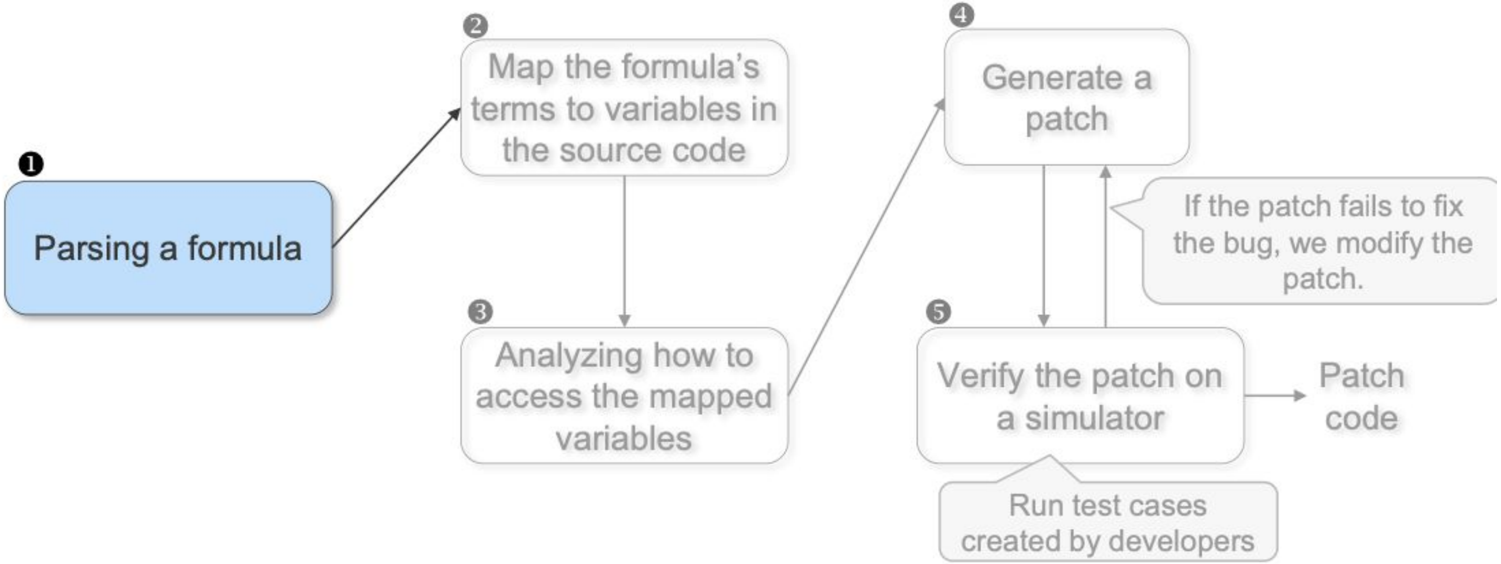
Sailboat policy: $\square$ {(armed = false)} ^ {(SAIL_ENABLE = True) ^ (WNDVN_TYPE = False) → (pre_arm_checks = error)}

Post-conditions

```
bool AP_Arming_Rover::pre_arm_checks() {
  if (rover.g2.sailboat.sail_enabled()
    && !rover.g2.windvane.enabled()) {
      printf("Sailing enabled with no WindVane");
      return false;
```

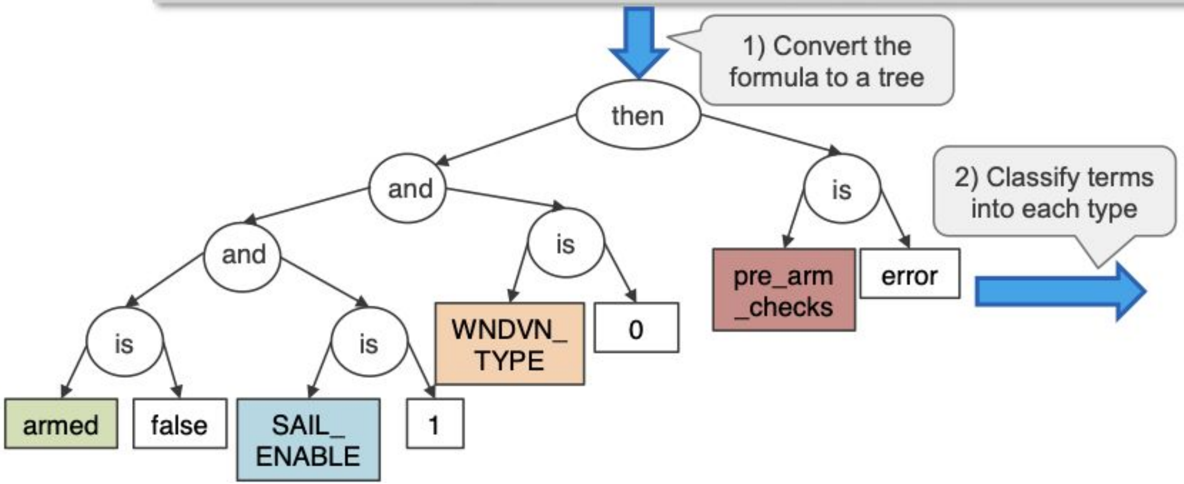


PGPatch: Overview



1. Parse the Formula

Sailboat policy in PPL syntax: If *armed* is *false* and *SAIL_ENABLE* is *1* and *WNDVN_TYPE* is *0*, then *pre_arm_checks* is *error*



Term	Type
armed	State (P)
SAIL_ ENABLE	State (C)
WNDVN_ TYPE	State (C)
pre_arm_ checks	Function

2. Map formula terms to variable in source code

Heuristics:

1. RV software port the configuration parameters from XML files to source code

2. RV software's strict coding conventions eg: Each variable's name denotes a physical state

Configuration parameter name

Class name

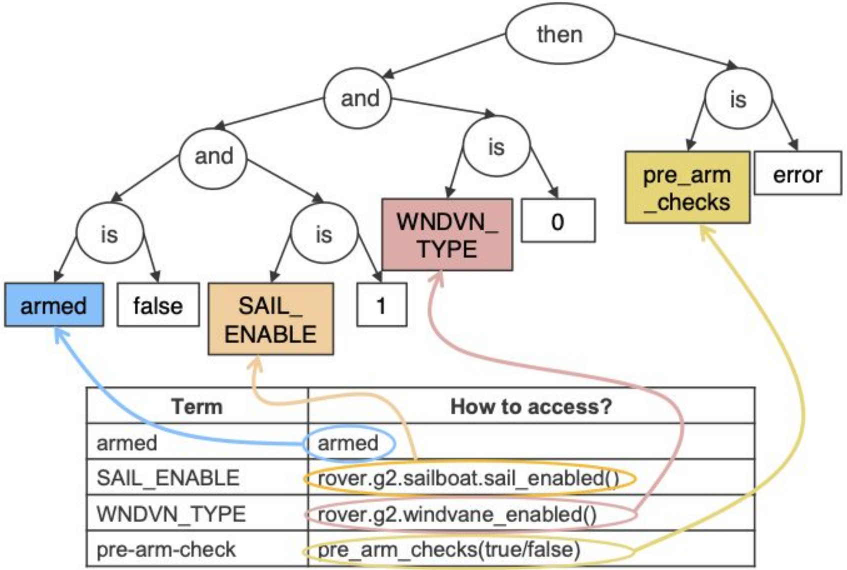
Variable name

```
AP_GROUPINFO_FLAGS("ENABLE", 1, Sailboat, enable, 0, AP_PARAM_FLAG_ENABLE)
```

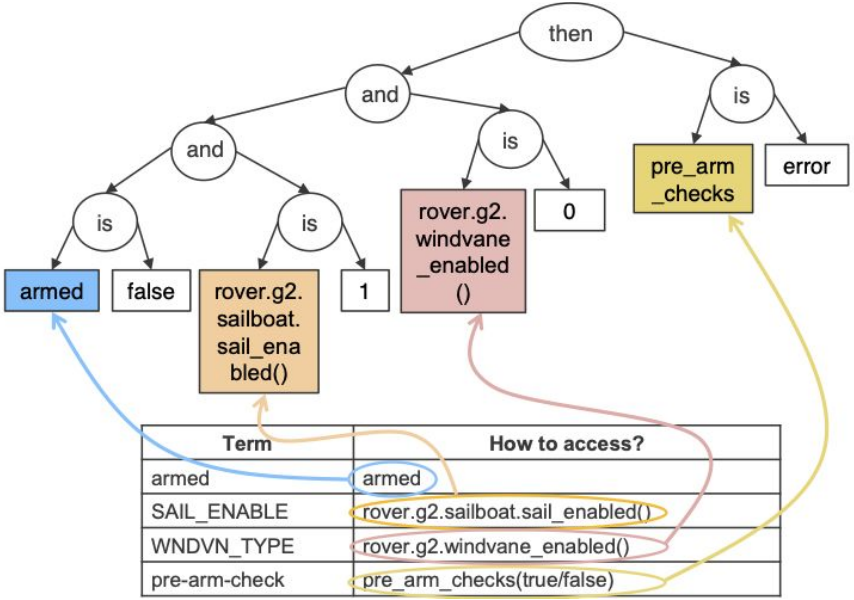
Term	Type
armed	State (P)
SAIL_ ENABLE	State (C)
WNDVN_ TYPE	State (C)
pre_arm_ checks	Function

Term	Mapped variables/functions
SAIL_ENABLE	Private ***enable*** in Sailboat class
WNDVN_TYPE	Private ***_direction_type*** in AP_WindVane class
pre-arm-check	***pre_arm_checks*** function

3. Analyze how to access the mapped variables

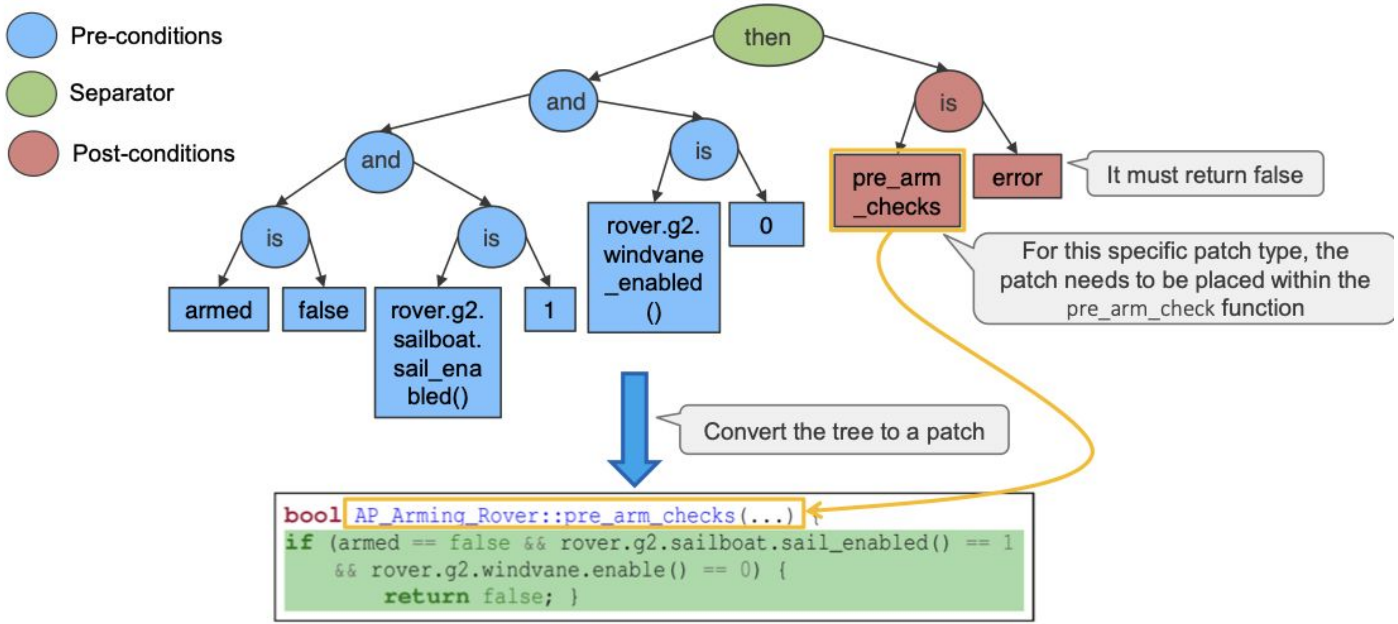


Term	How to access?
armed	armed
SAIL_ENABLE	rover.g2.sailboat.sail_enabled()
WNDVN_TYPE	rover.g2.windvane_enabled()
pre-arm-check	pre_arm_checks(true/false)

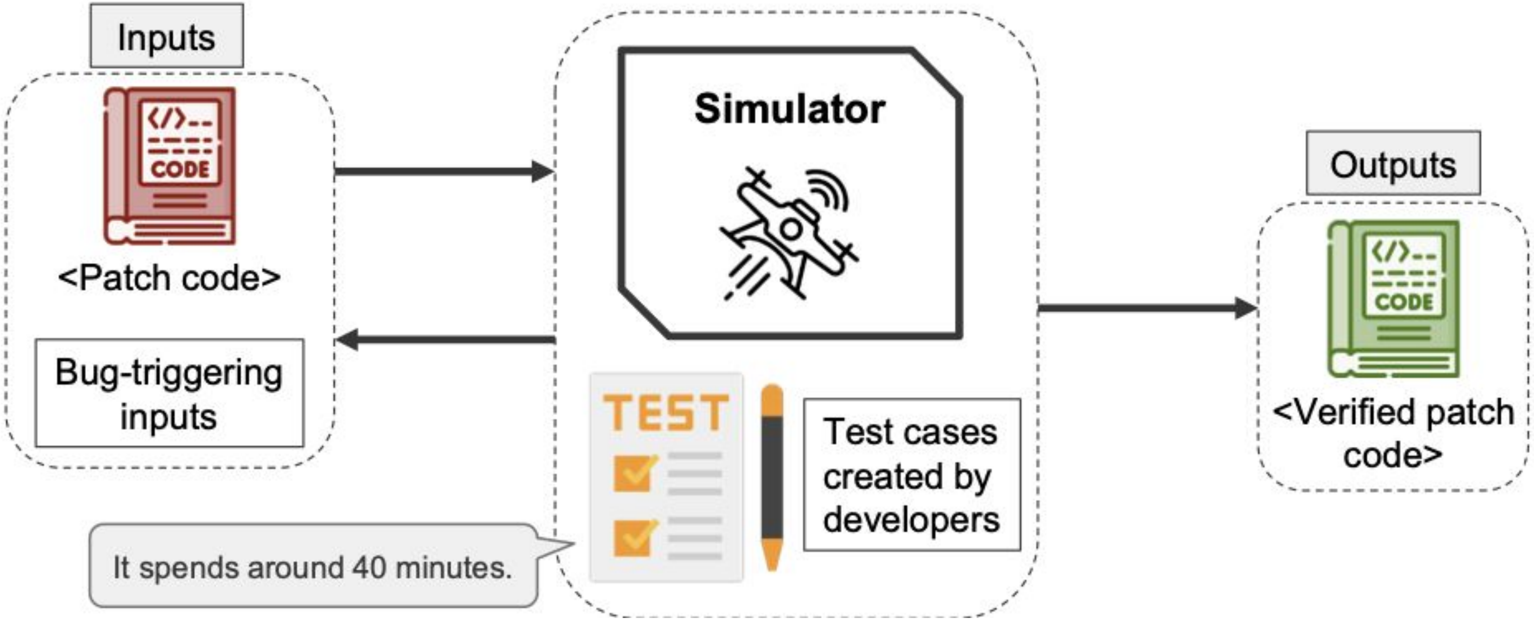


Term	How to access?
armed	armed
SAIL_ENABLE	rover.g2.sailboat.sail_enabled()
WNDVN_TYPE	rover.g2.windvane_enabled()
pre-arm-check	pre_arm_checks(true/false)

4. Generate patch



5. Patch Verification



Supports 5 patch types

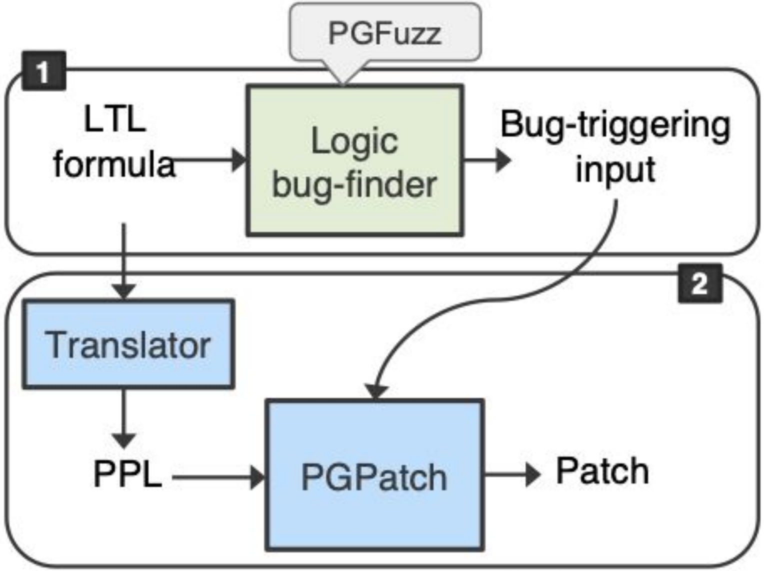
1. Disabling a statement
2. Checking valid ranges of configuration parameters
3. Updating a statement
4. *Adding a condition check*
5. Reusing an existing code snippet

Evaluation

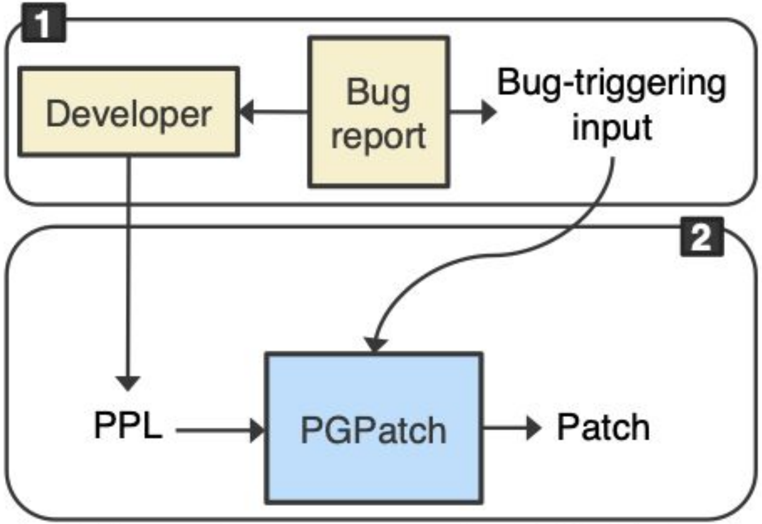
	Selected bugs	Patchable bugs	Fixed bugs
ArduPilot (A)	70	38	32
PX4 (PX)	70	27	24
Paparazzi (PP)	70	29	21
Total	210	94	77

- RV control software
  - ArduPilot
  - PX4
  - Paparazzi
- Dataset
  - 94 logic bugs from GitHub commit history
  - 203 logic bugs from RV fuzzing works (PGFuzz and RVFuzzer)
- PGPatch succeeds in fixing 258 out of 297 bugs
  - **86.9% success rate**

User Study: Usage scenario



1. First usage scenario of PGPatch



2. Second usage scenario of PGPatch

User Study: Evaluation

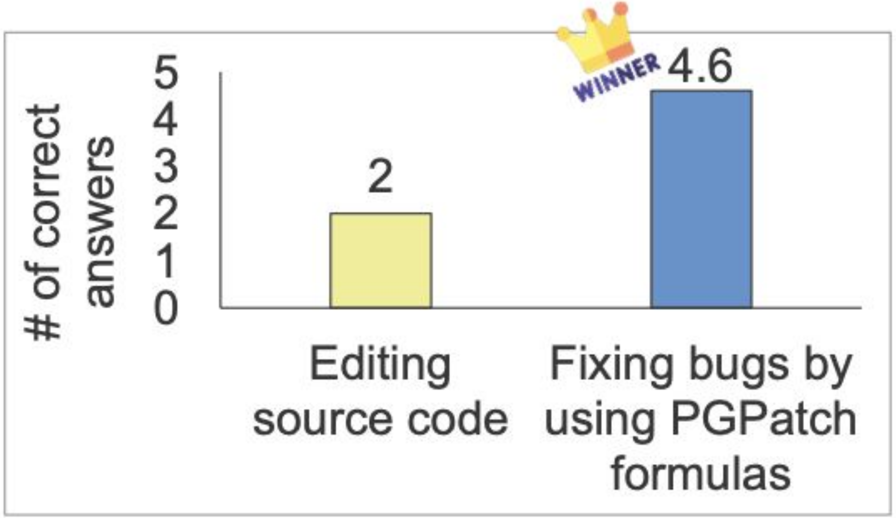
How efficient is PGPatch in patching logic bugs **compared to manual patching** ?

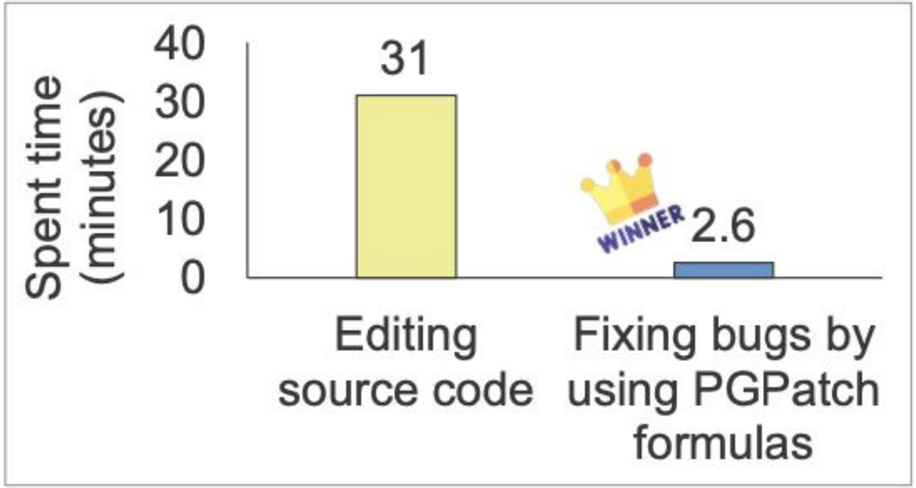
- Recruit
  - 6 RV developers
  - 12 experienced RV users
  - 1 subject was an official ArduPilot developer
- Ask participants to create
  - 5 PGPatch formulas
  - 5 corresponding source-level patches

Bug origin	Fuzzing			Commit history		
	A	PX	PP	A	PX	PP
Fixed bugs	140	24	17	32	24	21
Performance damage	0	0	0	0	0	0
Different from developers' patches	N/A	N/A	N/A	2	0	0
Total	181			77		

TABLE III: Summary of the qualitative evaluation.

User Study: Evaluation





Is less error-prone compared to manually patching bugs

Logical Bugs in Drones and Swarms (1)

A survey of recent papers

Presented by Akshith for A58

Oregon State University

Papers:

Part 1 **No code! Just behavior.**

a. SwarmFlawFinder: Discovering and Exploiting Logic Flaws of Swarm Algorithms, Jung et. al,
IEEE Symposium on Security & Privacy May 2022

Part 2 **Yes code! Code analysis.**

b. PGFuzz: Policy-Guided Fuzzing for Robotic Vehicles, Kim et. al.
The Network and Distributed Systems Security Symposium, Feb 2021

c. PGPatch: Policy-Guided Logic Bug Patching for Robotic Vehicles, Kim et. al.
IEEE Symposium on Security & Privacy May 2022

Threat Model

We know what the mission and algorithm is !

No sensor spoofing, No malware in the system !

Basically looking for **design flaws** in the algorithm / software implementation.

Not memory corruption bugs

Only logical bugs

Why Focus on Logical Bugs?

- Survey of 1250 Software Bugs:

  92.8% Logical Bugs

  1.8% Memory Corruption Bugs
- 97% of logical bugs can **lead to real physical harm**.

SwarmFlawFinder: Discovering and Exploiting Logic Flaws of Swarm Algorithms

Chijung Jung, Ali Ahad*, Yuseok Jeon†, and Yonghwi Kwon**

*University of Virginia , †UNIST

2022 IEEE S&P

Drone Swarms

Complex system of drones that **coordinate** to complete a task.

- Search and Rescue
- Monitoring wildfires
- Agricultural Shepherding

Motivation

1. Test with adversarial scenarios.
2. Show critical logical flaws in swarm algorithm.

Systematize by building an effective and automated **test system** to find **critical logic flaws** in swarm algorithms.

which is similar to …

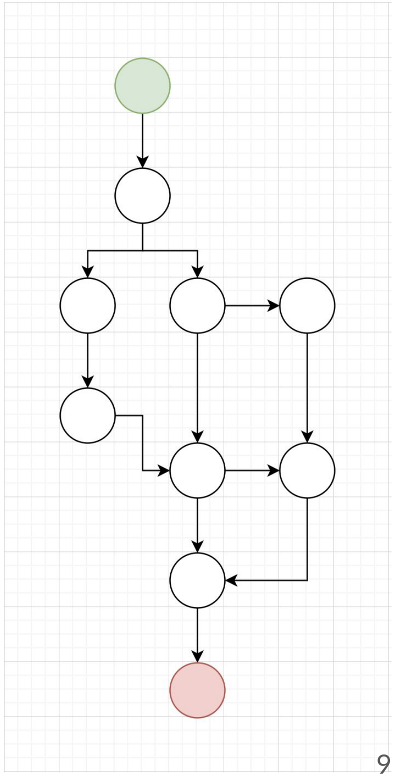
Fuzzing in Traditional Software Testing

How to efficiently find **test inputs** that **cause a crash** due to **software flaw?**

Random (Fuzz) Testing **Traditional Software**

- random data as **test inputs** to a program
  - efficient strategies exist
- monitor for **crashes**, or potential **memory leaks**

coverage is a good proxy for how good a **test input** is



But… Random (Fuzz) Testing a **Swarm Systems**

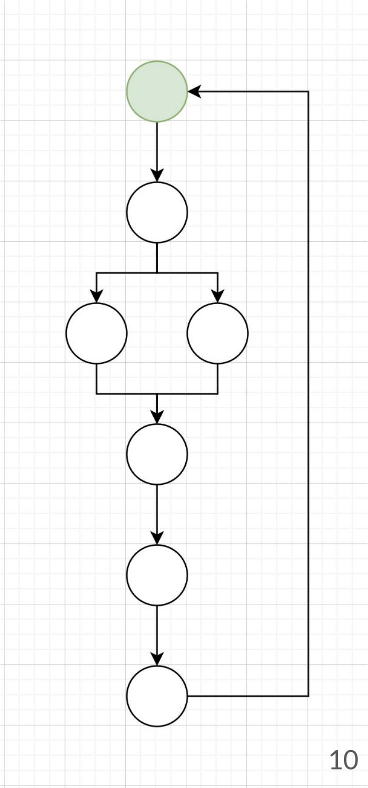
coverage is **NOT** a good proxy for how good a **test input** is

Robotic system in general are designed to have:

- less-diverse control flow
- more-diverse data variance

Makes traditional software coverage-based methods

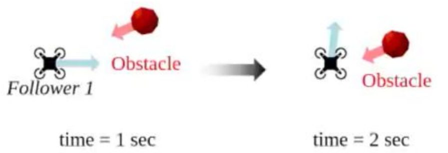
- ineffective in determining a test cases effectiveness
- ineffective in guiding the test generation



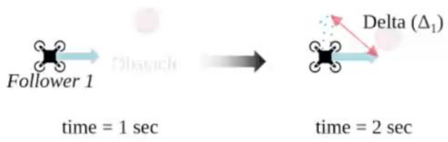
Contributions

1. Based on the idea of **Counterfactual Execution**
2. Proposes an **abstraction of Swarm's Behavior** (DCC - Degree of Causal Contribution)
3. **Fuzz based on DCC** as feedback.

Step 1 Counterfactual Execution

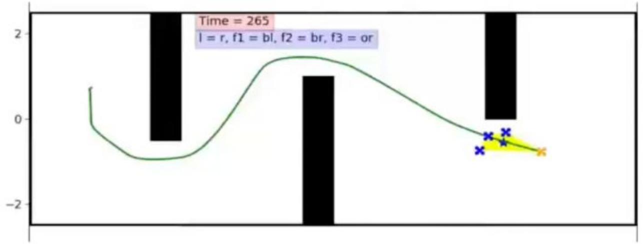


Original execution

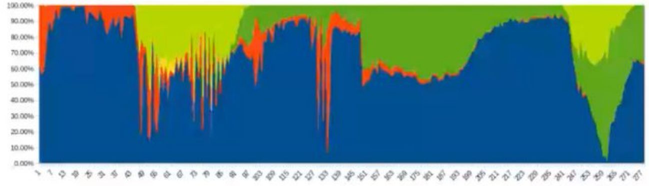


Counterfactual execution
(*without the obstacle*)

Step 2 Abstraction of Swarm's Behavior

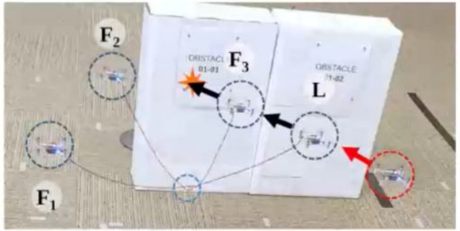


Top-view of the simplified mission



Degree of Causal Contribution (DCC) for Follower 1

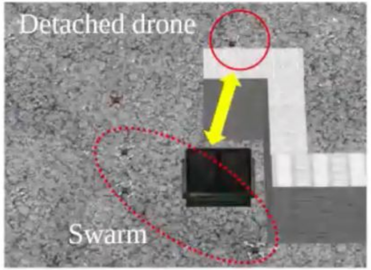
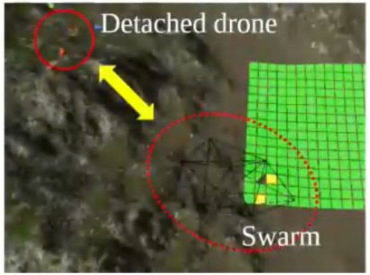
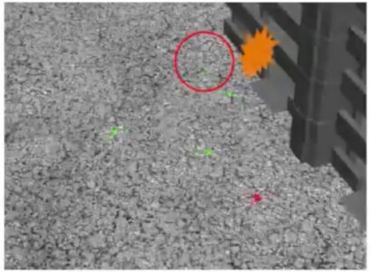
Step 3 Greybox Fuzzing using DCC as feedback



Attack drone causing a victim drone (F3) to crash into the wall (physical experiment).

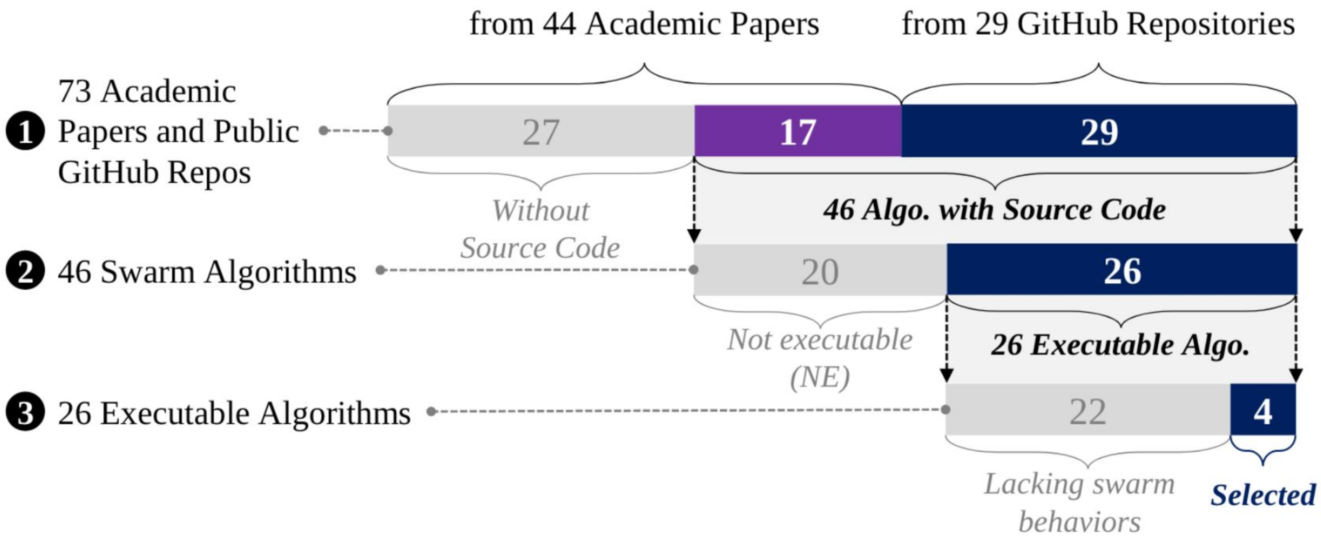
Corresponding event in simulation.

42 Logic Flaws

Name	Adaptive Swarm	SocraticSwarm	Sciadro	Pietro's
SLOC	3,091	9,920	3,851	752
Objective	Multi-agent navigation	Coordinated search	Distributed target search	Coordinated search and rescue
Unique # of logic flaws	20	8	6	8
Example of logic flaws	 Drone is detached from a swarm	 Drone is detached from a swarm	 Drone crashes into external objects	 Victim drones try to detour without considering the surrounding

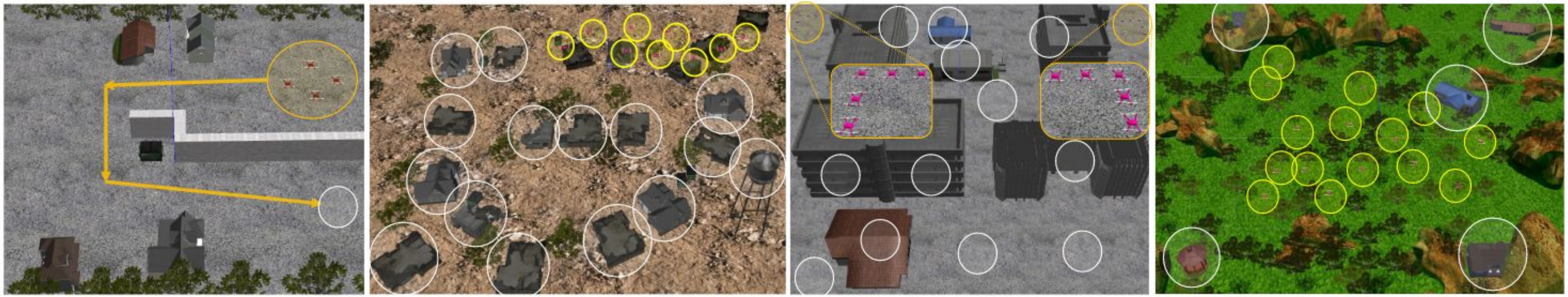
SwarmFlawFinder

Swarm Algorithms that were tested



Algorithm Selection Process

Swarm Algorithms



(a) Adaptive Swarm (Navigation) (b) SocraticSwarm (Coordinated search) (c) Sciadro (Distributed search) (d) Pietro's (Search and rescue)

collision avoidance logic is present in all 4

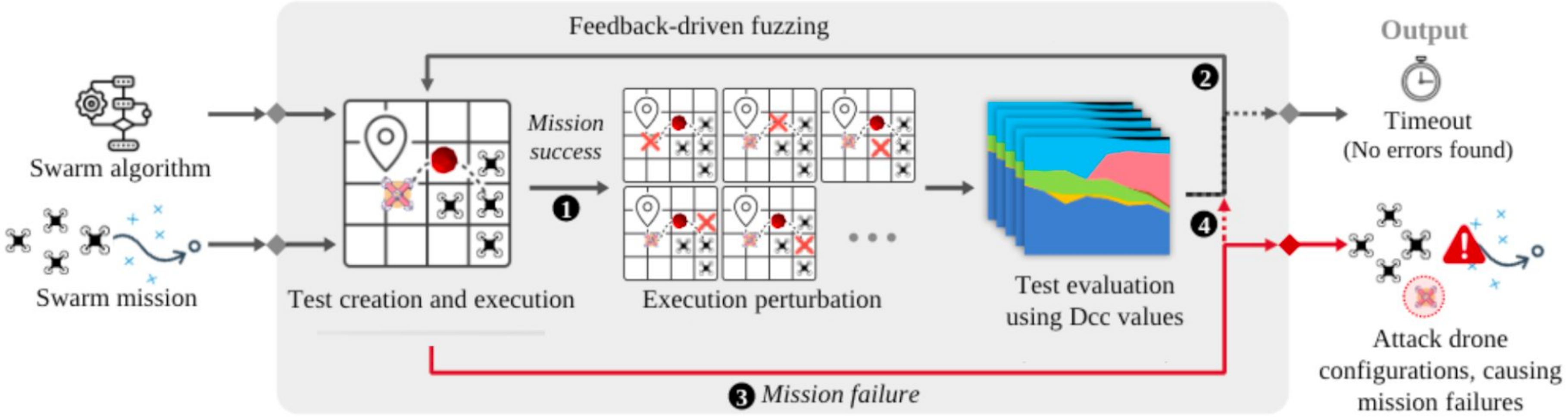
Threat Model … again

We know what the mission and algorithm is !

No sensor spoofing, No malware in the system !

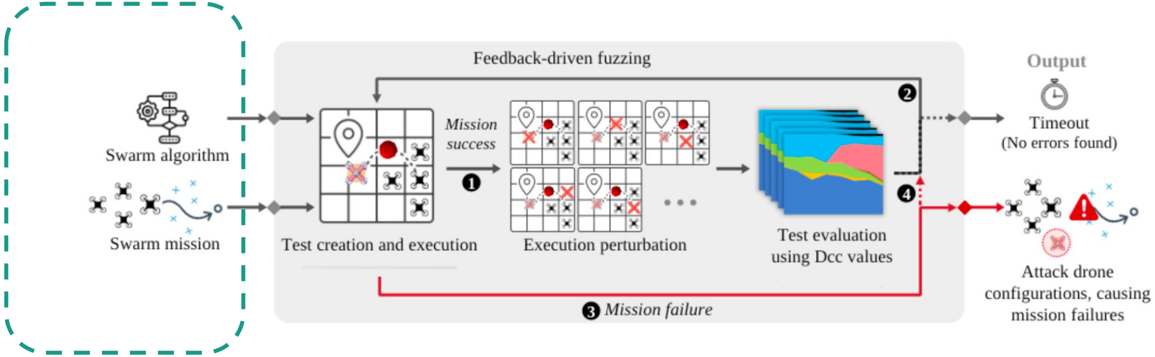
Basically looking for **design flaws** in the algorithm / software implementation.

Overview - Testing Loop



We Know/Given

- Swarm Mission
- Swarm Algorithm



We know what the mission and algorithm is !

No sensor spoofing, No malware in the system !

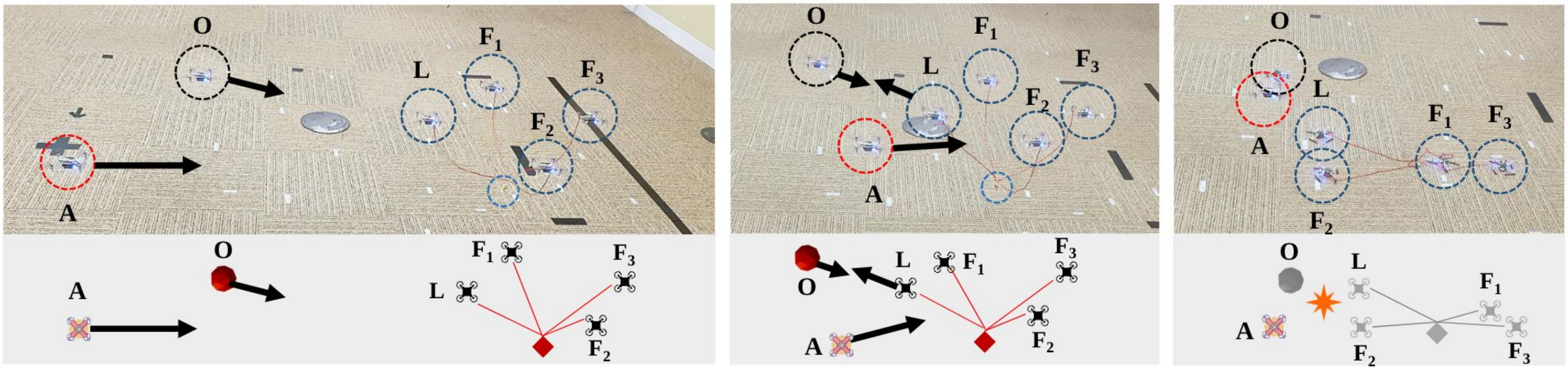
Basically looking for design flaws in the algorithm / software implementation.

We know what the mission and algorithm is !

No sensor spoofing, No malware in the system !

Basically looking for **design flaws** in the algorithm / software impleme

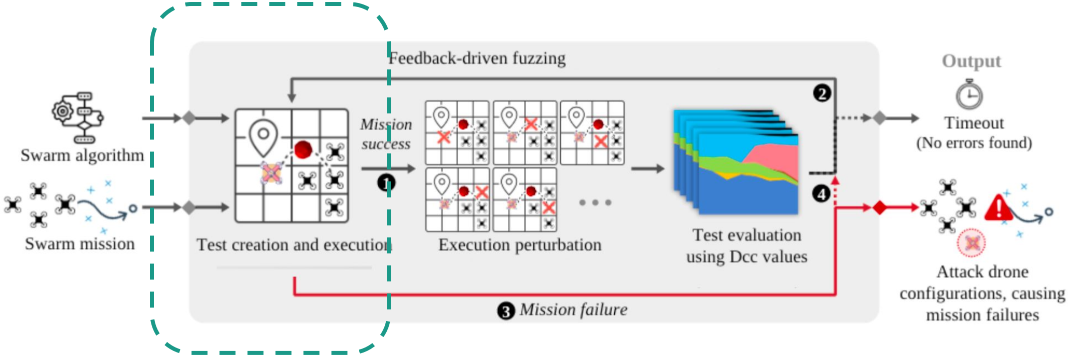
How do we provide a test input?



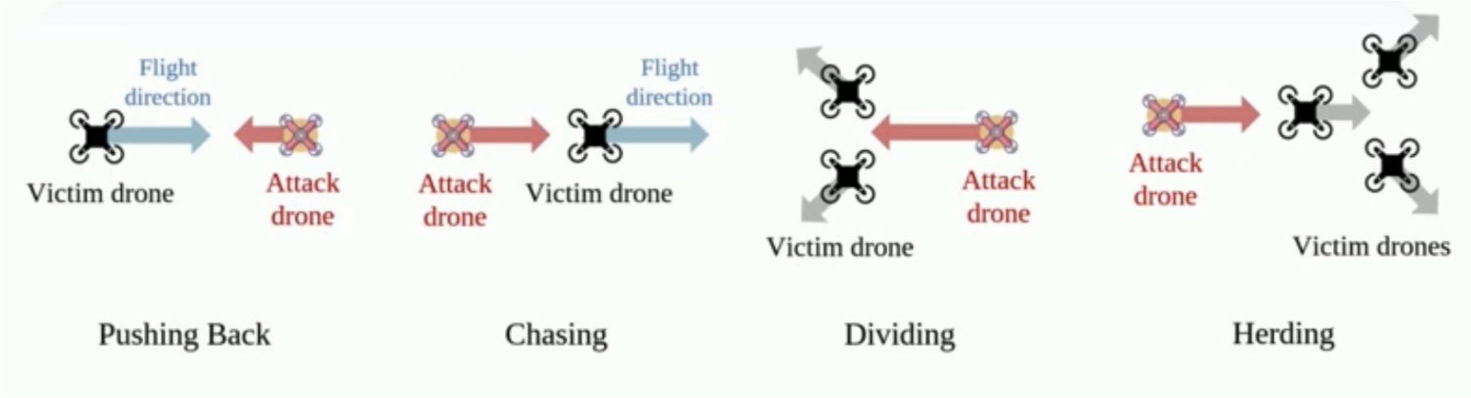
(a) Attack drone and obstacle approach the victim swarm (b) Attack drone influences a victim drone (c) Leader drone crashed into the obstacle

Use an attack drone.

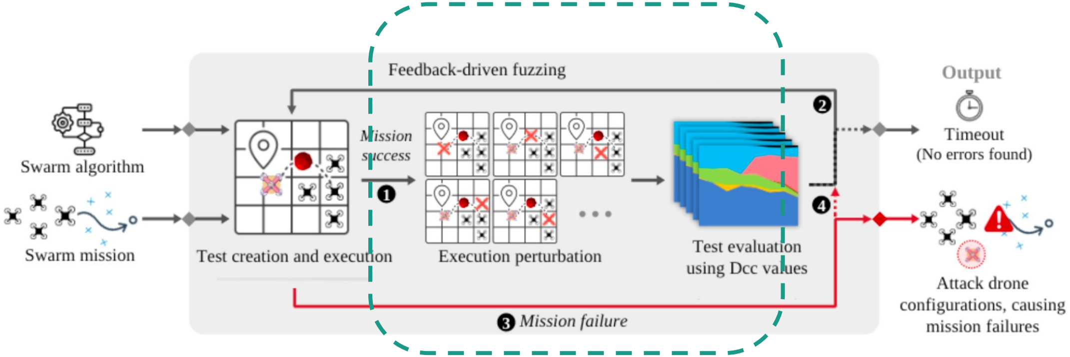
Test Case



- attack pose : {x,y,z}
- attack strategy : {chasing}



Exec and Eval



Did the mission **succeed or fail**?

Drone crashes

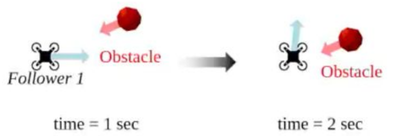
Takes more than 2x the time relative to the unperturbed execution (without attack drone).

Did the attacker cause a **new behaviour**?

But how do you define a behavior? What is a new behavior?

Degree of Causal Contribution (based on counterfactual execution)

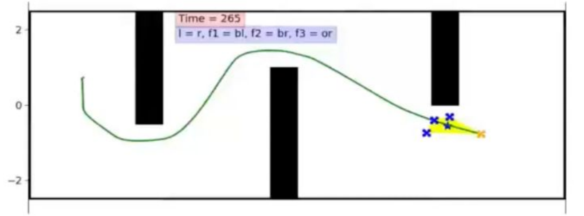
Step 1 Counterfactual Execution



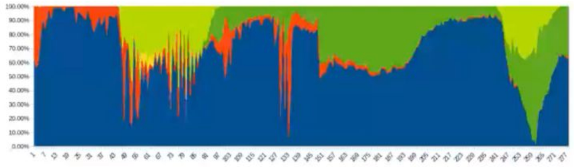
Original execution

Counterfactual execution
(*without the obstacle*)

Step 2 Abstraction of Swarm's Behavior

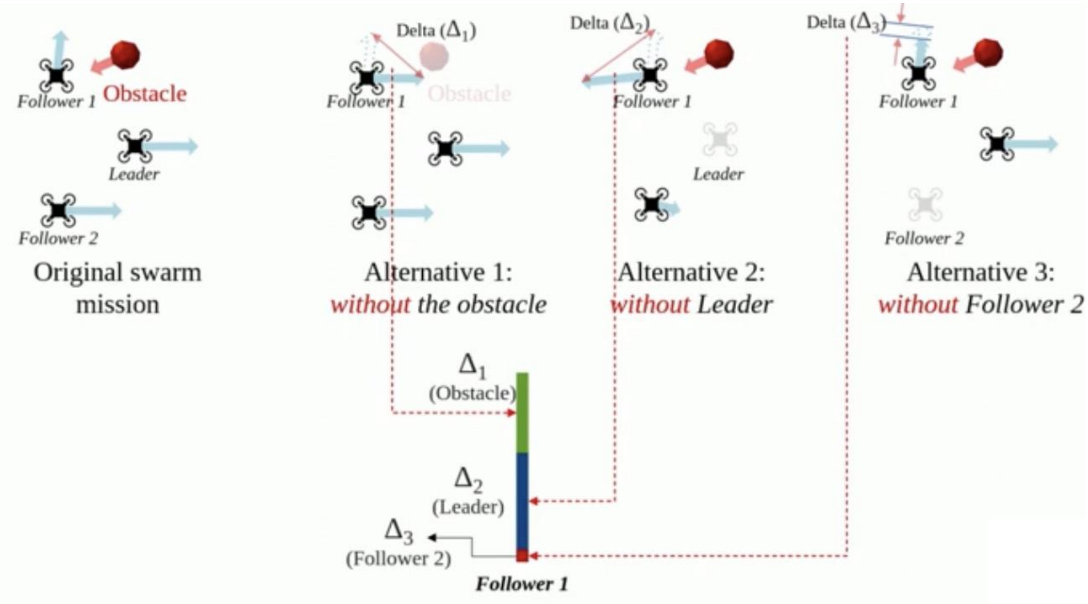


Top-view of the simplified mission



Degree of Causal Contribution (DCC) for Follower 1

Degree of Causal Contribution (w.r.t Euclidean Distance)



Degree of causal contribution (DCC) values

Normalized Cross Correlation:NCC (Degree of similarity of the DCC)

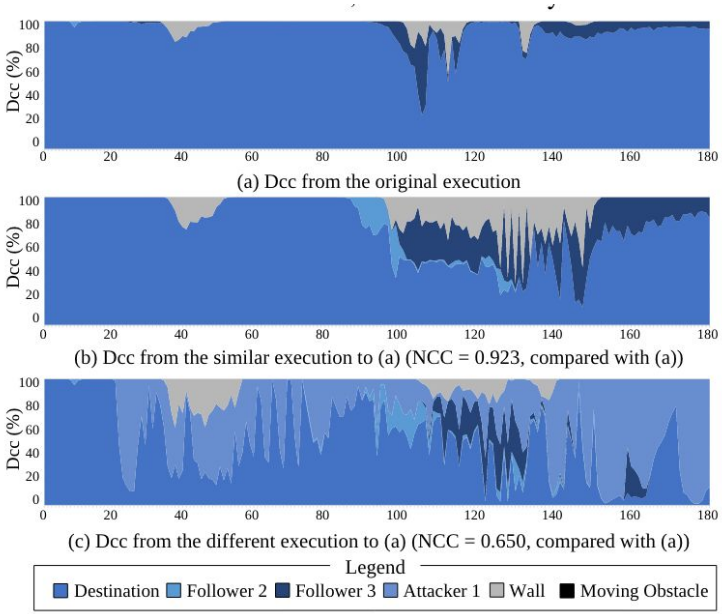


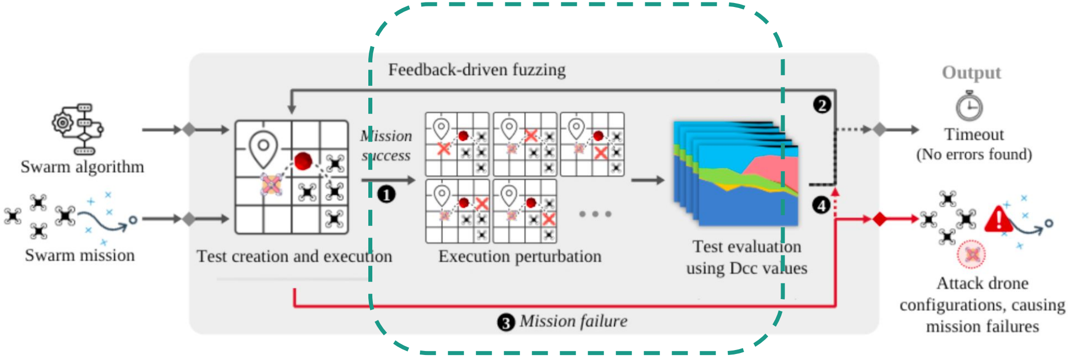
Fig. 6. Example of NCC scores from three executions.

Original Execution

Test Input causing **Similar Behavior**

Test Input causing **New Behavior**

Exec and Eval



Did the mission **succeed or fail**?

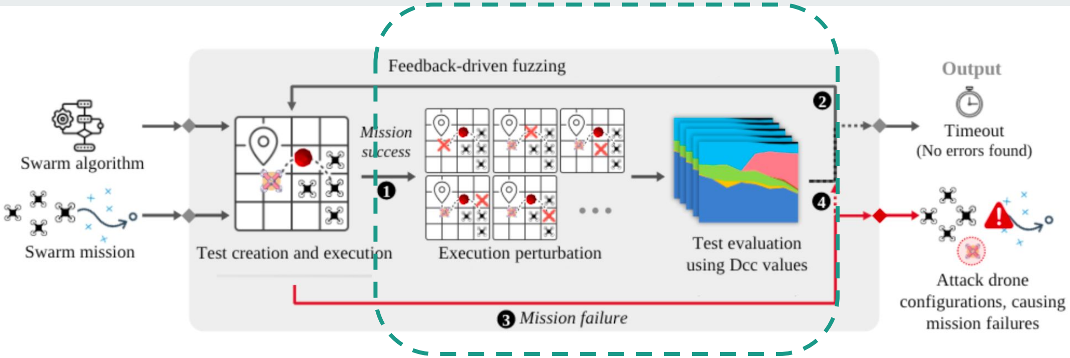
Drone crashes

Takes more than 2x the time relative to the unperturbed execution (without attack drone).

Did the attacker cause a **new behaviour**?

But how do you define a behavior? What is a new behavior?

DCC and NCC

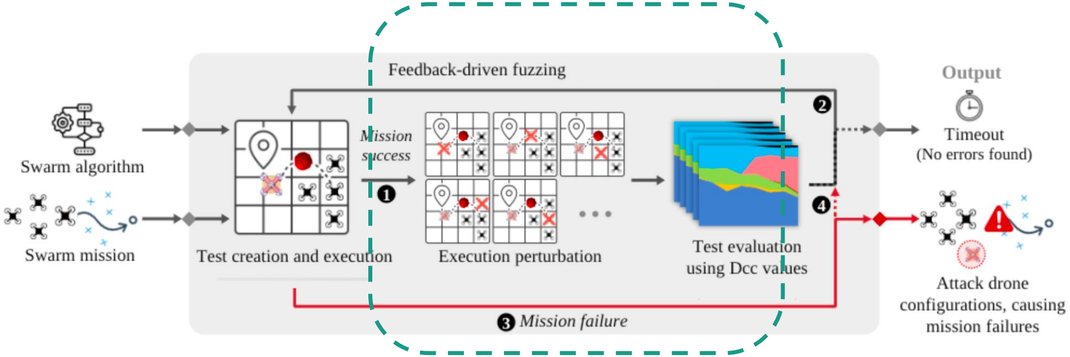


Test Case Mutation

- if New Behavior:
  - make small mutation
  - change pose alone
- if Same Behavior:
  - make big mutation
  - change pose and strategy

Baseline

Exec and Eval



Did the mission **succeed or fail**?

Drone crashes

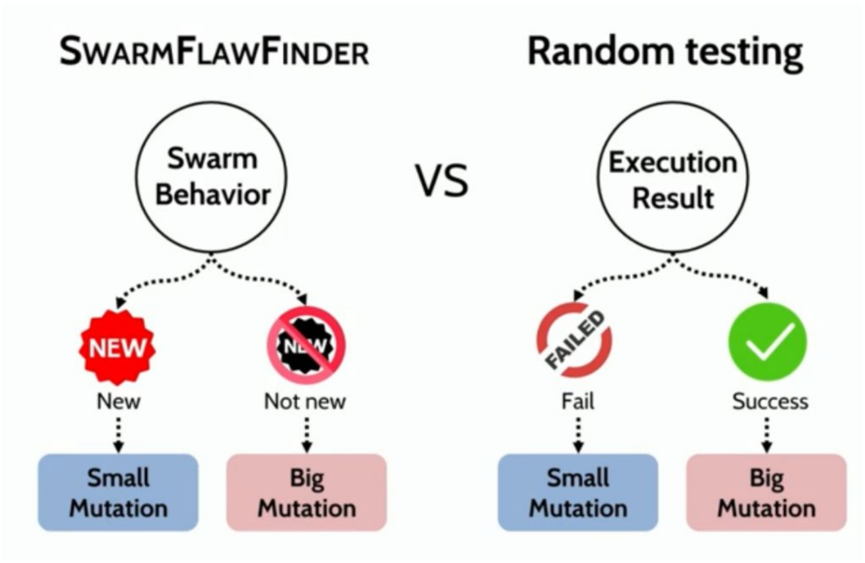
Takes more than 2x the time relative to the unperturbed execution (without attack drone).

Did the attacker cause a **new behaviour**?

But how do you define a behavior? What is a new behavior?

DCC and NCC

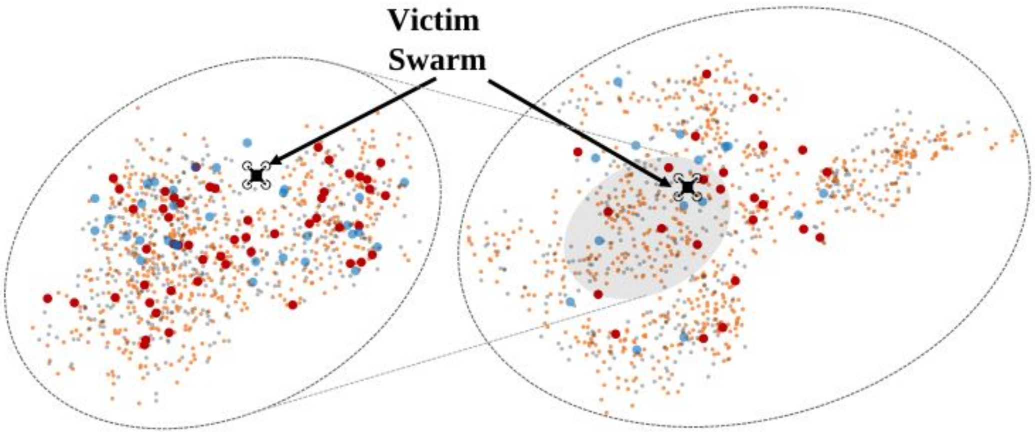
SwarmFlawFinder vs Random testing



Results

How effective are the test cases generated from SwarmFlawFinder vs the Random testing?

Test cases Distribution



(a) Visualized test cases generated for **A1** by SWARMFLAWFINDER

(b) Visualized test cases generated for **A1** by the random testing approach

Legend:

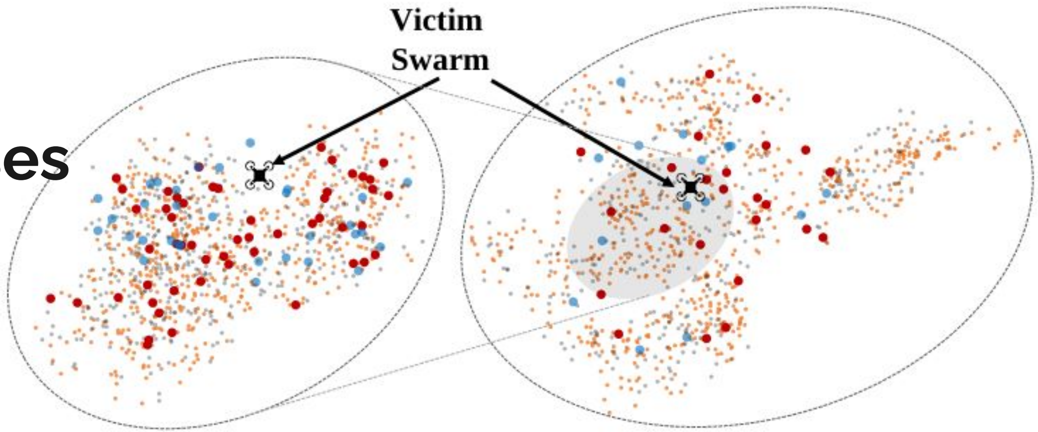
Red dots - Mission Failures
Blue dots - Mission Success

Big Dots - Unique Behavior
Small Dots - Duplicate Behavior

plotted w.r.t initial spatial distance of the attacker from the swarm.

Distribution of Test cases

- 2x more unique swarm behaviors
- in much smaller search space
- 25% more failure cases



(a) Visualized test cases generated for **A1** by SwarmFlawFinder

(b) Visualized test cases generated for **A1** by the random testing approach

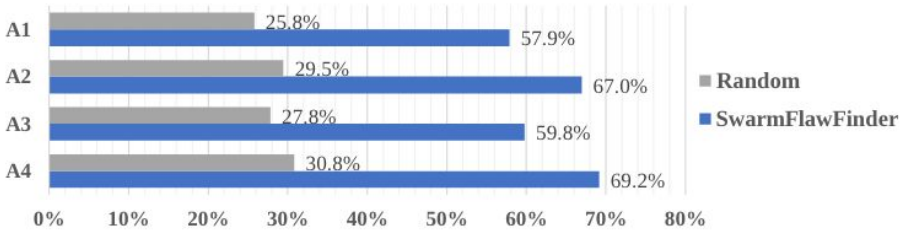
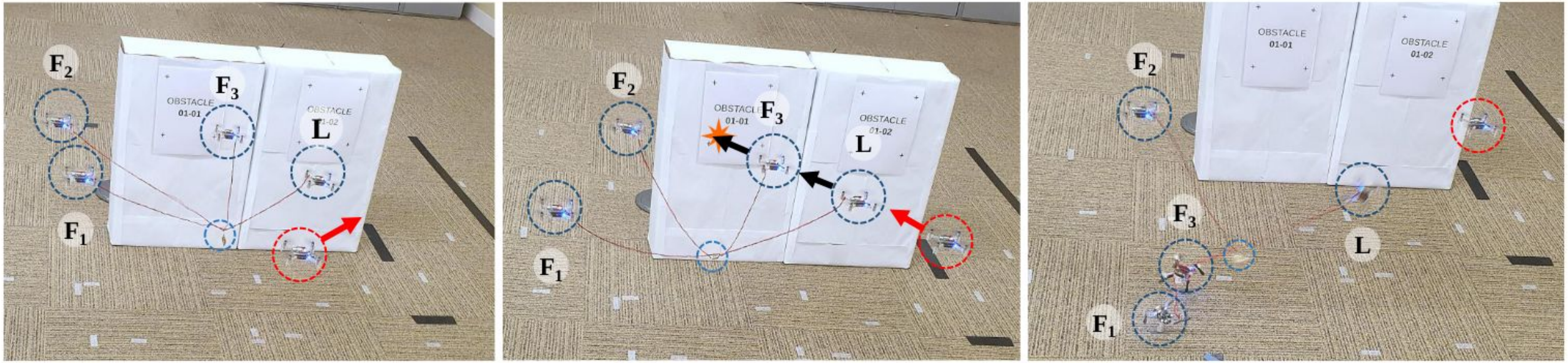


Fig. 9. Coverage of Unique Dcc Values.

Root Cause Analysis (Manual Analysis)

Root cause: Example 1



(a) Attack drone pushes the leader drone

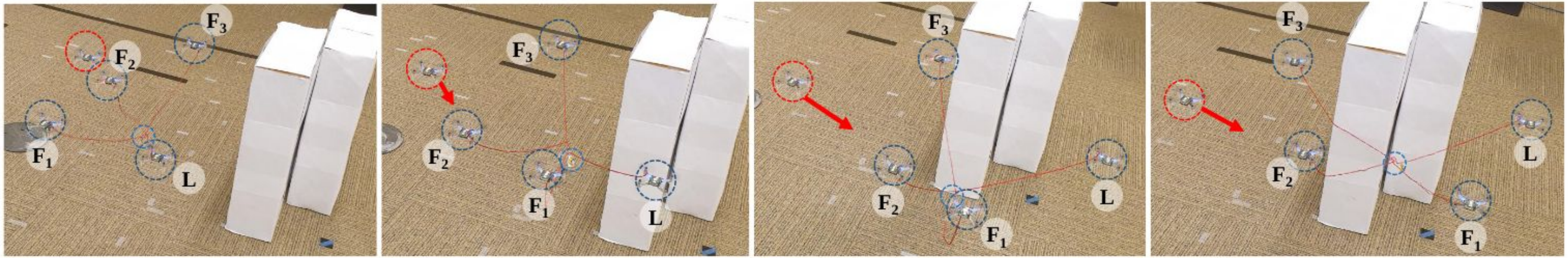
(b) Leader drone moves back without considering the F_3, making F_3 crashing into the wall

(c) Mission failed with a crash (The attack drone still alive)

Fig. 10. Attack drone causing a victim drone (F_3) to crash into the wall.

leader does not consider followers as external objects

Root cause: Example 2



(a) Attack drone chases a victim drone

(b) The chased victim drone blocks the other drone's way, making it stuck behind the wall

(c) Other drones make progress

(d) The entire swarm cannot make progress due to the drone stuck behind the wall

Fig. 11. Attack drone pushes a victim drone F_2 to suspend the swarm's progress.

algorithm computes the centroid of all drones to measure the current position of the swarm
as the centroid is not falling behind, the leader keeps moving forward

Comments

Comments

The authors acknowledge that:

- there can be **more sophisticated attack strategies**, which may improve the SwarmFlawFinder's performance.
- **do not argue** that DCC is a direct abstraction of the swarm behavior, but it is an approximation of the abstraction.
- however **argue** that it captures the behavior differences of swarm algorithms effectively.

Thank you!

Extra Slides

A Logical Flaw wrt Naive Multi Force Handling

Naive Multi Force handling

1. F_G = Force towards goal
2. F_O = Force against obstacle
3. F_A = Force against attacker

Net Force results in a Crash

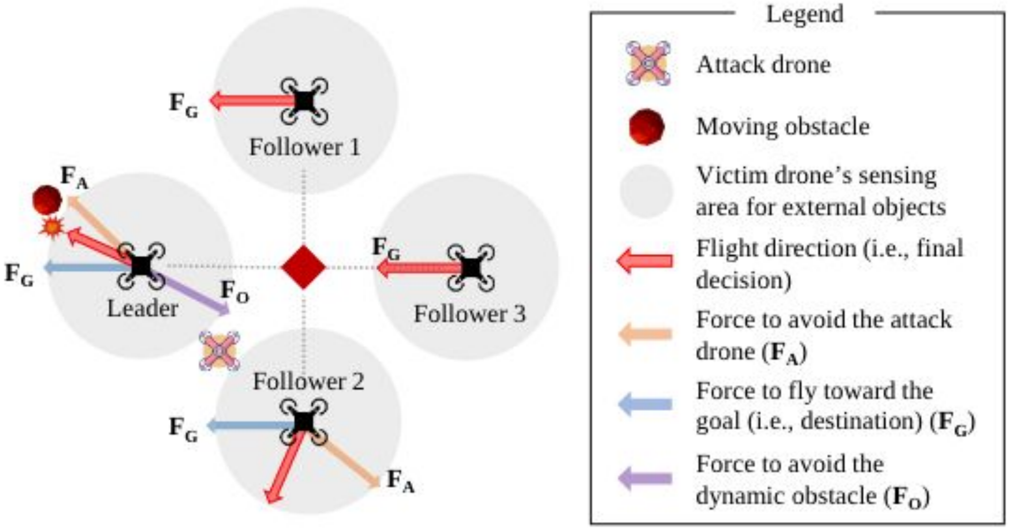
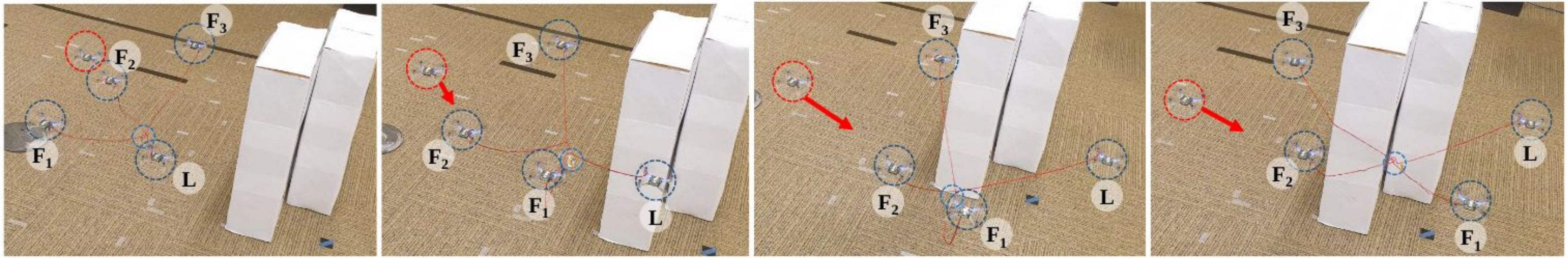


Fig. 3. Crash (caused by a logic flaw) found by SWARMFLAWFINDER.

Attack Example



(a) Attack drone chases a victim drone

(b) The chased victim drone blocks the other drone's way, making it stuck behind the wall

(c) Other drones make progress

(d) The entire swarm cannot make progress due to the drone stuck behind the wall

Fig. 11. Attack drone pushes a victim drone F_2 to suspend the swarm's progress.

DCC Calulation

DCC based on Counterfactual Causality

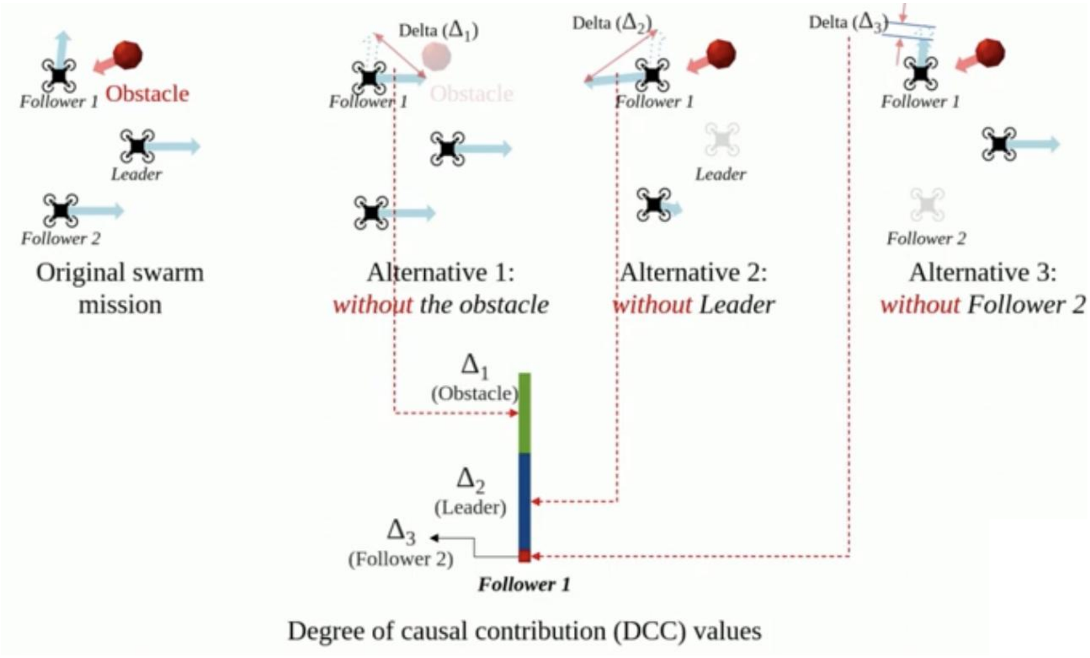
Degree of Causal Contribution DCC : Impact of external factors measured by drone's reaction.

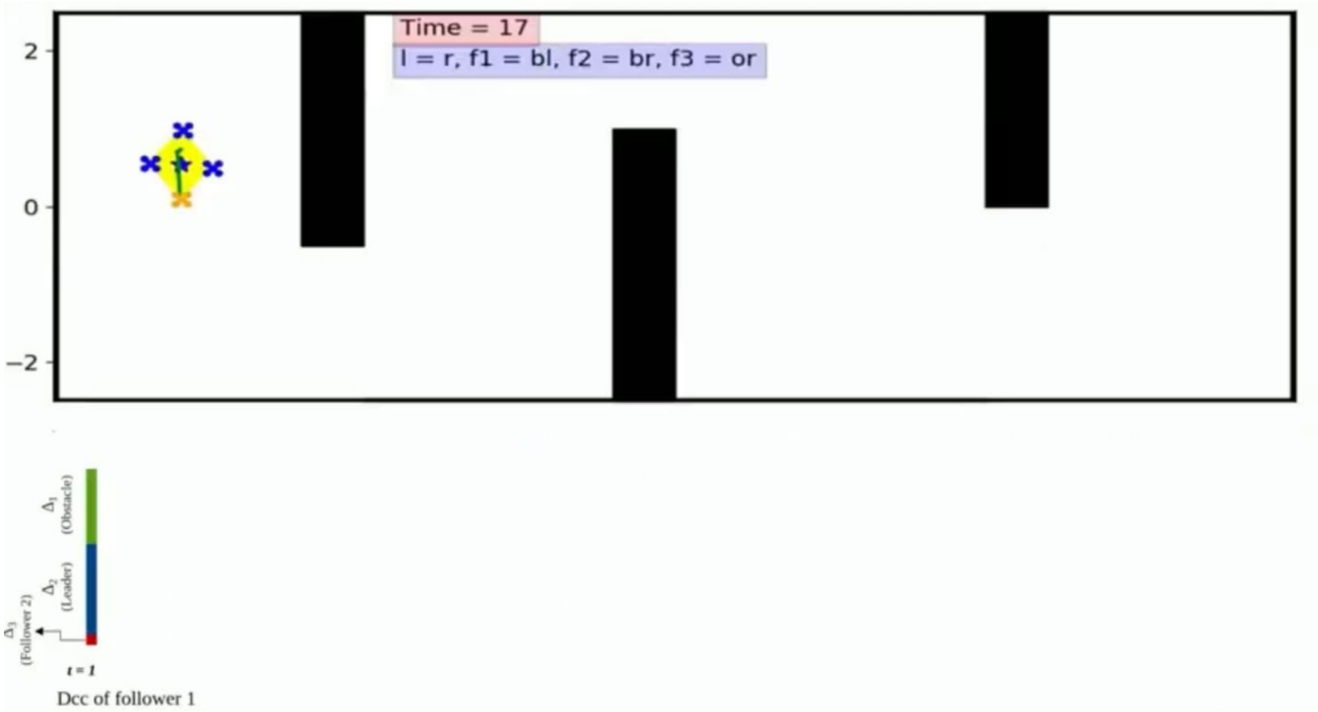
Counterfactual causality: If A has not occurred, B would not have occurred.

Actual execution - all external factors present.

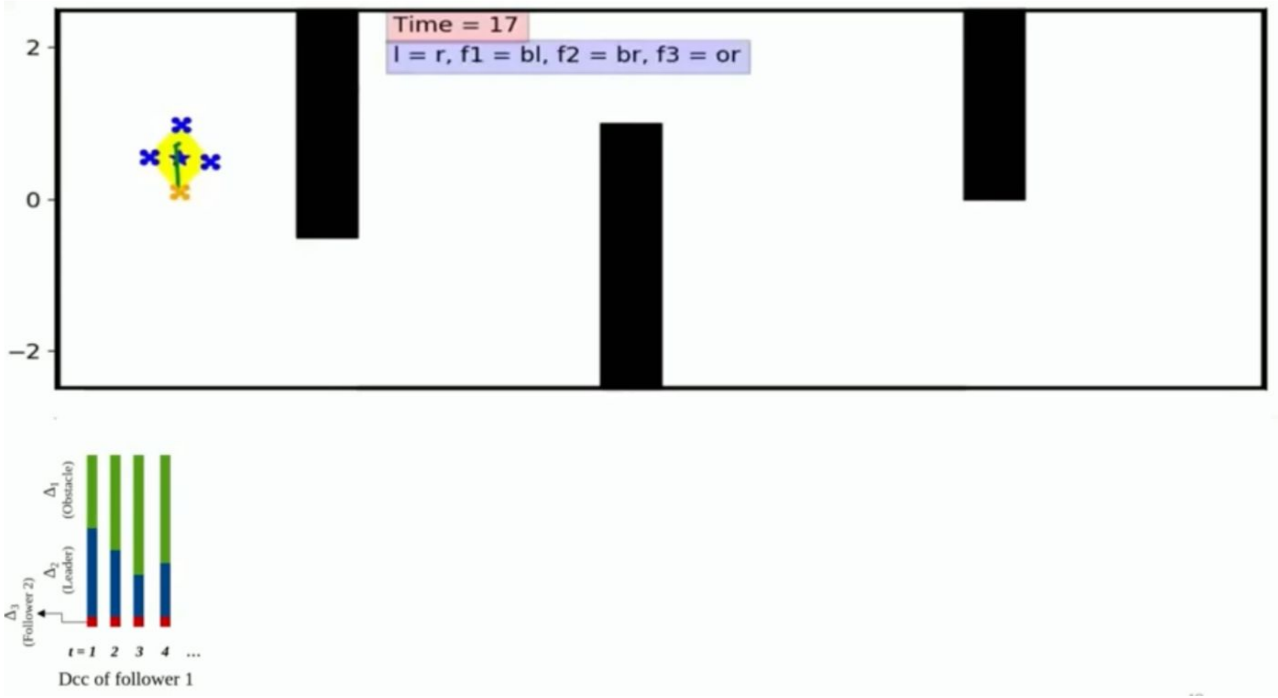
Alternative Executions - without external factors one at a time.

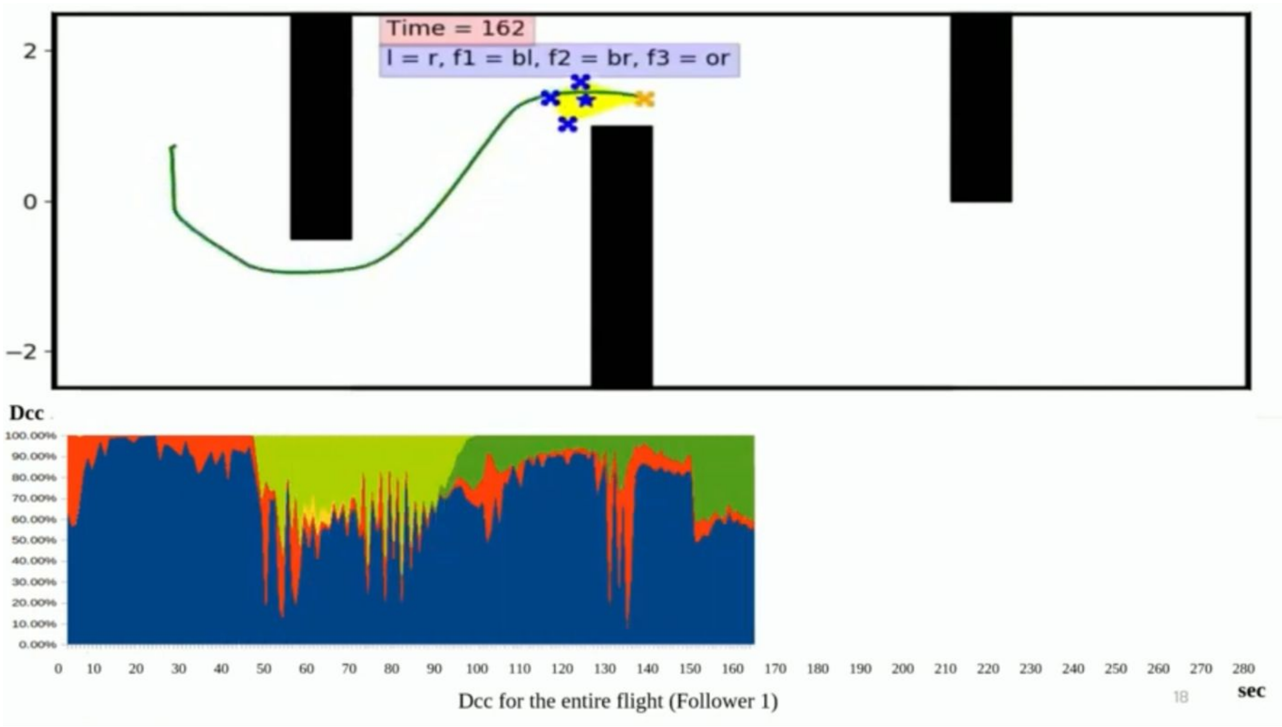
Computing DCC



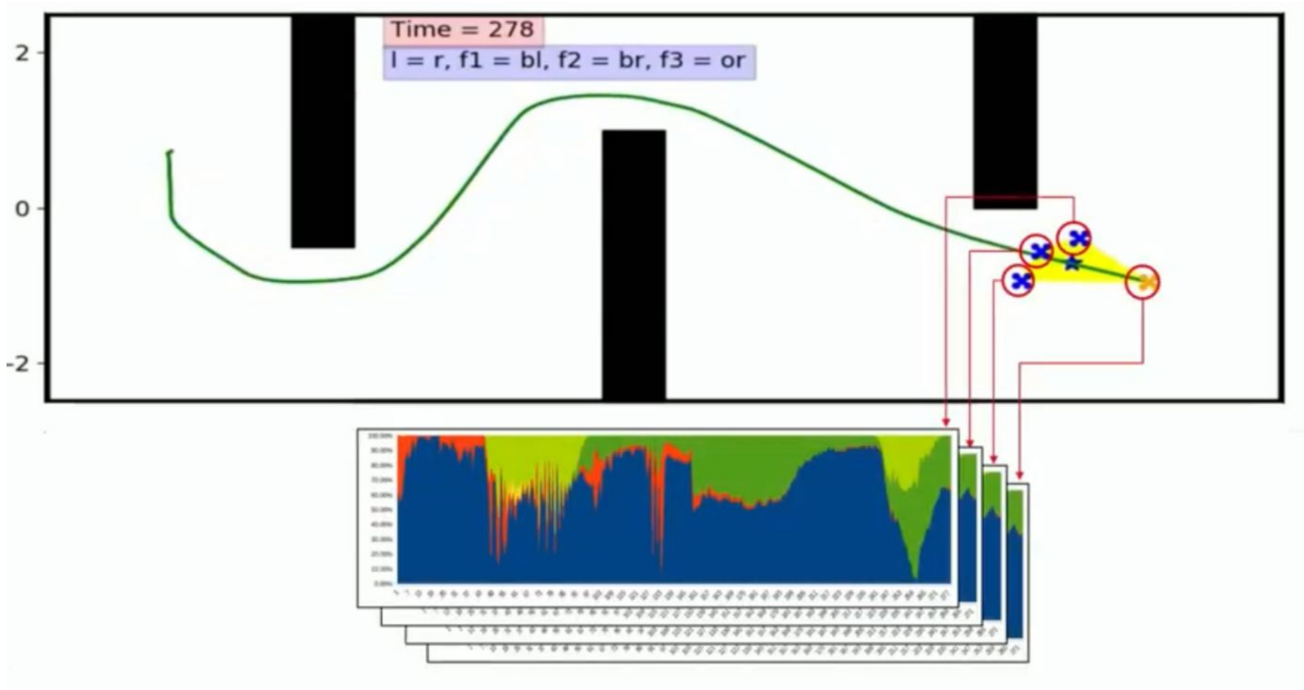


Time = 17
l = r, f1 = bl, f2 = br, f3 = or
2
0
−2
Δ1
(Obstacle)
Δ2
(Leader)
Δ3
(Follower 2)
t = 1
Dcc of follower 1


Time = 17
l = r, f1 = bl, f2 = br, f3 = or
2
0
−2
Δ1 (Obstacle)
Δ2 (Leader)
Δ3 (Follower 2)
t = 1 2 3 4 ...
Dcc of follower 1



Time = 162
l = r, f1 = bl, f2 = br, f3 = or
Dcc
Dcc for the entire flight (Follower 1)
sec



Time = 278
l = r, f1 = bl, f2 = br, f3 = or
2
0
-2

Thank you!

Questions?

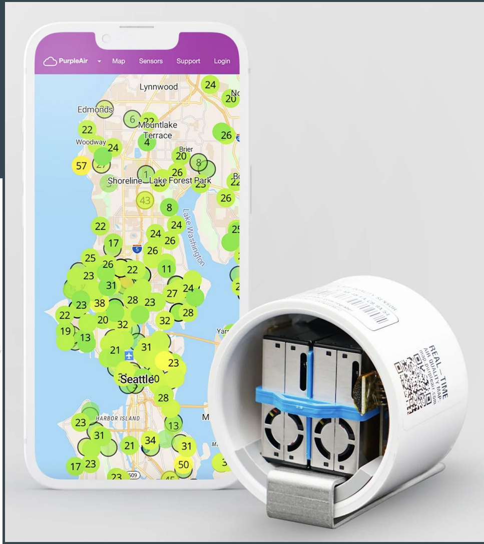
System Call Processing Using Lightweight NLP for IoT Behavioral Malware Detection

John Carter, Spiros Mancoridis, Malvin Nkomo, Steven Weber & Kapil R. Dandekar

Introduction

- IoT devices have quickly become used in many aspects of everyday life, such as security cameras, UAVs, air quality sensors and many more, which makes their security increasingly important
- In this work, we look at a small, yet usable, IoT ecosystem as a testbed for deploying and detecting malware
- Specifically, we use a multi-modal open-source IoT platform named **VarIoT** which has dozens of connected devices

https://www.amazon.com/Security-Wireless-Outdoor-Spotlight-Detection/dp/B09DSMZ387

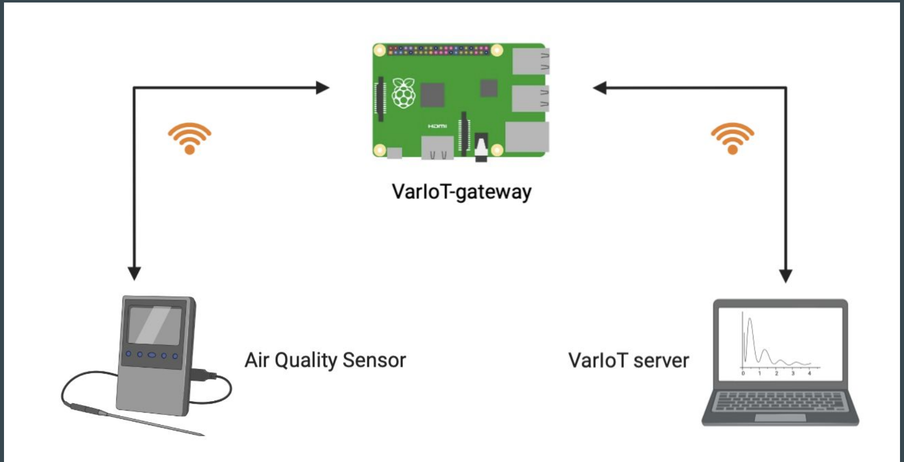


https://www2.purpleair.com/products/purpleair-pa-ii-flex


https://mashable.com/deals/june-8-foldable-hd-drone

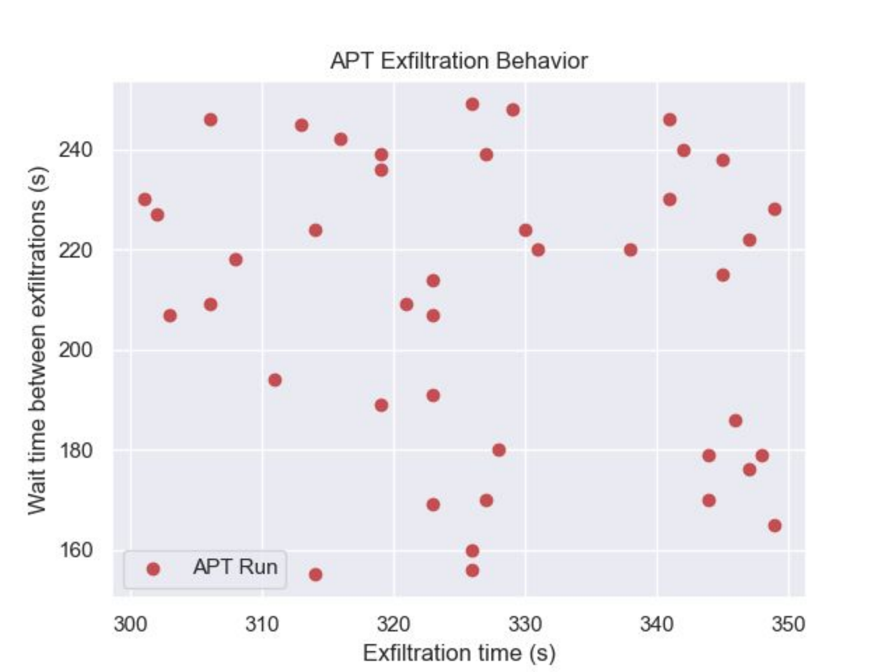
Introduction

- The VarIoT-gateway connects an Air Quality Sensor and a VarIoT server for remote data sharing
- The Air Quality Sensor communicates with the VarIoT server once every minute and uses TLS encryption
- Our goal is to deploy malware onto the gateway and detect it using behavioral malware detection
- *We show that while a machine learning model trained with a simple unigram representation of system calls works well for noisier and more disruptive malware, it does not perform as well for stealthier malware*



Malware - Advanced Persistent Threat

- Advanced Persistent Threat (APT)
  - An APT is a type of malware often used for espionage and spying, sometimes by nation-states and other larger organizations
  - In this work, the APT is designed to copy and exfiltrate the contents of files to a user-specified remote host
  - A C&C server initiates and supervises the data exfiltration
  - It is randomized in terms of its exfiltration behavior, which is shown in the figure to the right

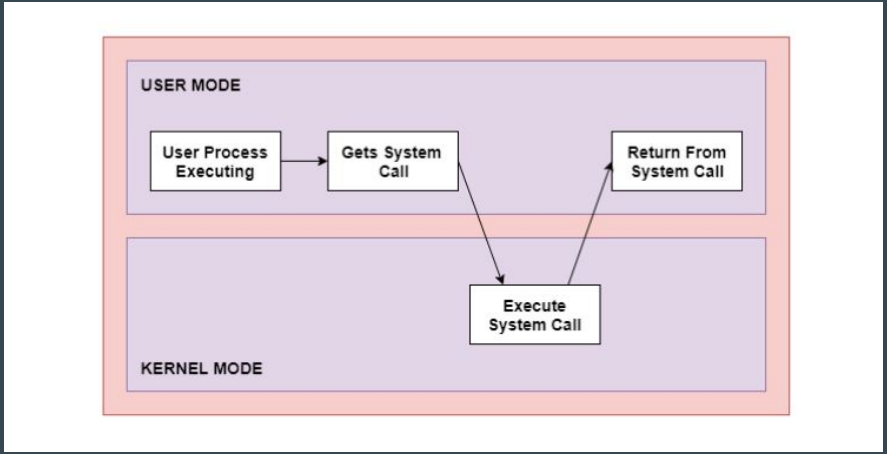


Malware - Denial of Service using Netwox

- Denial of Service (DoS)
  - The DoS is a simpler type of malware that seeks to make a host inoperative by overloading the host with packets
  - In this work, the DoS malware uses a TCP Reset Attack to sever the connection between the IoT device and the VarIoT server
    - A TCP Reset Attack listens to an ongoing TCP connection and then sends a spoofed packet with the “R” flag set to the victim, which will terminate the TCP connection
- **Netwox**, a popular network utility, is used for the TCP Reset attack
  - **netwox** is first downloaded onto the gateway by our malware using the standard **apt-get** procedure common on Linux machines
  - It is unpackaged and ready to attack the communication between the Air Quality Sensor and the VarIoT server using the **netwox 78 attack**

Data Collection

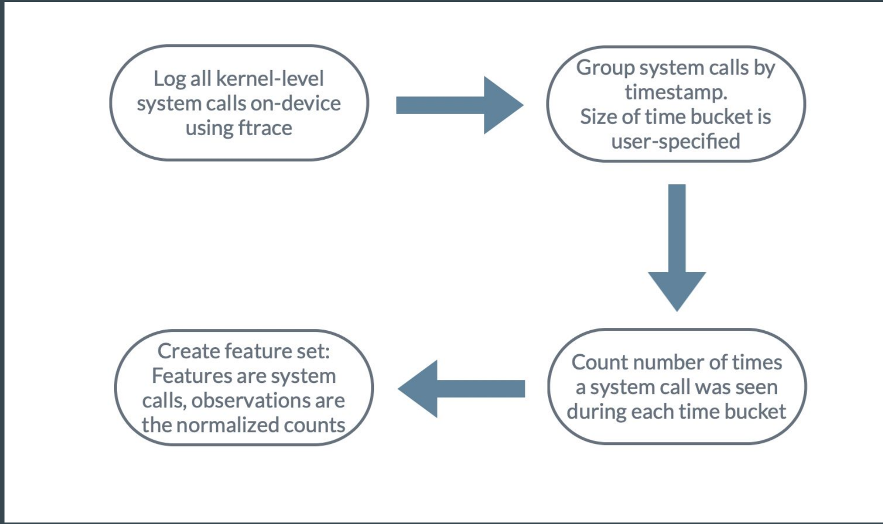
- The raw data consists of system calls executed on the VarIoT-gateway
  - Collected during periods of benign behavior and of malware execution on the device
- Grouped by timestamp using a user-specified window size, which is a parameter that breaks up the total amount of data collection time into a user-specified number of buckets



https://www.tutorialspoint.com/what-are-system-calls-in-operating-system

Data Processing using NLP

- bag-of-*n*-grams approach
  - The feature set is composed of the number of observations of each *n* consecutive system calls in a particular time window
  - A value of $n = 1$ was chosen, which means the feature set consists simply of the number of times each system call was observed during each time window
  - The number of observations of the system calls were then normalized using Term Frequency-Inverse Document Frequency (TF-IDF)

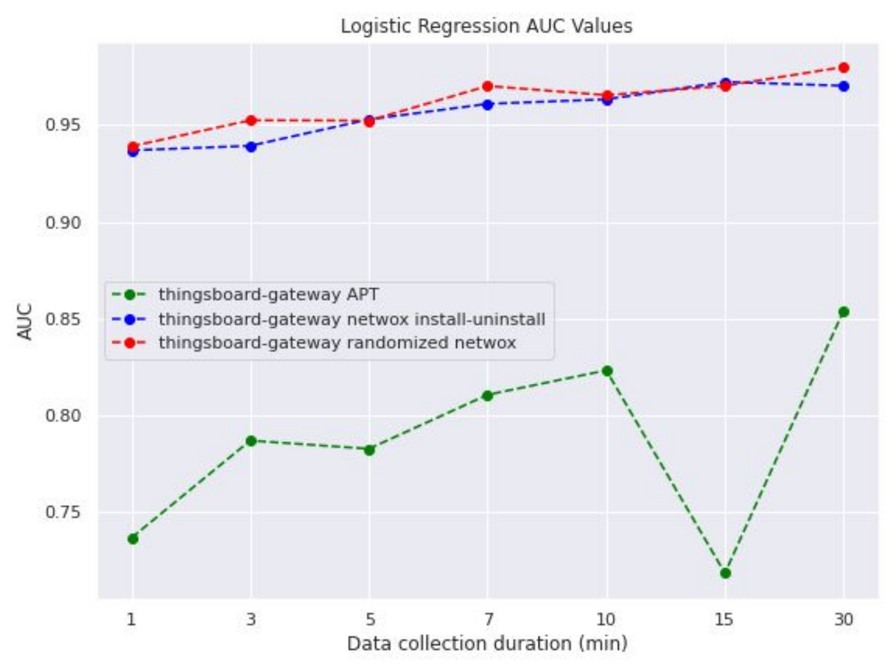


Experimental Results Overview

- Area Under the Receiver Operating Characteristic Curve (AUC) is used to measure the efficacy of the models
- AUC measures a classifier's ability to differentiate between classes in the data and is useful as a summary of the Receiver Operating Characteristic (ROC) curve
- Three specific malware are used for evaluation
  - Stealthy APT malware
  - A simple installation and uninstallation script, which is responsible for repeatedly downloading **netwox**, unpackaging and installing it, and then removing it from the device. *This is useful to show how easily these simple ML models can detect the malware before any execution starts, which is especially important for zero-day attacks*
  - The randomized **netwox**, which not only encompasses the installation/uninstallation process, but also executes the **netwox** TCP Reset Attack for a random duration of time.

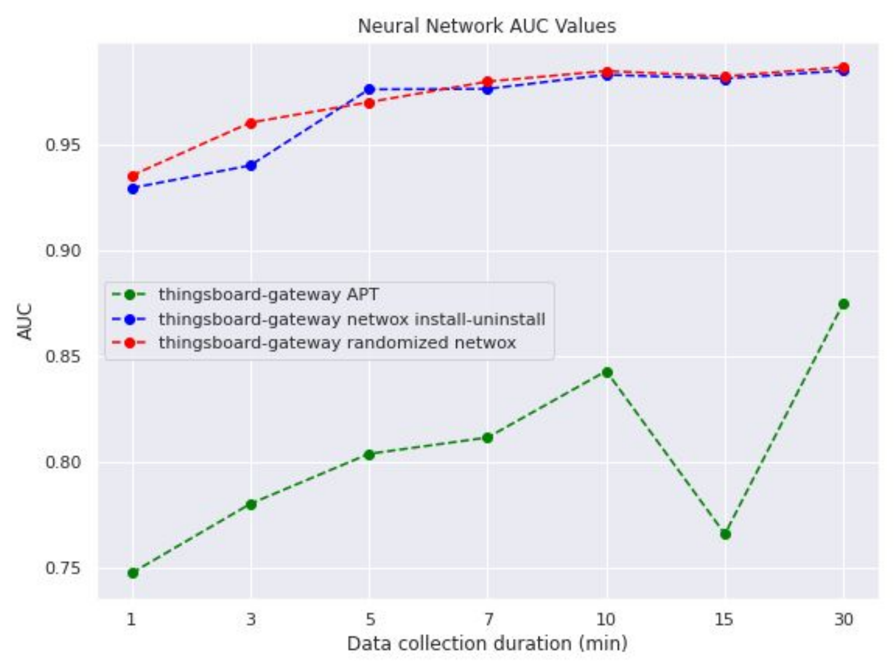
Experimental Results - Logistic Regression

- Logistic Regression is one of the most lightweight, yet effective, machine learning models suitable for our task
  - LR also does not require much data for training, also making it ideal for our problem space
- The results show that the LR model could easily detect the **netwox**-related malware, but struggled more to detect the APT



Experimental Results - Neural Network

- As with the LR model, the NN is easily able to detect the **netwox**-related malware, but again struggled more to detect the APT
- The AUC values for both the LR and NN models follow the same trajectory and have essentially the same values for the **netwox**-related malware, with the only major difference being that the NN had marginally better results for the APT malware



Conclusions

- Two lightweight and efficacious machine learning classifiers were built
  - Both were successful in classifying malware, especially the **netwox**-related malware
- The models can detect the installation/uninstallation malware using only 1 minute of training data with greater than 90% Area Under the Curve
  - A very useful finding for users - if malware can be stopped early before it executes, the user has a chance to prevent malware from damaging their system
- Both models were also able to detect the randomized **netwox** with greater than 90% Area Under the Curve
- The models were significantly less successful in classifying the APT malware
  - The randomization of the APT's behavior, as well as its much smaller system call footprint, make it more difficult to detect using only the lightweight NLP data representations used in this work

Future Work

- Use more advanced NLP techniques, such as Recurrent Neural Networks (RNN), Gated Recurrent Units (GRU), and Long Short-Term Memory (LSTM)
- Use other IoT devices that communicate using a variety of protocols, such as:
  - Bluetooth
  - LoRa
  - ZigBee
  - SigFox
  - UHF RFID
  - mmWave radar
- Expand this research to include UAVs and potentially more types of UAV-focused malware attacks, such as spoofing and jamming on UAV GPS systems and Man-in-the-middle attacks

Extension to UAV

- Attack surface
  - Drone on-board control
  - Raspberry Pi taking photos from the drone
  - Drone remote control
  - Computer processing data from the Raspberry Pi
- Each of these places can be targeted by malware and are possible locations for malware detectors

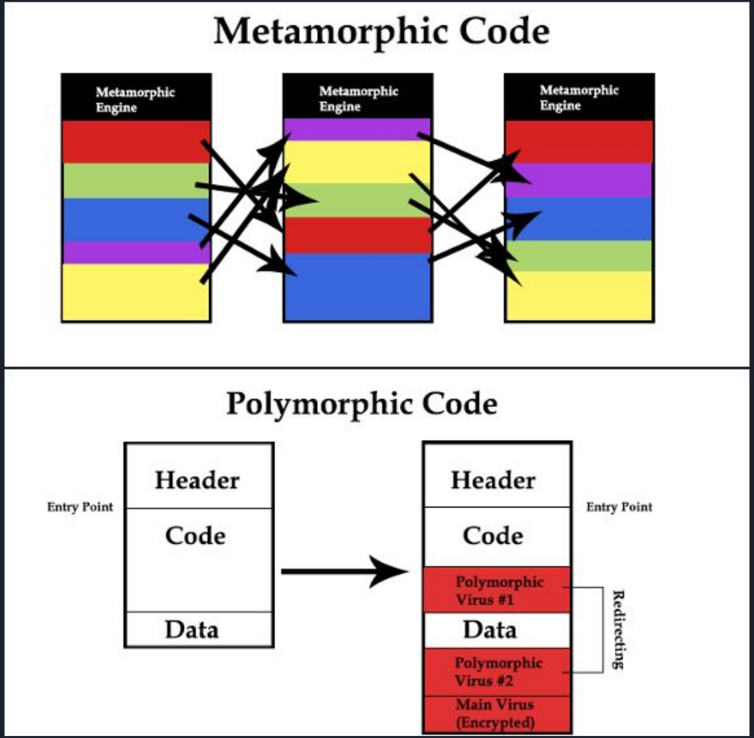
Malware Survey

A58 TIM

Spiros Mancoridis, John Carter
Drexel University

Introduction

- Software has had a net positive impact on society, but a small subset of users impact society negatively with software called “malware”
- Malware can infect any host, including Internet of Things (IoT) devices like Unmanned Aerial Vehicles (UAVs)
- These malware can take different forms
  - Encrypted malware
  - Polymorphic malware
  - Metamorphic malware

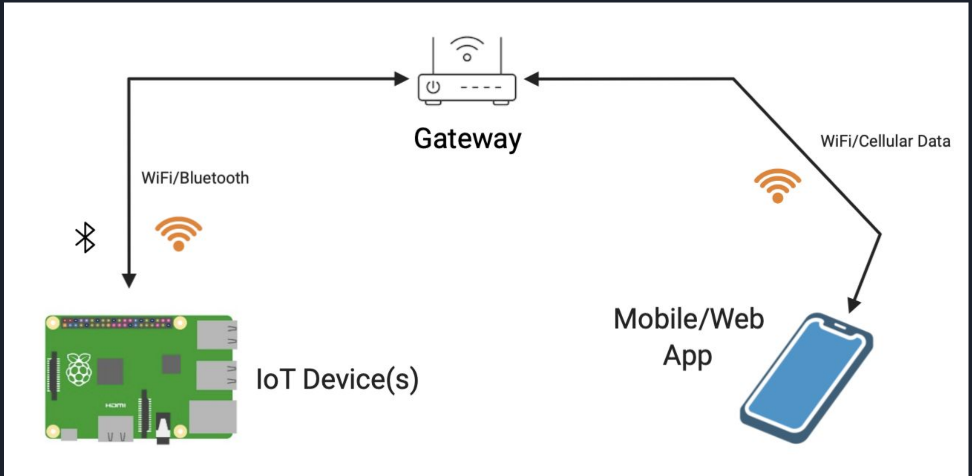


https://www.bizety.com/2016/02/05/deep-learning-neural-nets-are-effective-against-ai-malware/

IoT Ecosystem

The IoT Ecosystem has three main components, but can include four

- Device - hosts such as UAVs, smart thermometers, etc.
- Gateway - the device's access to the outside world
- App - some server with which a user can manipulate the device
- Cloud - optional, but can be used for data storage from the device as well as hosting the user app, etc.



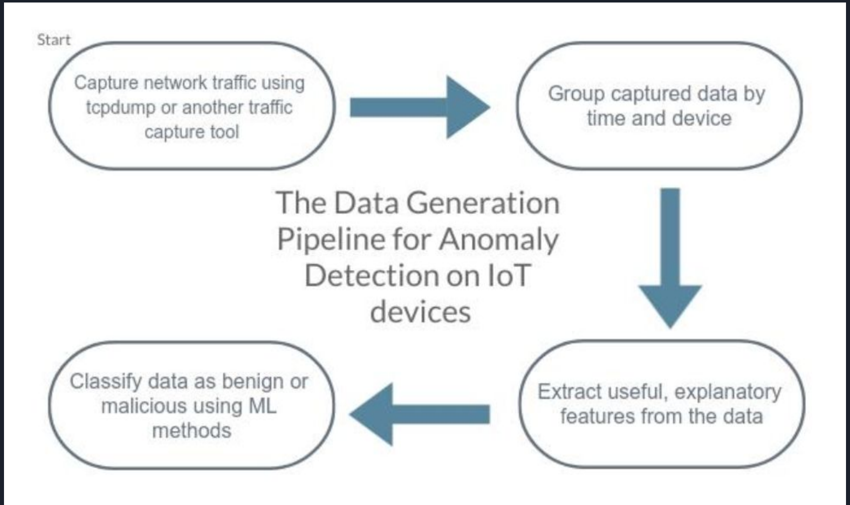
Types of Malware Attacks

Common malware attacks can include:

- Viruses - any malware that spreads between hosts by replicating themselves
- Trojans - attacker places malicious code into a benign application to gain control of the IoT device in order to, for example, exfiltrate data
- DoS attacks - attacker overloads component(s) of IoT device, such as the CPU or memory access, leaving it unable to process requests
- Intrusion - attacker tries to gain control of a shell on the victim via ssh
- Power cut - attacker removes the power source from IoT device
- Overheating - attacker places heat source near victim, causing it to overheat and malfunction

Malware Detection Methods

- With or without machine learning (ML)?
  - Non-ML malware detection uses signature analysis
  - ML uses tools such as honeypots to capture malware data and study its behaviors
- Anomaly detection has become the preferred method due to
  - Limited resource consumption
  - Works well for zero-day attacks

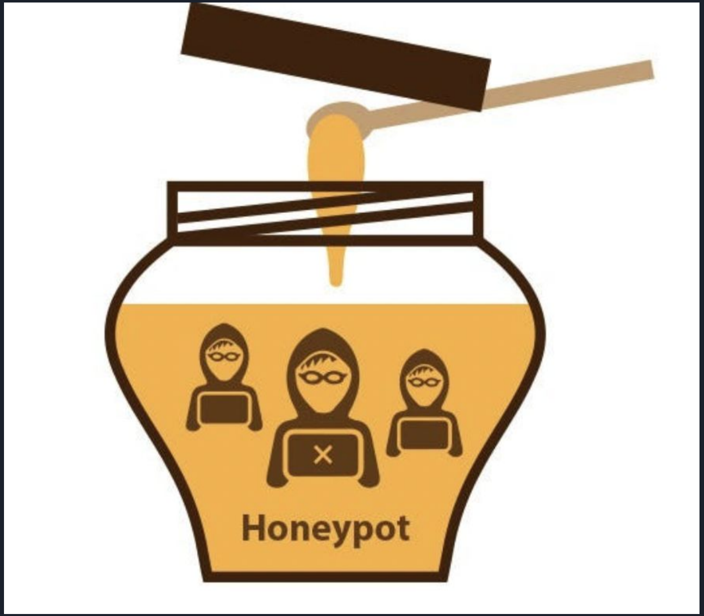


Example ML-based detection pipeline

Data Acquisition

A common way to create malware data organically is to use a honeypot, which lures potential hackers and allows for security researchers to study their code

- Can accomplish this with lax security measures, as well as other things
- Example is IoTPOT, which uses Telnet as its siren
- Another way is to create multiple virtual private network (VPN) tunnels forwarding to an IoT device



https://medium.com/@mr.jchens/how-to-set-up-a-honeypot-in-10-minutes-580e5d990d32

Malware Mitigation Methods

- After detecting malware running on IoT devices, the next step is to mitigate the impact
- Propagation risk is high given the network interconnectivity of IoT devices
- One option is to confine infected nodes, but this is not usually feasible since it often renders the device useless and can be based on a false positive
- Another option is a centralized framework to mitigate the effects of on a larger scale
    - IoT device is connected to a cloud server that collects data related to known IoT vulnerabilities
    - The server maintains vulnerability mitigation policies for known vulnerabilities and exposures (CVEs) of the specific device it protects

Conclusion

- Research in securing UAVs and the networks in which they reside is an important and interesting area of research
- This research is also at the intersection of two interesting and timely areas of research: machine learning and cybersecurity
- Anomaly detection is useful for this task
  - Traditional machine learning models
  - RNNs for sequential data
- Network traffic data passing through the gateway and the system calls being executed by the Linux kernel are useful datasets for malware anomaly detection

FAA Demo

John Carter, Spiros Mancoridis, Malvin Nkomo

21 March 2024

Drexel UNIVERSITY

Introduction

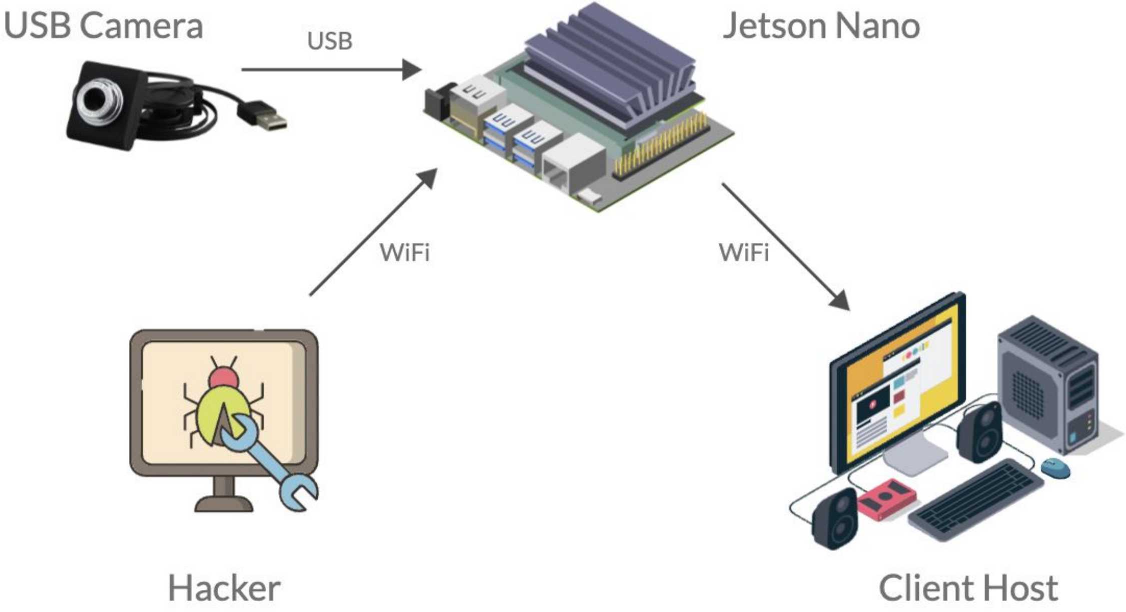
- Many types of attacks can be directed at IoT devices
  - In this work, we focus on video streaming attacks
- Using an IoT ecosystem at Drexel comprised of Jetson nano based edge IoT gateway running a the web server, a client, and a hacker, we show how a video feed can be disabled
- Although the camera is stationary, the attack could be just as easily launched if the camera were streaming video from a UAV using the VarIoT ecosystem that supports multimodal sensors with WiFi, BLE, LoRa and ZigBee
  - The payload is reduced by implementing an edge deployment of the IoT ecosystem
- We show that a video feed can be easily disabled by standard Linux command line networking tools

Drexel UNIVERSITY

Experimental Setup

Four components

- USB Camera
- Jetson Nano - video web server
- Hacker - gains remote access to Jetson Nano and launches attack
- Client - views video stream from Jetson Nano



Drexel UNIVERSITY

Malware Landing

- We assume the malware has already landed for simplicity, but it could have infected the camera server via a remote attack and then download the malicious netwox payload
- Some ways the malware could land on device
    - Weak default passwords for IoT devices and routers
    - Backdoors in downloaded or third-party software
    - Buffer overflow vulnerabilities
    - Race conditions in OS kernel software (like Dirty Cow)
- In this case, the network utility `netwox` is installed via apt-get and then initiates the attack

TCP Reset Attack using `netwox`

- The attack uses the network utility `netwox`
  - `netwox` provides a suite of network utilities, one of which is a TCP Reset Attack
  - Also includes attacks such as syn flooding and tools like `traceroute`
- A TCP Reset attack is a denial-of-service (DoS) attack which floods a server with many bogus packets making it unable to respond to genuine requests
  - In this case, they are TCP packets with the reset flag set to 1
- The attack is run directly on the Jetson Nano by a host that is connected remotely
- While the attack is running, the video web server is inaccessible from the client

Drexel UNIVERSITY


2024-02-29
16:32:06-08

Conclusion

- A video feed was viewed on a client connected to a video web server living on a Jetson Nano
- Malware living on the Jetson Nano was initiated via a remote connection from another host
- The malware executes a TCP Reset Attack against the port from which the web server is streaming the video
- Although this video is from a camera connected to a stationary Jetson Nano, this could easily be a video feed from a UAV in flight

Drexel UNIVERSITY

References

1. https://web.ecs.syr.edu/~wedu/Teaching/cis758/netw522/netwox-doc_html/tools/index.html
2. https://web.ecs.syr.edu/~wedu/Teaching/cis758/netw522/netwox-doc_html/tools/78.html
3. https://nordvpn.com/cybersecurity/glossary/tcp-reset-attack/
4. https://developer.nvidia.com/blog/jetson-nano-ai-computing/
5. https://www.elecbee.com/en-27983-No-Drive-Mini-USB-Camera-For-Raspberry-Pi

Drexel UNIVERSITY

ASSURE A58
Black hole network attack

October 17, 2024

Steven Weber

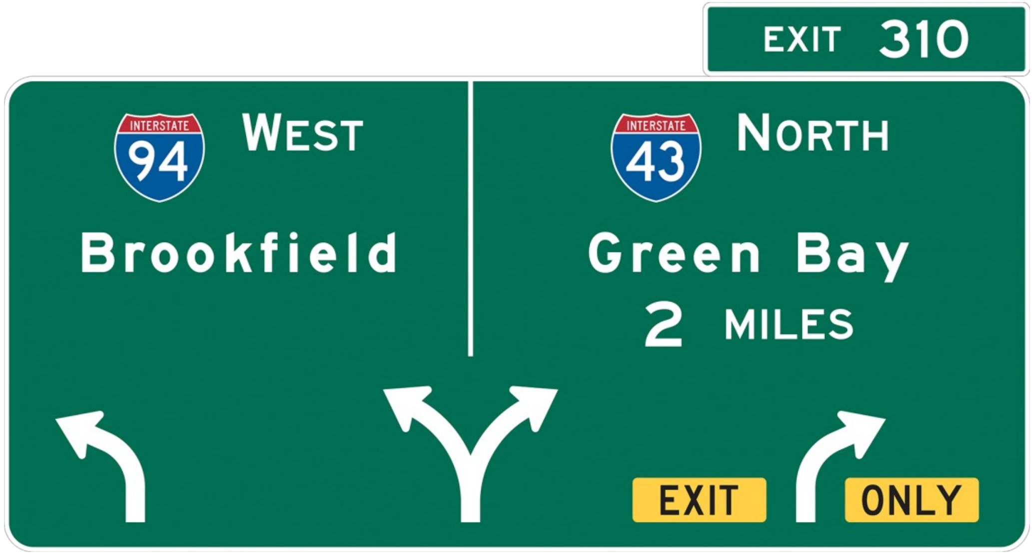
Drexel University

Outline

1. Routing protocols
2. Black hole routing attack
3. Black hole routing attack for UAS

Routing protocols

At each location in the network, the routing protocol should provide direction as to the next stop on the lowest cost route to each possible destination.



Network abstraction as a directed graph

- Treat all computing resources (computers, routers, switches, relays, etc.) as vertices.
- Treat every direct communication link between resources as a directed edge.
- Ensure each vertex knows the next hop on the lowest cost path to each destination (e.g., 1 knows that 3 is the best way to get to 4)

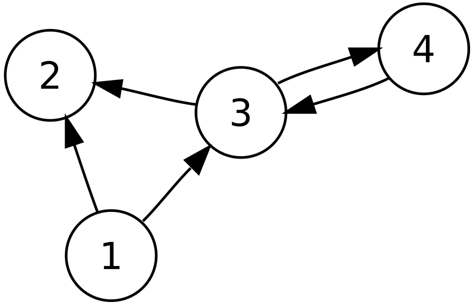
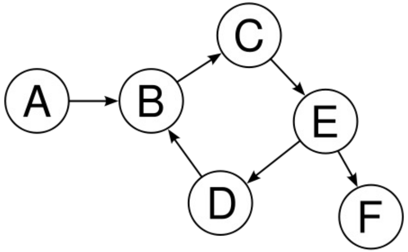


Figure from https://en.wikipedia.org/wiki/Directed_graph

Handling multiple paths by computing cost-to-go

- There are often multiple paths to a destination: how does B decide whether to use relay C vs. D in getting to E?
- Selecting between multiple routing options requires a cost measure; there are many possible criteria (we will focus on the simplest: hop count)

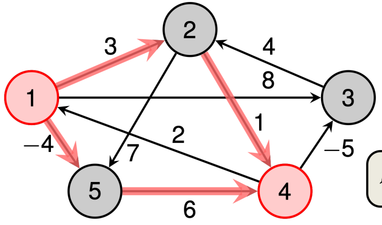


https://computersciencewiki.org/index.php/The_web_as_a_directed_graph

All-pairs shortest path (e.g., Floyd-Warshall)

- Routing requires solving the "all pairs shortest path" (ASPS) problem in a distributed manner.
- Floyd Warshall algorithm solves ASPS, leverages the principle of dynamic programming.

Example of Shortest Path via Matrix Multiplication (Figure 25.1)



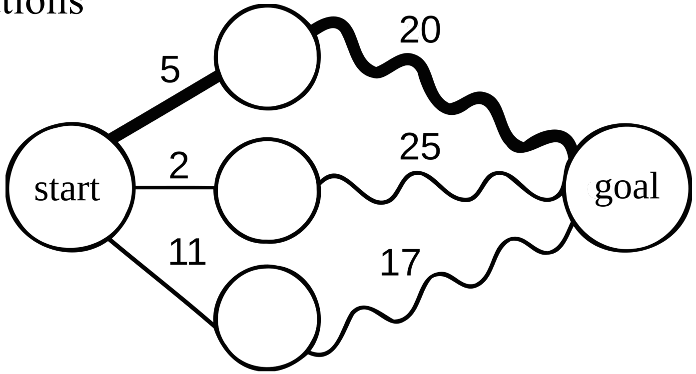
$$\ell_{1,4}^{(2)} = \min\{0+\infty, 3+1, 8+\infty, \infty+0, -4+6\}$$

$$L^{(1)} = W = \left(\begin{array}{ccc|c|c} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{array}\right) \qquad L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \boxed{?} & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

https://www.cl.cam.ac.uk/teaching/1516/Algorithms/apsp.pdf

Dynamic programming recursion

- Dynamic programming is the basis for most algorithms that find lowest cost routes
- DP relies on program decomposition, instantiated using the Bellman optimality equation (not shown here)
- Routing protocols exchange messages regarding cost-to-go to compute solutions



https://en.wikipedia.org/wiki/Dynamic_programming

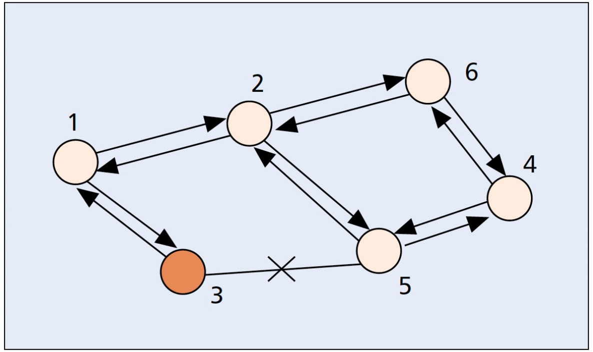
Outline

1. Routing protocols
2. Black hole routing attack
3. Black hole routing attack for UAS

Black hole routing attack

Vertex 3 is malicious:

- it may report lower than actual "costs to go" to its neighbors in order to attract more traffic its way (e.g., 1 chooses 3 instead of 2 to get to 4)
- it may fail to relay traffic sent to it (e.g., 3 doesn't relay 1's traffic)



Hongmei Deng, Wei Li, and Dharma P. Agrawal, "Routing Security in Wireless Ad Hoc Networks" *IEEE Communications Magazine,* vol. 40, no. 10, pp. 70-75, October 2002 https://ieeexplore.ieee.org/document/1039859

Outline

1. Routing protocols
2. Black hole routing attack
3. Black hole routing attack for UAS

Black hole routing attack for UAS

SBHA: An undetectable black hole attack on UANET in the sky

Runqun Xiong[1] | Lan Xiong[2] | Feng Shan[1] | Junzhou Luo[1]

[1]School of Computer Science and Engineering, Southeast University, Nanjing, China

[2]School of Cyber Science and Engineering, Southeast University, Nanjing, China

Correspondence
Runqun Xiong, School of Computer Science and Engineering, Southeast University, Nanjing, China.
Email: rxiong@seu.edu.cn
Lan Xiong, School of Cyber Science and Engineering, Southeast University, Nanjing, China.
Email: lxiong@seu.edu.cn

Funding information
National Natural Science Foundation of China, Grant/Award Numbers: 62172091, 61602112, 61632008; Jiangsu Provincial Key Laboratory of Network and Information Security, Grant/Award Number: BM2003201; Key Laboratory of Computer Network and Information Integration of the Ministry of Education of China, Grant/Award Number: 93K-9; International S&T Cooperation Program of China, Grant/Award Number: 2015DFA10490; Collaborative Innovation Center of Novel Software Technology and Industrialization and the Collaborative Innovation Center of Wireless Communications Technology

Summary

With their high flexibility and versatility, unmanned aerial vehicles (UAVs) have maneuvered their way into many applications. Thanks to their ability to plan and coordinate, multiple UAVs complete tasks more effectively, which boosts their popularity in battlefield surveys, formation performances, and targeted searches. However, the risk of security threats also rises alongside their popularity. The UAV ad hoc network (UANET) has endeavored to contend with such risks through the optimized link state routing (OLSR) protocol. To test the security and strength of this effort, we present a sky black hole attack (SBHA) algorithm for OLSR, which is undetectable, based on the UANET's multi-hop routing and the OLSR's known topology. This algorithm obtains the network's maximum profits by approaching and then replacing the calculated topology center and traffic center in UANET. Because of the ever-changing topology, SBHA aims at UANET's single central node that cannot be detected in advance. This attack is difficult to detect by UANET and therefore difficult to defend. The simulation results show that SBHA can cause greater damage to UANET compared to a traditional black hole attack, and ordinary defense algorithms cannot reduce the negative impact of SBHA on UANET. In addition, SBHA also gains UANET control, and leads to drastic changes in UAVs' movement trajectory, which has more intuitive effects.

KEYWORDS
black hole attack, NS-3, OLSR, UAV ad hoc network

Runqun Xiong, Lan Xiong, Feng Shan, and Junzhou Luo, "SBHA: An undetectable black hole attack on UANET in the sky," *Wiley Concurrency and Computation: Practice and Experience,* vol. 35, no. 13, 2021 https://onlinelibrary.wiley.com/doi/10.1002/cpe.6700

Optimized Link State Routing Protocol (OLSR)

- Optimized Link State Routing (OLSR): a routing protocol designed for mobile wireless networks (such as networks involving UAS)
- OLSR maintains a set of multipoint relays (MPR) which relay messages between nodes, and are used in computing routes
- By positioning itself correctly, a malicious UAS can ensure that it will be selected as an MPR by the MPR selection protocol
- In fact, the Sky Black Hole Attack (SBHA) replaces the Central Node:

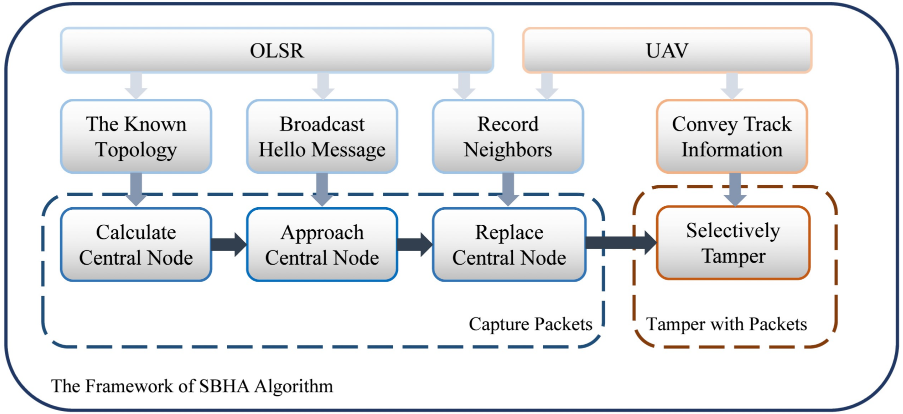
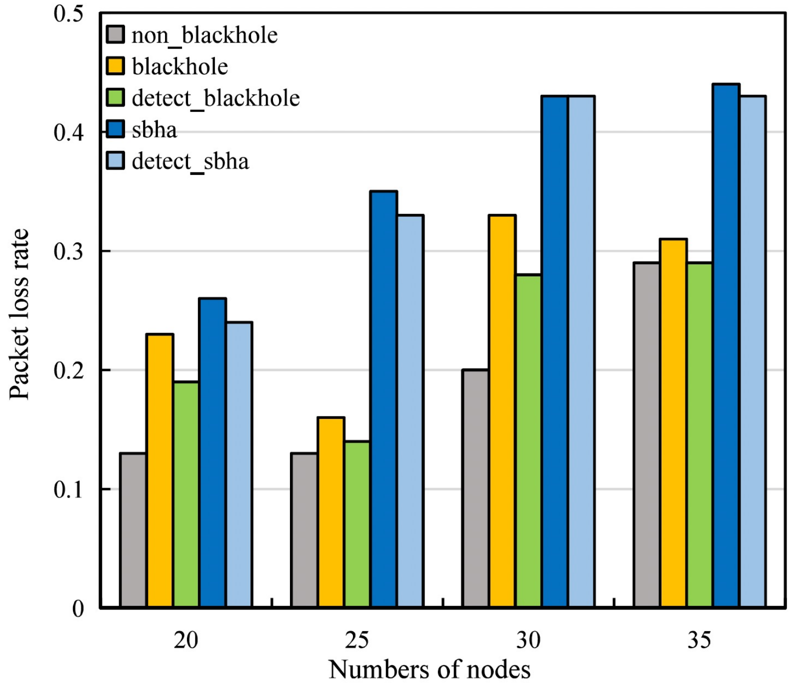


FIGURE 1 Sky black hole attack (SBHA) algorithm framework. OLSR stands for optimized link state routing protocol and UAV stands for unmanned area vehicle

Representative simulation results

SBHA significantly increases the packet loss rate relative to a "normal" black hole attack.



IoT Malware Survey

John Carter and Spiros Mancoridis

Department of Computer Science, Drexel University

19 April 2022

1 Introduction

Over the past few decades, software has evolved from being an obscure tool used by few, to a ubiquitous tool used by virtually everyone. While software has had a net positive impact on society, a small subset of users use it to impact society negatively. The software they write, called "malware," is costly and difficult to detect and mitigate. The malware infects any host they manage to infect, including Internet of Things (IoT) devices.

The Internet of Things can be described as a network consisting of "smart objects," which are everyday items with Internet connectivity embedded into them to give them remote data sharing capabilities [1]. The number of active IoT devices has risen sharply during the past decade, and as a result, their security is very important. Many devices perform essential tasks that need to be running continuously and uninterrupted, such as security cameras, home locks, heart monitoring devices, and even Unmanned Aerial Vehicles (UAVs). Such devices are the potential targets of malware, as are the components that help power them: gateway devices and the cloud.

There are several common IoT attack models, such as Denial-of-Service or Distributed Denial-of-Service (DoS/DDoS) attacks, jamming, and spoofing [26]. DoS or Distributed DoS attacks refer to when attackers flood the target server (with which the IoT device communicates) with bogus requests, leaving the server unable to fulfill the requests of the IoT device. Jamming refers to when attackers send fake signals to interrupt ongoing communication between the device and the server(s) with whom the device is communicating. This results in a depletion of the device's resources, such as power and/or bandwidth. Lastly, spoofing refers to when attackers impersonate a genuine IoT device to gain unauthorized access to an IoT system in hopes of launching another attack once inside, perhaps a DDoS attack.

Detecting malware attacks can be difficult. For instance, an attacker could embed malware into trusted applications and/or could send malware over protocols that are traditionally allowed by firewalls and access lists [22]. Another problem is that attackers can try to obfuscate their malware or encrypt it, which presents further challenges for someone trying to figure out what is happening on their network [22]. Scale tends to exacerbate these problems. Because of this, an organization with more hosts on the network will generate more network traffic, thus making it even more difficult to manually or automatically scrutinize the large amounts of data [22].

New methods for obfuscating malware have emerged, built on previous methods to make their detection more difficult. One of the first methods used to try to circumvent traditional anti-virus software was *encrypted malware*. Encrypted malware makes detection more difficult because they make the bit sequence of the malware binary different than all of the bit sequences in the malware signature databases created by anti-virus companies. Another way that adversaries can make malware less detectable is by creating *polymorphic malware*, which alter the decryption code each time a copy of the malware is created. This succeeds in making the detection process more difficult, but not impossible, because once the code is decrypted, the malware can be analyzed in the computer's local memory. Once adversaries found that polymorphic malware was not the best solution, a new idea emerged: *metamorphic malware*. Metamorphic malware modify all of their malware code, rather than only the decryption code, every time the code is copied during the malware propagation process. Metamorphic malware are less prevalent because they are harder to create, but are more alarming due to their ability to bypass anti-virus software.

2 The IoT Ecosystem

The IoT ecosystem is divided into four main components: the app, the router or gateway device, the cloud, and the IoT device. Each of these components are important and need to be functioning correctly for the IoT device to work properly and as expected. Below, is an illustration of these components, and each has a section in this document devoted to their roles in the ecosystem.

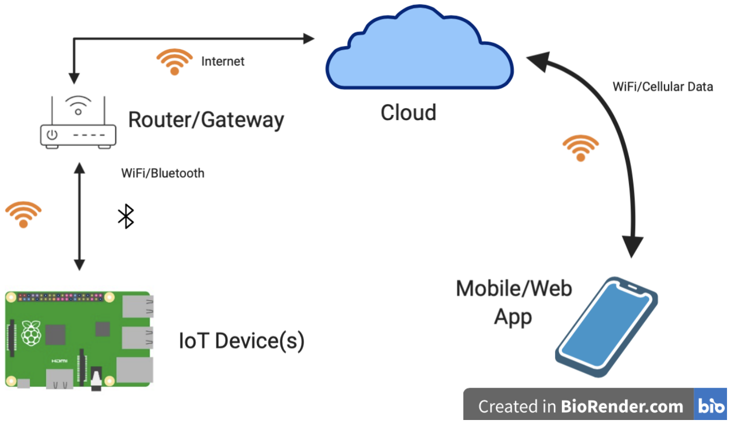


Figure 1: The IoT ecosystem consists of a mobile/web app, an IoT device, a gateway, and the cloud.

2.1 App

The app is the part of the IoT ecosystem with which a consumer interacts. Usually either through a web app or a mobile app, these apps are where users can configure their device, change the settings of the IoT device (such as the temperature on a Smart thermometer), and get the information the IoT device is there to collect (*i.e.,* "what is the current temperature of my home?").

The most likely and possible attack vector for malware to manifest itself in apps is through permissions granted to an app by an operating system. For example, the Android mobile operating system (OS) has malware due to malicious apps that exploit excessive permissions for certain apps that are available for download [13]. Each app running on the Android OS must declare the permissions it requires to run, which provide access to device functions such as "`INTERNET`" or "`SMS_RECEIVED`" [13]. Attackers can create malicious apps that declare the permissions they need to run, and thus are granted unneeded access to data such as text messages received or Internet activity. These apps could potentially be used to control IoT devices, such as an app that provides the live camera feed of an IP camera.

However, in terms of the IoT ecosystem, this component is less likely than the others to be hacked or targeted for a number of reasons. The first reason is that the app is likely on a device that the consumer uses regularly, and thus monitors frequently. If something suspicious or malicious is happening on the app, the user is much more likely to spot it rather than something suspicious happening on a device that is likely not near them. The second reason is related to the first. Since the app is probably either on a user's laptop or mobile device, it's more likely to be patched and kept up-to-date. If the app is running on a phone or tablet, this is probably even more likely, as far fewer malware are able to infect a cell phone in contrast with an IoT device running an outdated version of Linux. The user probably will have changed the default passwords and credentials on these devices as well, which is not commonly done on IoT devices. While the app is an important component in any IoT network, for the aforementioned reasons, this survey focuses more on the other three components of the IoT ecosystem.

2.2 Router/Gateway Device

The router is an essential part of the IoT ecosystem, as it allows for the IoT device and the user to be connected. The information exchanged between the device and the user is at the mercy of the information the router allows to be exchanged. As a result, there is an increasing amount of malware targeting the router. One example is when infected routers are recruited to be part of DDoS attacks, similar to IoT devices being recruited for the same purpose [4].

A router-specific example is malware that forces the router to drop certain packets, making communication difficult or impossible. Often this is accomplished by requesting a packet resubmission when the packet has already been submitted successfully. This action can harm the IoT device's battery life, and diminish the network's throughput and increase its delay time [24]. One way to combat this issue is to secure the gateway device or access point (AP), which will then ensure that the communication flowing through it is unhampered. To accomplish this, for example, one may implement an intrusion detection system (IDS) on the access point, and let the IDS decide whether the access point is infected or not, as described in [24]. The IDS first keeps track of the number of packets flowing through the IoT device, which includes the packets sent as a result of a NACK (no-acknowledgement) packet sent from the gateway. The access point keeps track of the number of uplink packets successfully received from the gateway. In addition, each IoT device updates the AP regarding the number of packets sent through the non-main channel to the AP at a regular time interval. In prior work, a higher time interval T yields a more accurate classification of the anomaly with a higher probability [24]. This method proved useful for determining if there is an adversary corrupting a communication between an IoT device and an access point by way of an infected gateway device.

Securing the router connecting the IoT ecosystem is imperative, since without the router the ecosystem is useless due to none of the devices being able to communicate.

2.3 The Cloud

The cloud usually consists of storage on servers belonging to a third party, such as Amazon Web Services (AWS), where data is stored. For example, perhaps a user has a Raspberry Pi with a web camera attached acting as a security camera. The Raspberry Pi can transmit the feed to the user, but also to the cloud to save a record of the video data. Although the cloud has malware concerns of its own, usually these issues are monitored by their proprietors, such as Amazon, and are out of the scope of this document. The important aspect of the cloud that pertains to this research are the data retention policies employed by the cloud. In other words, users want to know (and have control over) what data is stored in the cloud, and for how long. The data retention policy will answer these questions and outline the data to keep or delete based on the amount of time it has been available in the cloud [12]. With this comes the problem of proof-of-deletion, which is basically the guarantee to the user that the cloud no longer has access to the data and that is has been permanently and irrecoverably deleted.

2.3.1 Docker Hub

A related issue is the security of reusable Docker Hub images. Docker containers have become popular alternatives to traditional virtual machines over the past few years to use applications shared over physical hosts [19]. Because of this, a registry called Docker Hub was created, which acts as a type of cloud application where users can upload and download Docker images. This registry shares both official and community images to users. Official images are public and certified by vendors, such as Oracle or Red Hat, while community images can be created by any user. The sharing of images between users presents a potential security breach in which a user could inject malware into an image that is then shared with other Docker users without their knowledge of the pre-installed malware. In addition, new images (called child images) can be created from current images (called parent images), which means malware can be embedded in parent images and passed along to numerous child images.

Another reason to be alarmed about possible vulnerabilities with Docker is that it, by default, runs with root privileges [25]. More than 350,000 images were analyzed in current research, and over 180 vulnerabilities were found on average in the images [19]. This research also exposed that the vulnerabilities found in the images often propagated from parent images to child images, similar to how malware are spread in other types of attacks [19].

Docker provides a way to certify images by running their `inspectDockerImage` tool, which minimally checks user-created images for adherence to some basic *best practices* and rules. However, work by Wist *et al.* showed that over 80% of certified images contain at least one critical vulnerability [25]. While there is some mechanism for certifying Docker images, as shown, the current way is certainly not comprehensive. Using machine learning anomaly detection could be a useful avenue of research to explore, as more needs to be done to guarantee the security of images downloaded from Docker Hub.

2.4 Device

The IoT device itself is a very important aspect of the ecosystem, and is often the target of malware. These devices take many forms, and can be anything from a wind meter to a refrigerator to a driving assistant in a car or a UAV.

2.4.1 Raspberry Pi

One device that is especially useful in IoT malware detection research is a Raspberry Pi. Raspberry Pi's are small, single-board computers that run a Linux distribution, often the Debian-based Raspbian, as well as other Linux distributions such as Ubuntu. The Raspberry Pi is desirable as a testbed for IoT research primarily for its ease of use and its use of the Linux kernel, as well as its ability to act as many different IoT devices, limited only by the users' configuration. For instance, a Raspberry Pi could be connected to a webcam and become an IP camera that is able to communicate with other hosts via `ssh`, or it could run downloadable Amazon Alexa software and become a customized AlexaPi [2]. Likewise, Raspberry Pi's can also be used for photography, surveillance and other tasks when connected to a UAV [17]. As such, many different IoT ecosystems can be created simply by changing the configuration of this one device.

2.4.2 IP Camera

A common type of IoT device that is the target of malware is an IP camera, which can be used for tasks such as security or surveillance. In these areas, their security is essential, as well as a guarantee of data integrity. If, for example, an IP camera in a bank is compromised by a looping attack, the camera could capture an actual video feed, and play back this old video recording when the bank is being robbed. Furthermore, any IoT device is susceptible to malware, and while some may be deemed more important than others, any device can be recruited to take part in a Distributed Denial-of-Service (DDoS) attack, or other type of coordinated attack.

2.4.3 Unmanned Aerial Vehicle

Another increasingly common IoT device is an Unmanned Aerial Vehicle (UAV). Originally used in military operations, UAVs have become popular for commercial and personal tasks as well due to the decreasing costs to own and operate them as well as their recent technological improvements [10]. They are often used for tasks in agriculture, commercial delivery, media applications, border control, search and rescue, *et cetera.* [15] [16] [17]. Since UAVs have grown in popularity, the interest in attacking them has grown proportionally. The attacks are often focused on the GPS systems guiding the UAVs as well as the data and communications streams between the UAV and the user [7]. Attacks on the GPS systems can include spoofing and jamming attacks, while the possible threat vectors can include errors in configuring communication, sensor, and system settings [7] [16]. It has also been shown that some UAVs are susceptible to man-in-the-middle attacks because of weak Internet security and other vulnerabilities [15]. Lastly, like many IoT devices, UAVs can also fall victim to DoS attacks [16]. A UAV is simply a specialized IoT device, so many of the attacks lodged against a typical IoT device are similarly used against UAVs as well. Since UAVs often perform critical tasks, the security of these devices is extremely important. As their popularity and use continues to grow, so will their vulnerability.

IoT devices can take on many forms, and attacks on these devices can likewise vary. Whether IoT devices are attacked using DDoS or physical attacks, these devices should be set up to withstand a variety of attacks from adversaries. The variety of known attacks will be explained more in Section 3 of this document.

3 Types of Attacks

Attacks on IoT devices are diverse, but usually fall into two broad categories: physical and virtual. Examples of physical attacks include overheating, which involves placing a heat source in close proximity to the victim device in order to overheat it, as well as cutting off power to the IoT device. Virtual attacks are attacks emanating from another computing device, and include attacks such as malware. Research by Shi *et al.* identified six different types of attacks on IoT devices [18], and the list of six is far from comprehensive:

1. Viruses - any malware that spreads between hosts by replicating themselves.
2. DoS attacks - attacker overloads component(s) of IoT device, such as the CPU or memory access, leaving it unable to process requests.
3. Trojans - attacker places malicious code into a benign application to gain control of the IoT device in order to, for example, exfiltrate data.
4. Intrusion - attacker tries to gain control of a shell on the victim via `ssh`. An example of this is a Remote Access Trojan (RAT).
5. Power cut - attacker removes the power source from IoT device.
6. Overheating - attacker places a heat source near the victim, causing it to overheat and malfunction.

The main idea presented in a paper by Shi *et al.* to detect this diverse group of attacks is to use energy consumption as a metric to determine whether or not a device is infected [18]. This is to overcome the problem of not being able to trust a (potentially) infected device after it has been compromised by an adversary. It also provides a way to detect both physical and virtual attacks.

Perhaps the most common attack on IoT devices is a DoS/DDoS attack. In a DDoS attack, malware takes over a device and is recruited to be part of a botnet and connects to other malicious IoT devices [22]. A botnet can be described as a group of connected computers recruited to take part in a coordinated task [22]. Once infected, the IoT device may behave normally for a time, but will eventually be used for a malicious purpose: disabling a targeted website or service, for example. One essential part of a DDoS attack is IP spoofing, which is the act of forging the sender's address in the IP header [8]. Specifically, spoofing is used in Volumetric and Reflector DDoS attacks. Volumetric attacks send a large volume of packets to a target. Reflector attacks involve spoofing the IP address of the victim in service requests sent to other servers [8]. The servers then respond to the victim device instead of the desired destination and flood the IoT device. After the victim is flooded with packet data, it may not be able to respond to legitimate requests due to insufficient bandwidth.

4 Malware Data Acquisition

One of the main ways to collect data for experimentation in IoT malware detection is to create a honeypot. This acts to lure would-be hackers in order to get their malware code and study it. Often this is accomplished by exploiting lax security on a device, such as using default passwords and ports left open unintentionally. Once the device is attacked, the owners of the honeypot are able to study and replicate the code, thus learning more about the malware targeting their devices. As a result, malware detection and mitigation software can be developed through reverse-engineering the captured malware sample. Since it is now known how the malware infects the device, all that needs to be done is prevent that method from working again. Unfortunately, the problem with this approach is that the creators of the malware will continue to find new ways to infect devices. However, there are instances where one of the families of IoT malware is found, such as Mirai, and thus gives us insight into other kinds of malware due to the similarities between different malware variants.

A honeypot specifically designed for IoT-related malware is IoTPOT, launched in 2015, which emulates Telnet services of various IoT devices to attract new viruses that use Telnet [11]. According to their research, the most commonly attacked IoT devices are DVRs, IP cameras, and routers. The IoTPOT architecture has a few components, the most important of which is the Frontend Responder, which is responsible for emulating different IoT devices by handling incoming TCP connection requests, banner instructions, authentication, and command interactions. It then sends these commands to the IoTBOX backend, which is a set of sandbox environments running different Linux configurations. IoTBOX determines the response to the command request, and forwards it back to the Frontend Responder, which then forwards it to the client. The Profiler, a second component, parses commands between the Frontend Responder and IoTBOX and saves them for later use to reduce the need to communicate with IoTBOX (also subjecting it to fewer malware). The third component is the Downloader, which examines the interactions for download triggers of remote files, such as malware binaries or files obtained from running `wget, ftp,` *et cetera.* The fourth component is the Manager, which handles configuration of IoTPOT, such as connecting IP addresses with device profiles. During 39 days of data gathering, over 70,000 hosts visited the honeypot [11]. There were three typical stages of attacks:

1. Intrusion - login attempts, in which adversaries try to log into the honeypot to gain access to the device.

2. Infection - discover and change the environment to enable downloading malware. Usually these activities are automated.

3. Monetization - a command and control (C&C) server is used to control the device and perform the malicious activities, such as a DoS attack or bitcoin

mining. The attacker is now able to use the newly recruited device for malicious activity.

Many of the attacks observed when IoTPOT was running were coordinated, in that one compromised host would infiltrate the victim and find out its login credentials and CPU architecture, and then send that information to other hosts so they can attack the victim as well [11]. Most of the attacks observed were UDP floods and different types of TCP floods, which is a type of Denial-of-Service attack in which the attacker overwhelms the target's ports with IP packets containing large datagrams. DNS and SSL attacks were also observed [11].

Another similar project, proposed in the CODASPY 2019 proceedings, increased the chances of their honeypot being attacked by creating multiple virtual private network (VPN) tunnels forwarding to an IoT device [23]. The usage of a real IoT device lends credibility to the honeypot, and by leaving it completely exposed to hackers, increases the chances of it being attacked. The key to this type of honeypot is to restrict all outside information to the honeypots, such as surrounding WiFi networks, and to set up a firewall to prevent the malware from propagating further on the network [23] [11].

In the last five years, a large amount of data has been collected from these various research projects, especially the IoTPOT project. Although it was conducted in 2015, that project continues to inspire others in the IoT security field and provides a guide for collecting data. While this data can be used to build more robust malware detection systems, there are still areas in which this data is not comprehensive. For instance, collecting more data in the area of securing the routers and gateway devices that connect the IoT devices, and not just in the securing of the IoT devices themselves, is a useful research avenue to explore. Most of the current data available focuses on securing the IoT devices themselves, without much thought given to securing the routers that connect them to the Internet.

5 IoT Malware Detection Methods

Malware detection can generally be approached in two ways: with and without the use of machine learning. Non-machine learning malware detection uses signature analysis. Static analysis often reviews the language and syntax structure while dynamic analysis uses tools such as honeypots to capture malware and study its behaviors [9]. Historically, most malware detection has been signature-based. This method works well on personal computers, but does not work as well on IoT devices, for a variety of reasons. Perhaps the most important reason is that IoT devices constantly contend with a scarcity of resources, as well as a lack of protection against metamorphic malware [3]. This includes memory as well as computing and electrical power. Because of these reasons, machine learning and deep learning methods have become useful due to their high detection rate and low resource consumption. Deep Learning uses a lot of resources to train the model, but uses relatively few resources to detect malware after the model has been trained. Traditional machine learning techniques include

Support Vector Machines, Logistic Regression, *et cetera,* while deep learning models are usually Artificial Neural Networks (ANNs) or their specializations such as deep ANNs, which include Convolutional Neural Networks. Recently, using Convolutional Neural Networks for anomaly detection has become more common. This process uses gray-scale images of binary files for malware classification. This topic is described in Section 5.2.

5.1 Anomaly Detection using Machine Learning

Recently, anomaly detection has become the preferred method for detecting malware on IoT devices due to its limited resource consumption and flexibility. It also has proven successful because of the limited and predictable behavior of IoT devices. IoT devices are usually set up to complete a few specific tasks, and because of this, they often communicate with a limited number of external servers, and their resultant network traffic behavior and execution behavior (via, for example, system calls) is predictable [5]. Anomaly detection also works well for zero-day malware attacks, since anomalies in kernel and network behavior can be detected almost instantaneously. The general anomaly detection pipeline consists of four main steps when collecting captured network traffic data:

1. Traffic capture - record metadata such as the timestamp, protocol, source IP and port, destination IP and port, packet size, and contents. `tcpdump` is useful for recording this data and saving it in a `pcap` file.

2. Group packets by device and time - separated by source IP, then divided into non-overlapping time windows.

3. Feature extraction - determine the most useful metadata to explain the data, such as the destination IP.

4. Binary classification - using ML methods such as Artificial Neural Networks (ANNs), SVMs, KNN, random forests, decision trees, *et cetera*, to classify data points as benign or malicious.

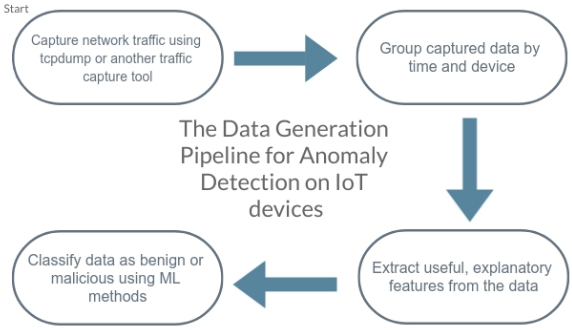


Figure 2: The Data Generation Process used for Anomaly Detection with Network Traffic data

This type of malware detection works well with the standard IoT ecosystem described above, and has been used by multiple research projects. A common ecosystem consists of a Raspberry Pi acting as a router, an IP camera (also possibly implemented as a Raspberry Pi), and any other IoT devices connected to the router, such as a thermostat or a light. There is some feature engineering that can be done to the collected data, and the features fall into two categories: stateless features and stateful features. Stateless features include packet protocol, size, and inter-packet interval, while stateful features include IP destination address cardinality and novelty, and bandwidth [5]. It has been shown that stateless features outperform stateful features in this type of anomaly detection [5].

The features needed for anomaly detection could also be drawn from system data, consisting of a log of system calls made during the data capturing timeframe. On Linux-based IoT devices, the command `ftrace` can be used to record system call information and create the log file [1]. Capturing the system calls during a period of known benign activity, as well as during a time of known malware execution, could provide insight into any connections between malware running inconspicuously and the system calls executed by the malware. The feature engineering process outlined above would be very similar in this case. In previous work by [1] and [2], a *bag-of-n-grams* approach was used, in which short sequences of system calls during a small period of time are considered. This approach often yields patterns between the system call n-gram sequences that make malware detection easier. In addition, there has also been work done where a combination of both system calls and network traffic data was captured and used together for feature engineering successfully [2]. In fact, it was shown that a malware detector based on combined system call and network traffic data detected malware better than the system call malware detector or network traffic malware detector did individually [2]. These methods can use RNNs and LSTMs as well, because they work on sequential data n-grams.

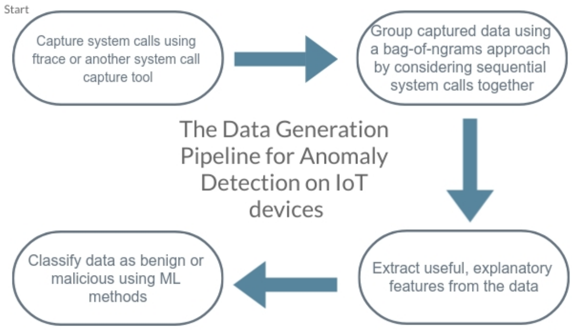


Figure 3: The Data Generation Process used for Anomaly Detection with System Call data

5.2 Image Recognition for Malware Detection

An alternative IoT malware detection method has emerged recently: using a Convolutional Neural Network (CNN) to classify binaries transformed into gray-scale images. Classifying code binaries in the form of images has proven to be successful, at least in a limited data scope. In work conducted by Su *et al.*, malware samples collected from two malware families, Mirai and Linux.Gafgyt, were able to be classified correctly 94% of the time, with a 5% false positive rate [21]. This research used malware samples collected by the IoTPOT honeypot, and were transformed from binaries to gray-scale images by reformatting them into an 8-bit string sequences [21]. A decimal encoding represents the value of a one-channel pixel, which is then formatted into a 64x64 image to be fed into a CNN. Their results indicated that malware images tend to be more dense than benign images [21].

In related work, application binaries are converted into gray-scale images, which are then transformed into sequences of patterns and fed into a Recurrent Neural Network (RNN) [20]. The steps to convert the binary are:

1. Perform raster scanning to find patterns in the image.
2. Use Cosine similarity to distinguish between patterns. The Cosine similarity measures the similarity between two non-zero vectors, and is defined to be the Cosine of the angle between them.
3. Convert the image into a sequence of patterns, and feed the result into a RNN.

This approach yielded the same 94% accuracy rate, but a downside of this approach, as discussed by the authors, is the latency that is involved in the image-based malware detection [20].

Convolutional Neural Networks can be difficult, time-consuming, and resource-consuming to train well enough to classify accurately. One solution to this problem is to upload the binaries to a cloud application with more resources that can perform the classification, and send the results back to the device. If the CNN were running on a large cloud application, it could be trained faster and provide quicker classification results to the IoT device without putting further constraints on the IoT device's resources.

Using image recognition for IoT malware detection is one of the newest fields of research within IoT security, and there are still many problems to mitigate in order to make it a viable solution on actual IoT devices. Training a normal CNN on a small IoT device seems impractical for the foreseeable future due to IoT resource constraints. As a result, a better solution is needed, and provides another avenue of research in IoT malware detection.

6 IoT Malware Mitigation Methods

After detecting malware running on IoT devices, the next step is to mitigate the impact of the malware infection. The risk of malware propagation is especially high in IoT devices, because whenever one device on a network is compromised, it is much easier to continue and infect more devices connected to the network.

One general mitigation idea is to confine the infected nodes and not let the malware spread. The biggest problem with this method, however, is that it often hampers the throughput of the network, thus degrading its performance [14]. This method also presupposes that the malware was detected correctly, which can be difficult considering that malware often try to hide themselves. If the malware successfully decoy themselves, then the confinement method will not be helpful. Similarly, if the detection algorithm produces a false positive, a node will be confined for no reason, which will also likely be detrimental to the network or could cause a denial-of-service. One way to help resolve this problem would be to set a threshold on the amount of throughput required of the network. Given this, the traffic flowing through the infected node can be regulated, and the overall throughput of the network can be tracked. If the traffic restriction results in a throughput that is lower than the required level, the restrictions can be eased until it returns to being above the required level of throughput again [14].

Another method for mitigating malware is a more centralized idea to mitigate the effects of malware on a larger scale, encompassing more than one network. This method connects to a cloud server that collects large amounts of data related to known IoT vulnerabilities. The idea is to connect an "appliance" directly to the IoT device that maintains vulnerability mitigation policies for known common vulnerabilities and exposures (CVEs) of the specific device it protects [6] by connecting to the cloud server and receiving them. Specifically, the security appliance is responsible for three tasks:

1. Communication - receives packets that are addressed to the vulnerable IoT device, processes and forwards them to the device at the discretion of the vulnerability mitigation policy.

2. Mitigation - called by the communication module. This module will have a list of vulnerability mitigation policies to execute.

3. Updater - responsible for receiving updates about newly discovered vulnerability mitigation policies for the IoT device.

The other component of the framework is the cloud-based service. This is responsible for the collection and affiliation of CVEs to specific devices, and for the generation and representation of vulnerability mitigation policies [6]. This framework ensures that IoT devices are up-to-date with recent security updates or patches and prevents exploitation of CVEs. Adopting the framework removes the responsibility of keeping the device up-to-date from the user. It is also efficient in that the security appliance only protects the IoT device from known vulnerabilities that apply directly to the type of IoT device to which it is connected.

While this framework appears to be a useful solution in theory, there are some drawbacks to note. For instance, in work by Hadar *et al.*, a Raspberry Pi 3 is used as the security appliance to communicate with the cloud server that connects to the IoT device [6]. If the IoT device itself is a Raspberry Pi, which is common, the cost of operating the device has at least doubled due to now needing two Raspberry Pi's. There is also the cost to creating and maintaining the cloud service to stay up-to-date with CVEs and be able to communicate with the many types of security appliances.

In summary, while the idea of having a cloud server and security appliance for each IoT device does seem to be efficient and increase the security of the IoT devices, it is likely not feasible because of the increased overhead that all consumers would need to contribute.

7 Conclusion

Research in securing IoT devices and the networks in which they reside is an important, and interesting, area of research. As the number of active IoT devices continues to grow, the importance of their security grows accordingly. This research is also at the intersection of two interesting and timely areas of research: machine learning and cybersecurity.

The use of anomaly detection, either through traditional machine learning models or through RNNs for sequential data, appears to be a viable means for completing this task. Likewise, using both the network traffic data passing through the router, as well as the system calls being executed by the Linux kernel, appear to be the best combination of data for the model to make accurate classifications.

References

[1] N. An, A. Duff, G. Naik, M. Faloutsos, S. Weber, and S. Mancoridis. 2017. Behavioral anomaly detection of malware on home routers. In *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. 47–54. https://doi.org/10.1109/MALWARE.2017.8323956

[2] N. An, A. Duff, M. Noorani, S. Weber, and S. Mancoridis. 2018. Malware Anomaly Detection on Virtual Assistants. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE Computer Society, Los Alamitos, CA, USA, xa;124xa;–xa;131xa;. https://doi.org/10.1109/MALWARE.2018.8659366

[3] Shiven Chawla and Geethapriya Thamilarasu. 2018. Security as a service: real-time intrusion detection in internet of things. *Proceedings of the Fifth Cybersecurity Symposium* (2018).

[4] Ahmad Darki, Alexander Duff, Zhiyun Qian, Gaurav Naik, Spiros Mancoridis, and Michalis Faloutsos. 2016. "Don't Trust Your Router: Detecting Compromised Routers". In *CoNEXT '16*. Irvine, CA, USA.

[5] Rohan Doshi, Noah Apthorpe, and Nick Feamster. 2018. Machine Learning DDoS Detection for Consumer Internet of Things Devices. *2018 IEEE Security and Privacy Workshops (SPW)* (May 2018). https://doi.org/10.1109/spw.2018.00013

[6] Noy Hadar, Shachar Siboni, and Yuval Elovici. 2017. A Lightweight Vulnerability Mitigation Framework for IoT Devices. In *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy* (Dallas, Texas, USA) *(IoTSamp;P '17)*. Association for Computing Machinery, New York, NY, USA, 71–75. https://doi.org/10.1145/3139937.3139944

[7] C. G. Leela Krishna and Robin R. Murphy. 2017. A review on cybersecurity vulnerabilities for unmanned aerial vehicles. In *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. 194–199. https://doi.org/10.1109/SSRR.2017.8088163

[8] Jelena Mirkovic, Erik Kline, and Peter Reiher. 2017. RESECT: Self-Learning Traffic Filters for IP Spoofing Defense. In *Proceedings of the 33rd Annual Computer Security Applications Conference* (Orlando, FL, USA) *(ACSAC 2017)*.

Association for Computing Machinery, New York, NY, USA, 474–485. https://doi.org/10.1145/3134600.3134644

[9] Youness Mourtaji, Mohammed Bouhorma, and Daniyal Alghazzawi. 2019. Intelligent Framework for Malware Detection with Convolutional Neural Network. In *Proceedings of the 2nd International Conference on Networking, Information Systems amp; Security* (Rabat, Morocco) *(NISS19)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3320326.3320333

[10] Weina Niu, Jian'An Xiao, Xiyue Zhang, Xiaosong Zhang, Xiaojiang Du, Xiaoming Huang, and Mohsen Guizani. 2021. Malware on Internet of UAVs Detection Combining String Matching and Fourier Transformation. *IEEE Internet of Things Journal* 8, 12 (2021), 9905–9919. https://doi.org/10.1109/JIOT.2020.3029970

[11] Yin Minn Pa Pa, Shogo Suzuki, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, and Christian Rossow. 2015. IoTPOT: Analysing the Rise of IoT Compromises. In *Proceedings of the 9th USENIX Conference on Offensive Technologies* (Washington, D.C.) *(WOOT'15)*. USENIX Association, USA, 9.

[12] Nisha Panwar, Shantanu Sharma, Peeyush Gupta, Dhrubajyoti Ghosh, Sharad Mehrotra, and Nalini Venkatasubramanian. 2020. IoT Expunge: Implementing Verifiable Retention of IoT Data. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy* (New Orleans, LA, USA) *(CODASPY '20)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/3374664.3375737

[13] M. Ping, B. Alsulami, and S. Mancoridis. 2016. On the effectiveness of application characteristics in the automatic classification of malware on smartphones. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*. 1–8. https://doi.org/10.1109/MALWARE.2016.7888732

[14] S. M. Pudukotai Dinakarrao, H. Sayadi, H. M. Makrani, C. Nowzari, S. Rafatirad, and H. Homayoun. 2019. Lightweight Node-level Malware Detection and Network-level Malware Confinement in IoT Networks. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 776–781.

[15] Nils Miro Rodday, Ricardo de O. Schmidt, and Aiko Pras. 2016. Exploring security vulnerabilities of unmanned aerial vehicles. In *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. 993–994. https://doi.org/10.1109/NOMS.2016.7502939

[16] Alessio Rugo, Claudio A. Ardagna, and Nabil El Ioini. 2022. A Security Review in the UAVNet Era: Threats, Countermeasures, and Gap Analysis. *ACM Comput. Surv.* 55, 1, Article 21 (jan 2022), 35 pages. https://doi.org/10.1145/3485272

[17] Arnab Kumar Saha, Jayeeta Saha, Radhika Ray, Sachet Sircar, Subhojit Dutta, Soummyo Priyo Chattopadhyay, and Himadri Nath Saha. 2018. IOT-based drone for improvement of crop quality in agricultural field. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*. 612–615. https://doi.org/10.1109/CCWC.2018.8301662

[18] Yang Shi, Fangyu Li, WenZhan Song, Xiang-Yang Li, and Jin Ye. 2019. Energy Audition Based Cyber-Physical Attack Detection System in IoT. In *Proceedings of the ACM Turing Celebration Conference - China* (Chengdu, China) *(ACM TURC '19)*. Association for Computing Machinery, New York, NY, USA, Article 27, 5 pages. https://doi.org/10.1145/3321408.3321588

[19] Rui Shu, Xiaohui Gu, and William Enck. 2017. A Study of Security Vulnerabilities on Docker Hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (Scottsdale, Arizona, USA) *(CODASPY '17)*. Association for Computing Machinery, New York, NY, USA, 269–280. https://doi.org/10.1145/3029806.3029832

[20] S. Shukla, G. Kolhe, P. Sai Manoj, and S. Rafatirad. 2019. Work-in-Progress: MicroArchitectural Events and Image Processing-based Hybrid Approach for Robust Malware Detection. In *2019 International Conference on Compliers, Architectures and Synthesis for Embedded Systems (CASES)*. 1–2.

[21] J. Su, D. V. Vasconcellos, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai. 2018. Lightweight Classification of IoT Malware Based on Image Recognition. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 02. 664–669.

[22] Luis Suastegui Jaramillo. 2018. Malware Detection and Mitigation Techniques: Lessons Learned from Mirai DDOS Attack. *Journal of Information Systems Engineering Management* 3 (07 2018). https://doi.org/10.20897/jisem/2655

[23] Amit Tambe, Yan Lin Aung, Ragav Sridharan, Martín Ochoa, Nils Ole Tippenhauer, Asaf Shabtai, and Yuval Elovici. 2019. Detection of Threats to IoT Devices Using Scalable VPN-Forwarded Honeypots. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy* (Richardson, Texas, USA) *(CODASPY '19)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/3292006.3300024

[24] Nalam Venkata Abhishek, Anshoo Tandon, Teng Joon Lim, and Biplab Sikdar. 2018. Detecting Forwarding Misbehavior In Clustered IoT Networks. In *Proceedings of the 14th ACM International Symposium on QoS and Security for Wireless and Mobile Networks* (Montreal, QC, Canada) *(Q2SWinet'18)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3267129.3267147

[25] Katrine Wist, Malene Helsem, and Danilo Gligoroski. 2020. Vulnerability Analysis of 2500 Docker Hub Images. *ArXiv* abs/2006.02932 (2020).

[26] L. Xiao, X. Wan, X. Lu, Y. Zhang, and D. Wu. 2018. IoT Security Techniques Based on Machine Learning: How Do IoT Devices Use AI to Enhance Security? *IEEE Signal Processing Magazine* 35, 5 (2018), 41–49.