Requirements Testing and Verification for Smart Systems Through Systematic Software Analysis

David Ke Hong
Dissertation Defense, May 15, 2019
Chair: Professor Z. Morley Mao

Smart end systems keep emerging

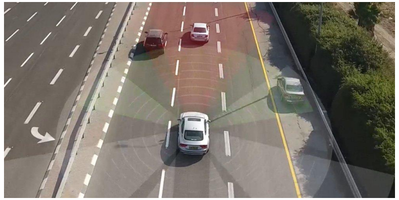
- Communication & information acquisition
  - Smartphone, wearable, IoT devices
- Transportation & mobility
  - Autonomous vehicle (AV)

Smartphone

Autonomous Vehicle

Key requirements

- **Performance** requirements
  - High mobility
  - Dynamic runtime
- **Security** requirements
  - Software complexity
  - Multi-party contribution
- **Safety** requirements
  - Driving safety logic in AV software



Thesis research goal

- **My thesis research**: Develop systematic software analysis approaches for testing and verifying key ***performance***, ***security*** and ***safety*** requirements of smart end systems
    - ***Static program analysis*** => completeness guarantee
    - ***Runtime profiling*** => capturing runtime dynamics

Thesis statement

Systematic software analysis approaches based on *static program analysis* and *runtime profiling, with domain-specific customization*, can lead to effective testing and verification of key *performance*, *security* and *safety* requirements for smart system software

Thesis work overview

Part I: Performance requirement testing and noncompliance diagnosis for mobile apps

Part II: Security vulnerability detection and mitigation in AV software systems

Part III: Self-driving safety requirement verification for AV software

Thesis contribution

- Performance requirement testing & problem diagnosis
  - **Thesis contribution**: *low-overhead, cross-layer* runtime profiling and performance diagnosis for smartphone systems
- Security and safety requirement verification
  - **Thesis contribution**: the first to apply static analysis for systematic discovery & mitigation of *new vulnerability* and verification of *safety requirement* for AV software systems

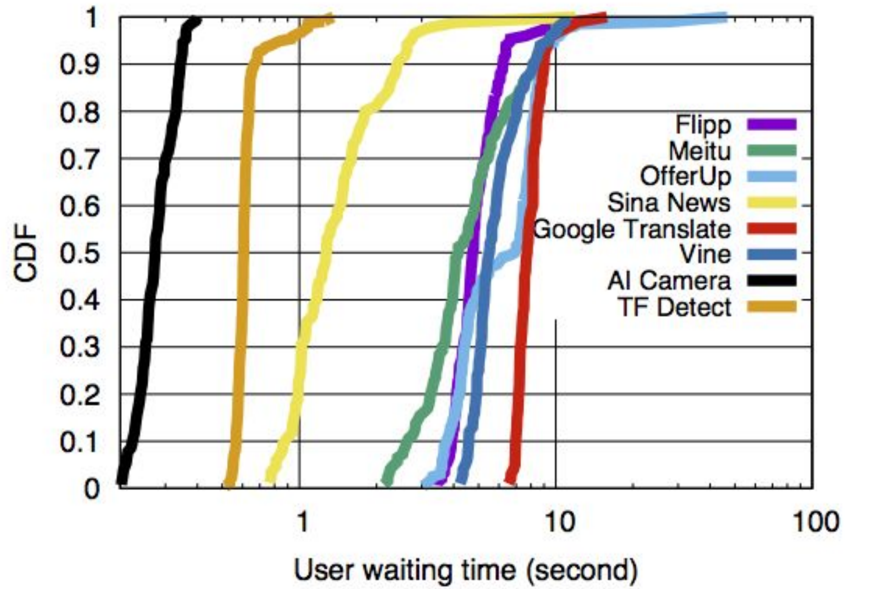
Part I: A Systematic, Cross-Layer Performance Diagnosis Framework for Mobile Platforms

- Platform support for performance requirement validation
- Runtime profiling and performance diagnosis

PerfProbe: A Systematic, Cross-Layer Performance Diagnosis Framework for Mobile Platforms. In MOBILESoft'19.

Background

- Unpredictable performance degradation violates the performance requirement for smartphone apps
  - 100 popular apps
  - Tail latency: **2~8x increase**

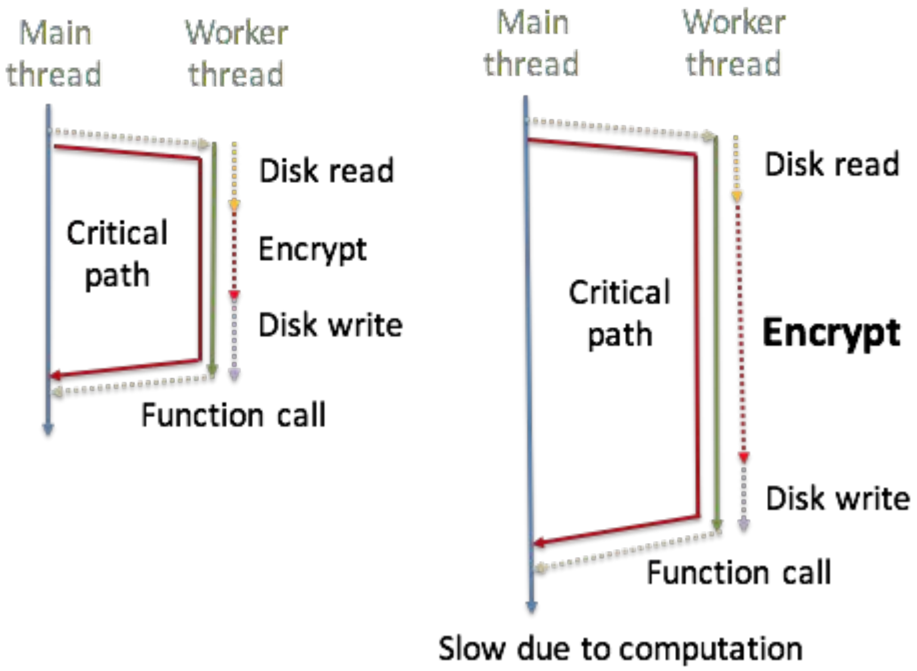


Contribution

- Profiling and ***associating app and system-layer runtime events*** can lead to
  - Holistic, cross-layer insights to better pinpoint the root cause of performance degradation
    - Built a ***low-overhead***, ***cross-layer*** performance profiling and diagnosis framework, PerfProbe, for mobile platforms
    - Existing work, e.g., AppInsight [OSDI ‘12], Panappticon [CODES'13], focusing on single-layer runtime profiling

Why cross-layer profiling

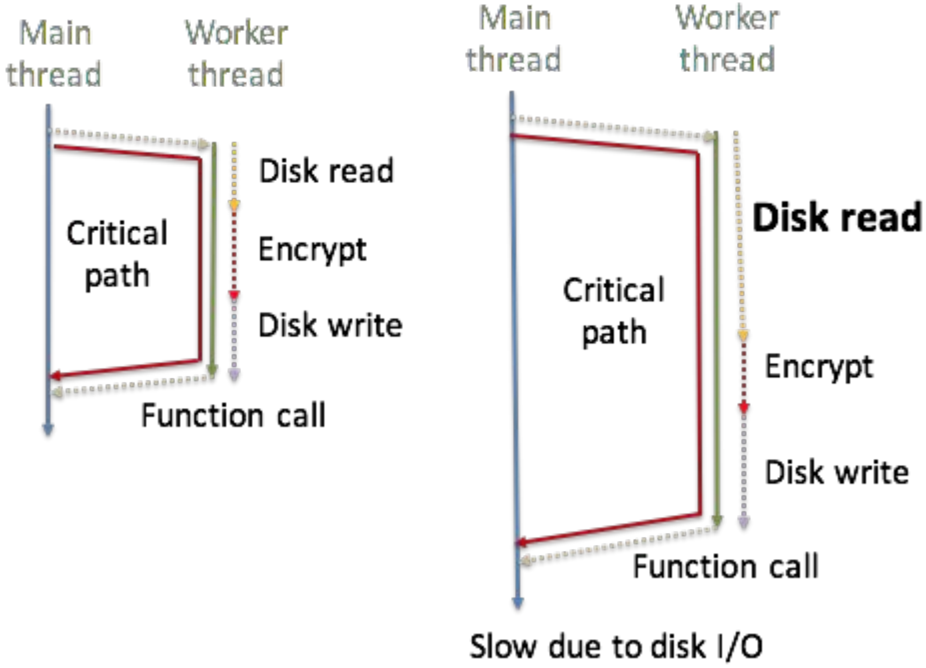
- Motivating example: encrypt a file on SD card



Performance degradation due to slowdown in **Encrypt**

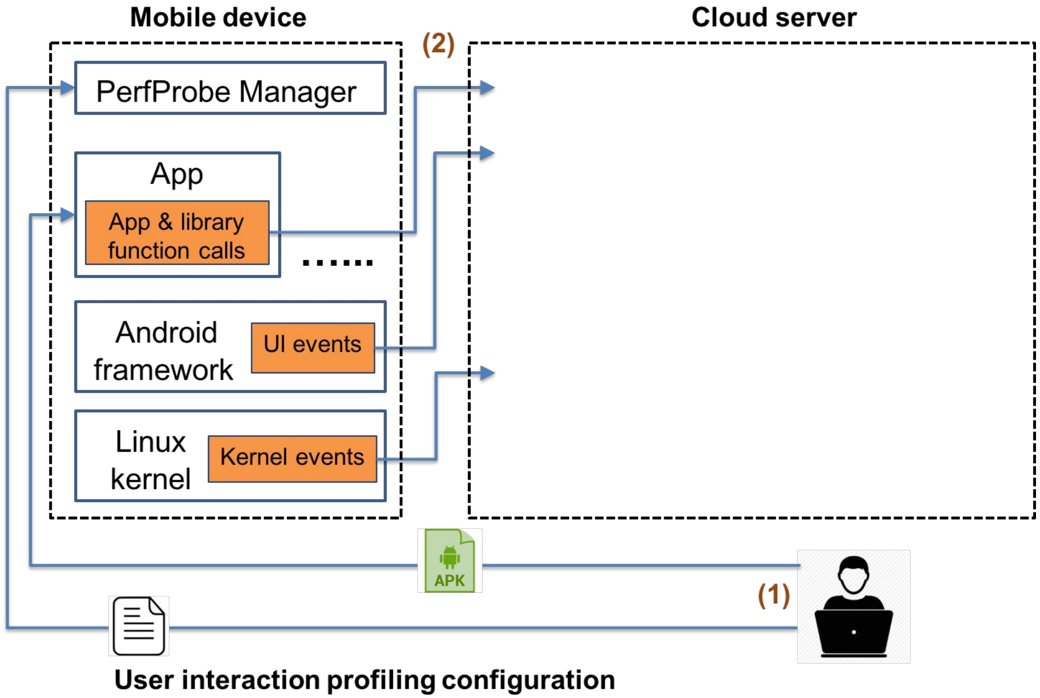
Why cross-layer profiling

- Motivating example: encrypt a file on SD card



Performance degradation due to slowdown in **Disk read**

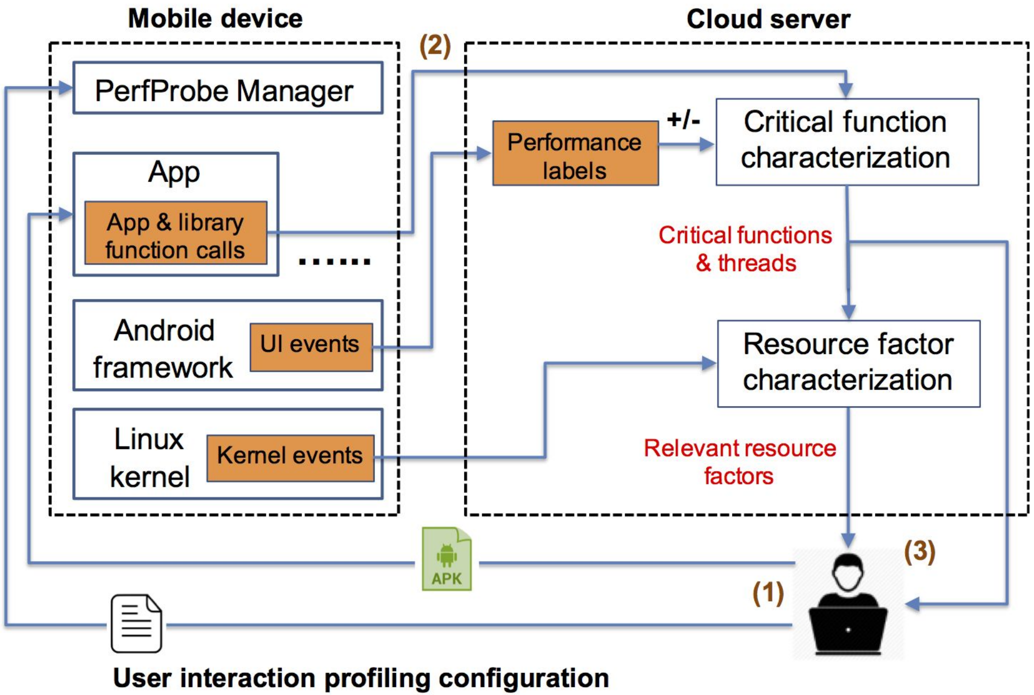
PerfProbe overview



On-device: runtime **performance monitoring** and profiling

1. App's call stack
2. OS event trace

PerfProbe overview



On-device: runtime **performance monitoring** and profiling

1. App's call stack
2. OS event trace

Server-side: cross-layer trace analysis for **problem diagnosis**

Experiment results

- Cross-layer profiling incurs < 3.5% increase of delay
  - Android's built-in profiling incurs 3-22% increase
- Usefulness of diagnosis findings
  - Guiding performance optimizing solutions to reduce latency of 6 popular apps by 32-86%
- Findings acknowledged by iNaturalist developer
  - Improve the response of a key interaction with **10x speedup**
  - Developer has **adopted our fixing suggestion** [link]

Conclusion (Part I)

- ***The first to design a low-overhead, cross-layer profiling and performance diagnosis framework*** for mobile platforms
- Improved performance of 6 popular Android apps using PerfProbe's diagnosis findings

Part II: Detecting and Mitigating Publish-Subscribe Overprivilege for Autonomous Vehicle Systems

- First characterization of overprivilege in AV systems
- Static analysis tool for systematic detection and mitigation of overprivilege

Autonomous vehicle software systems

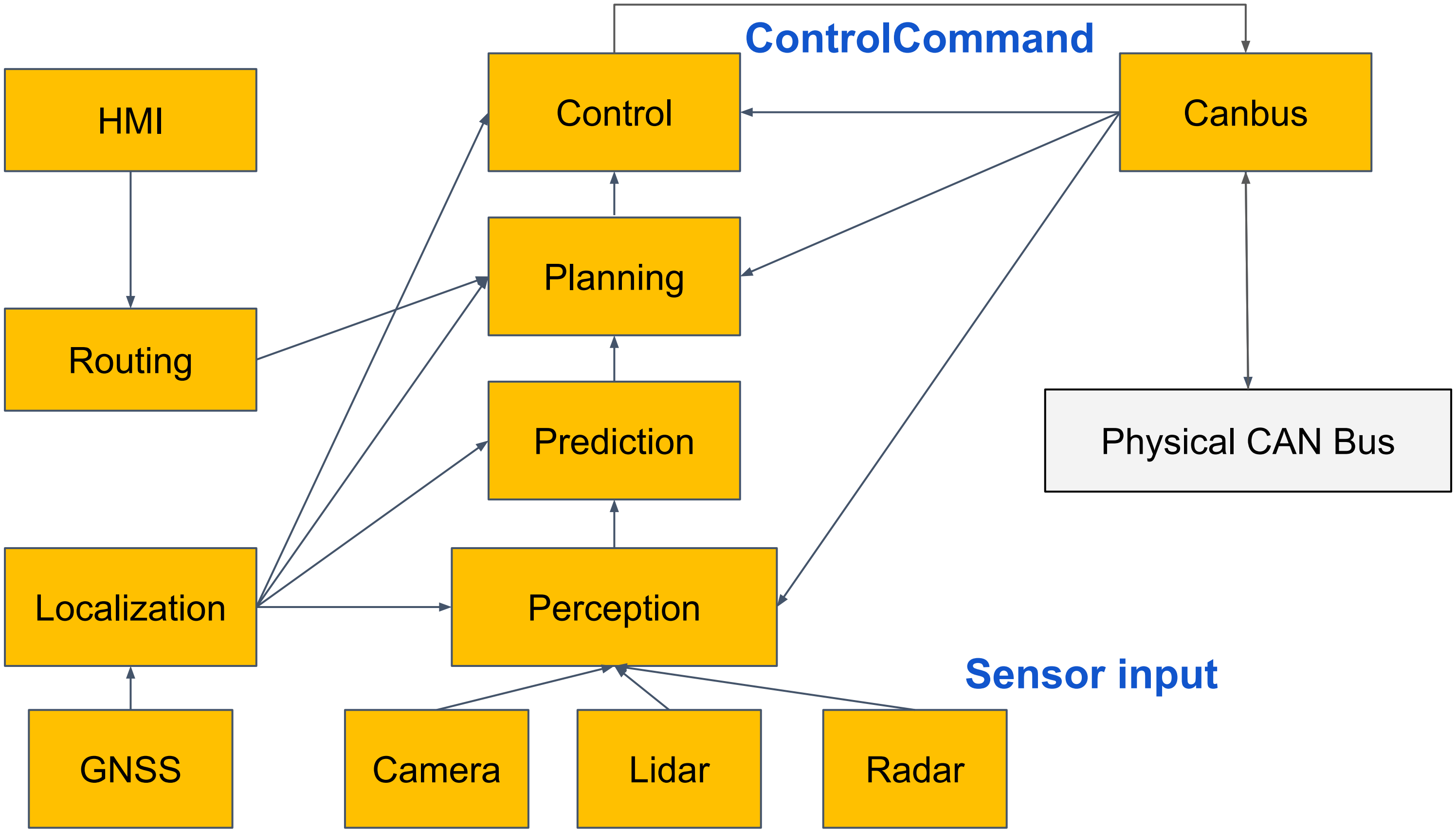
- *Robot Operating System* (ROS) middleware
  - Commonly used in various autonomous systems (e.g., AVs, drones, robots, etc.)

- ROS-based open-source AV platforms

AutoWare

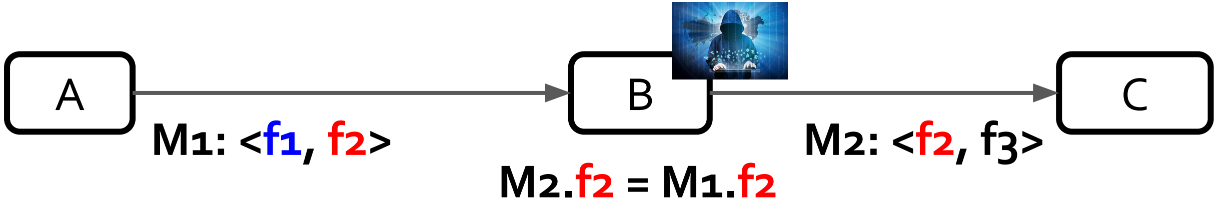
apollo

Publish-subscribe messaging in AV systems

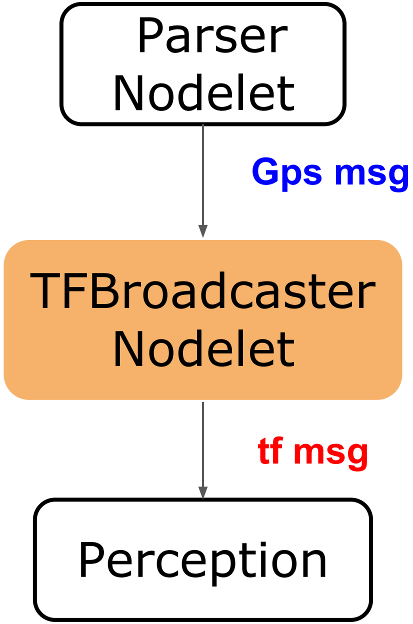


Publish-subscribe overprivilege characterization

- Subscriber-side overprivilege
  - Certain fields in a subscribed message are ***not read*** => ***over-granted read permission***
- Publisher-side overprivilege
  - Certain fields in a published message are ***not written*** by publisher => ***over-granted write permission***



Overprivilege in Baidu Apollo & Autoware



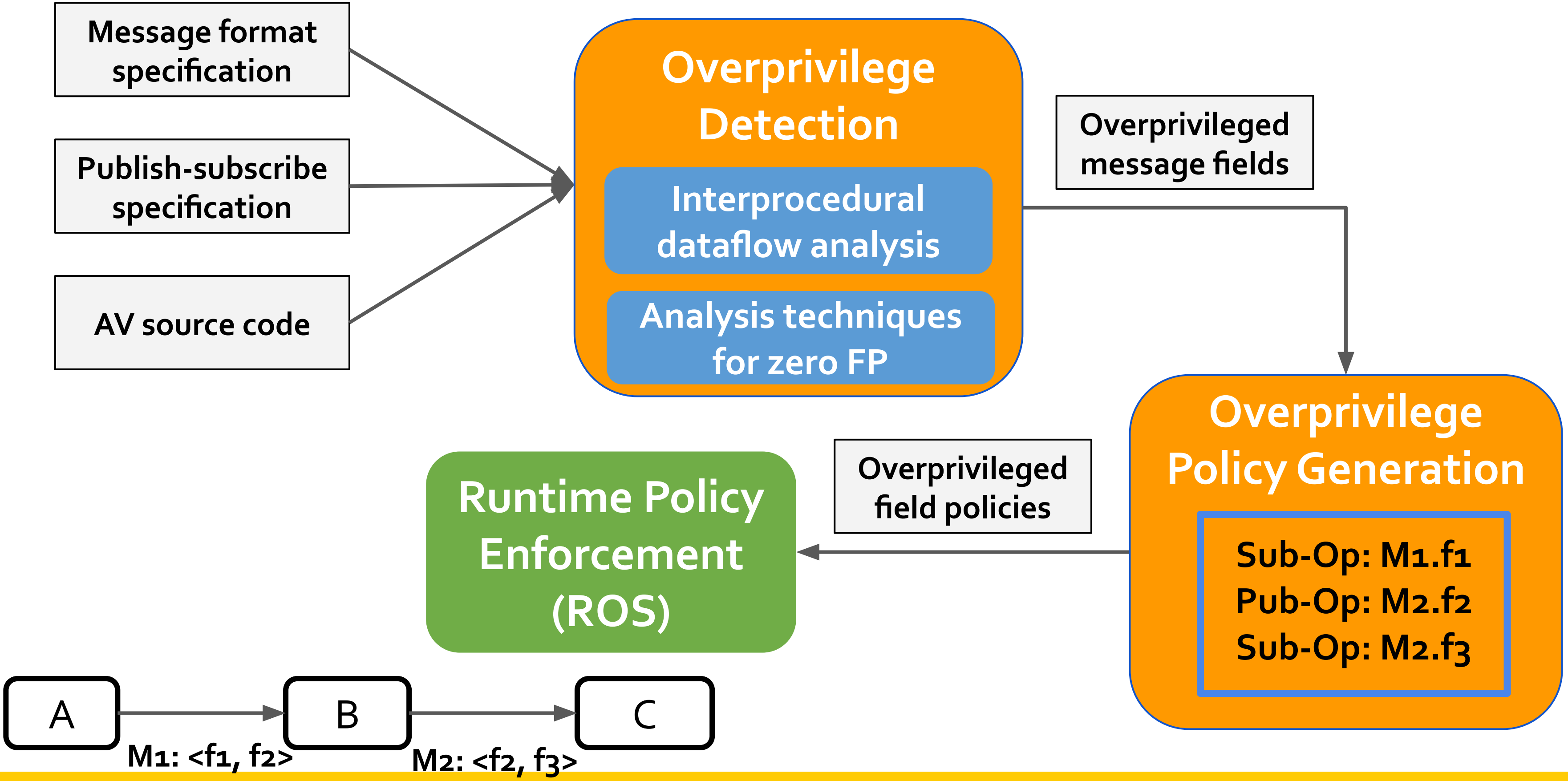

```
void TFBroadcaster::gps_to_transform_stamped(
    const ::apollo::localization::Gps& gps,
    geometry_msgs::TransformStamped* transform) {
    ……
    transform->header.stamp = time.fromSec(gps.header().timestamp_sec());
    ……
    transform->transform.translation.x = gps.localization().position().x();
    transform->transform.translation.y = gps.localization().position().y();
    transform->transform.translation.z = gps.localization().position().z();
    transform->transform.rotation.x = gps.localization().orientation().qx();
    transform->transform.rotation.y = gps.localization().orientation().qy();
    transform->transform.rotation.z = gps.localization().orientation().qz();
    transform->transform.rotation.w = gps.localization().orientation().qw();
}
```

Publisher-side overprivilege on *tf.transform*

Contribution

- Static program analysis ***incorporating AV-specific software programming models*** can lead to
  - Systematic discovery of security vulnerabilities and generation of access control defense policies in AV software systems
    - Built a publish-subscribe overprivilege detection and mitigation system, ***AVGuardian***, for ROS-based AV systems
    - Achieved zero false positive in overprivilege detection

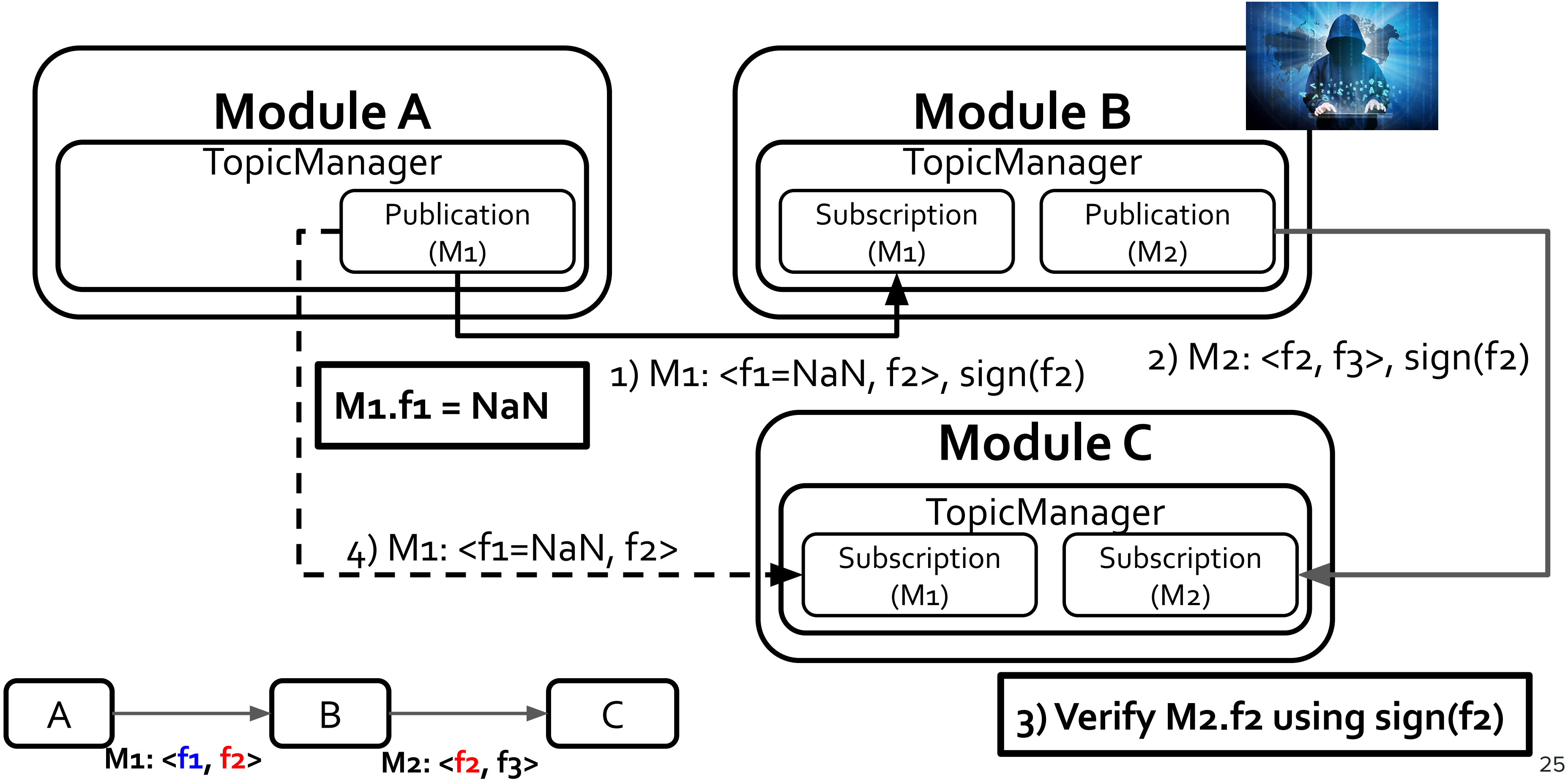
AVGuardian overview



Towards zero FP in overprivilege detection

- Challenges with static program analysis
  - **Virtual functions**
  - **Asynchronous event callbacks**
- Customized data flow analysis
  - Conservative subclass binding for virtual functions
  - Enumerating all possible orders of event callbacks
  - Reduced 28 false positives out of 523 true positives

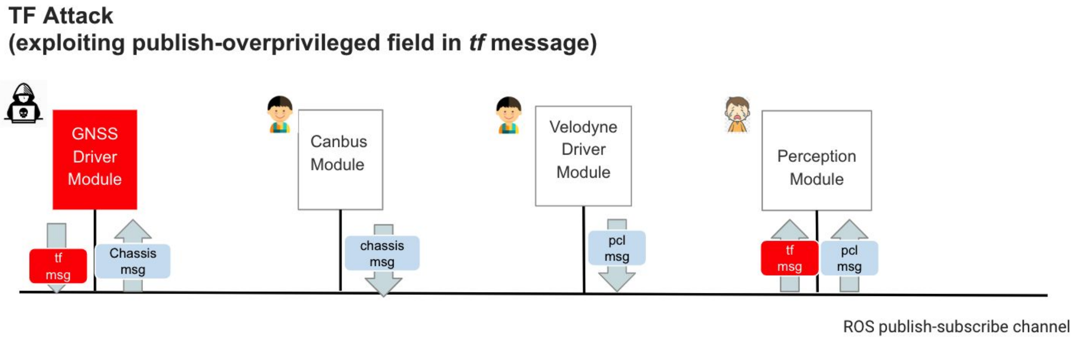
Defense: ROS-layer policy enforcement

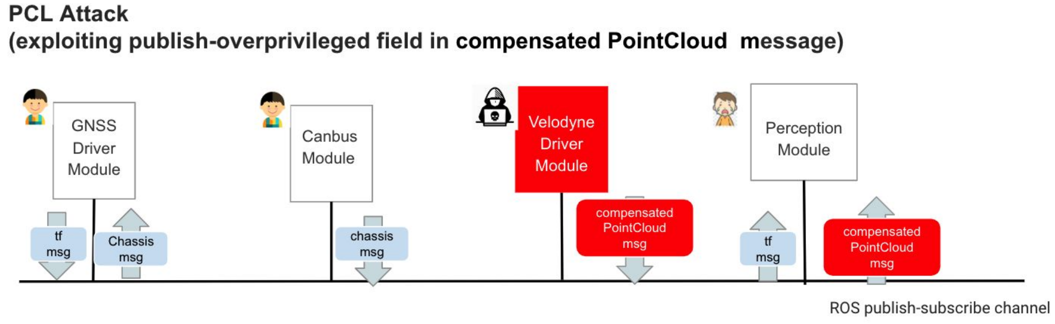


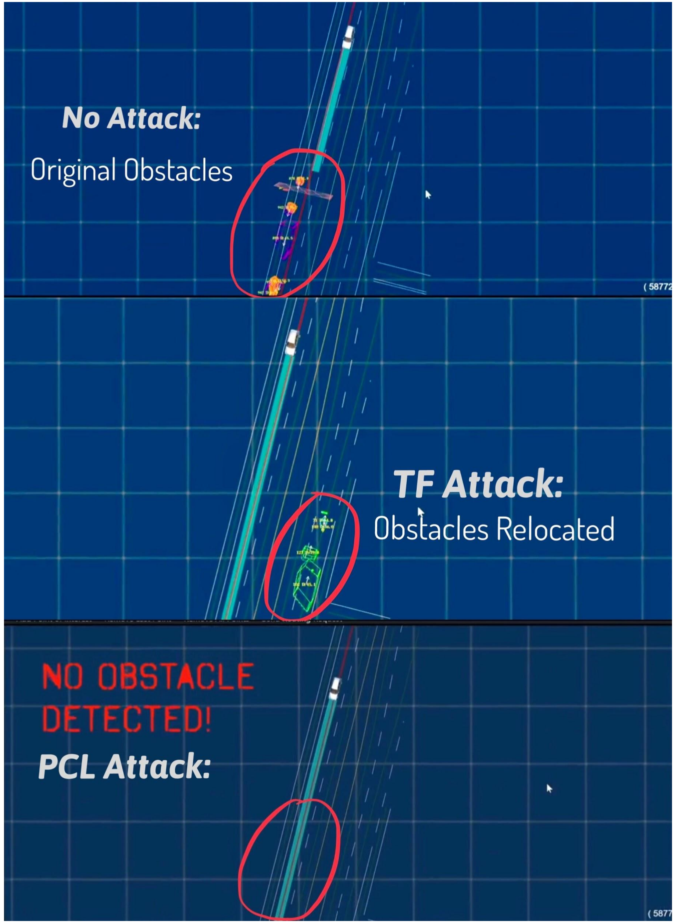
Vulnerability findings

- Exploits from publisher-side overprivilege
  - **TF attack** => obstacle relocation
  - **PCL attack** => obstacle remove
  - Security consequence: vehicle collision
- Exploit from subscriber-side overprivilege
  - **VIN stealing attack** => leakage of AV's VIN
  - Security consequence: AV owner's identity theft

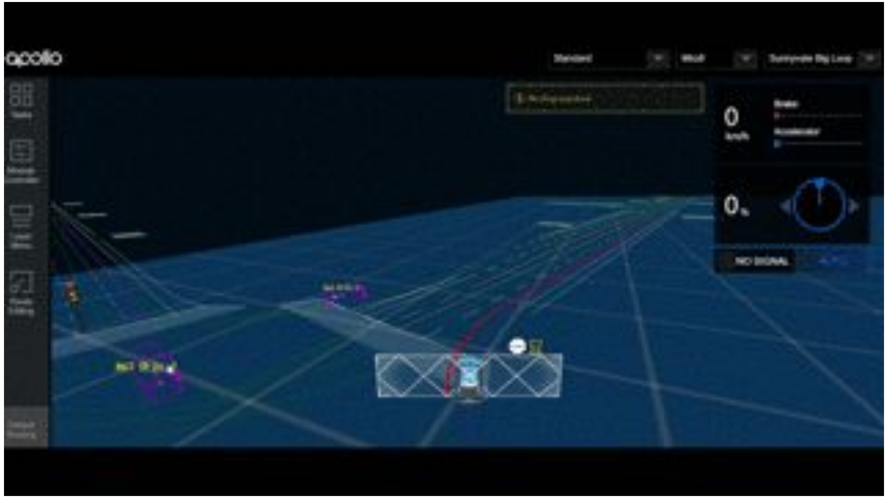
TF/PCL attack

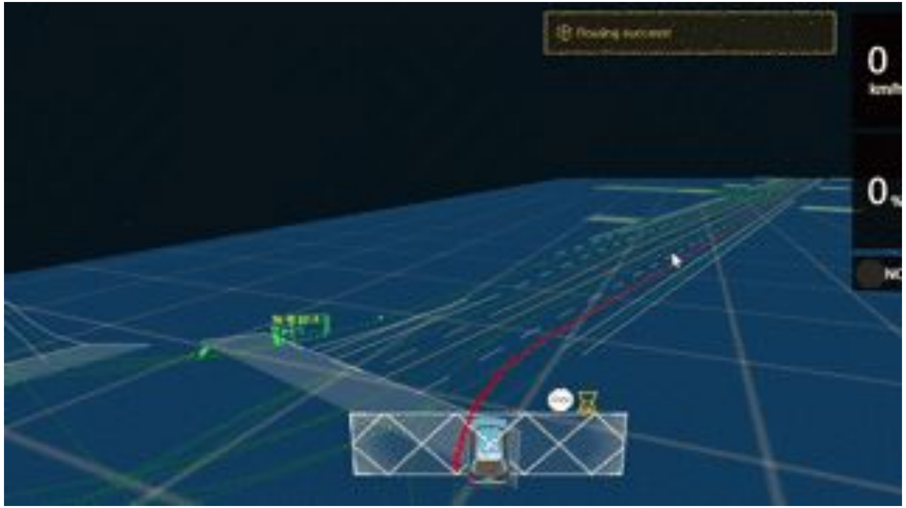






TF Attack: obstacle relocation

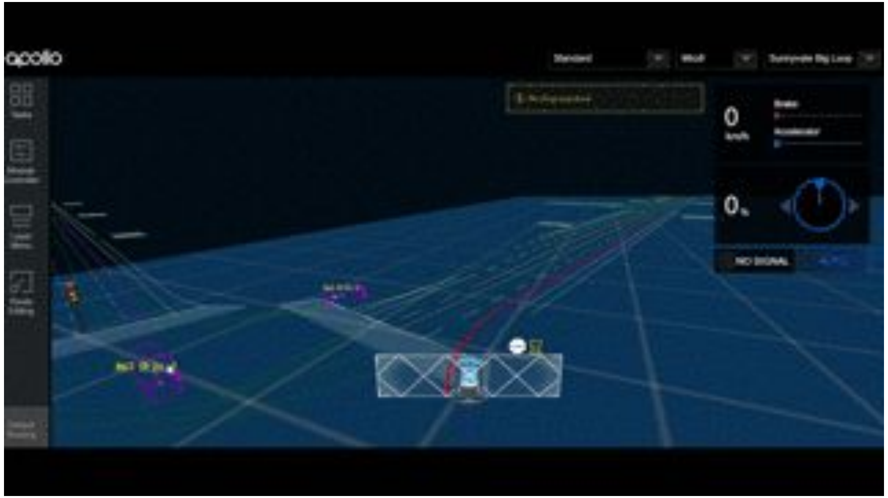




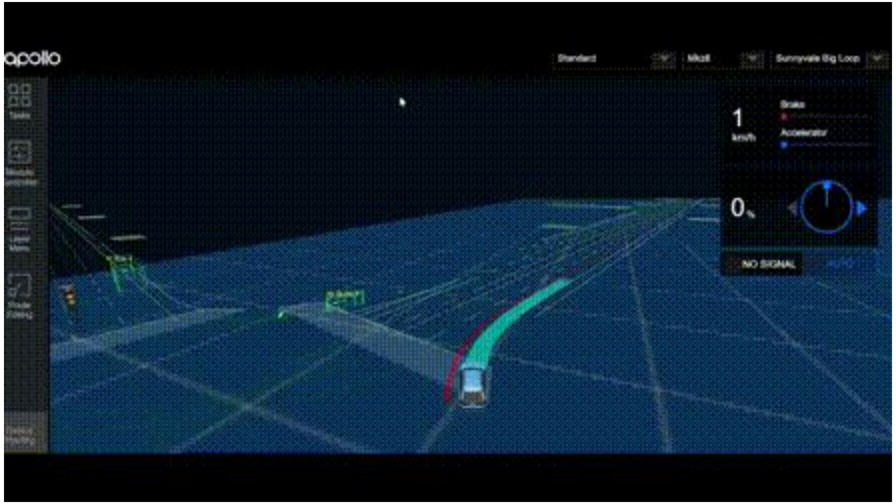
Control group video demo

TF attack video demo

PCL Attack: obstacle remove



Control group video demo



PCL attack video demo

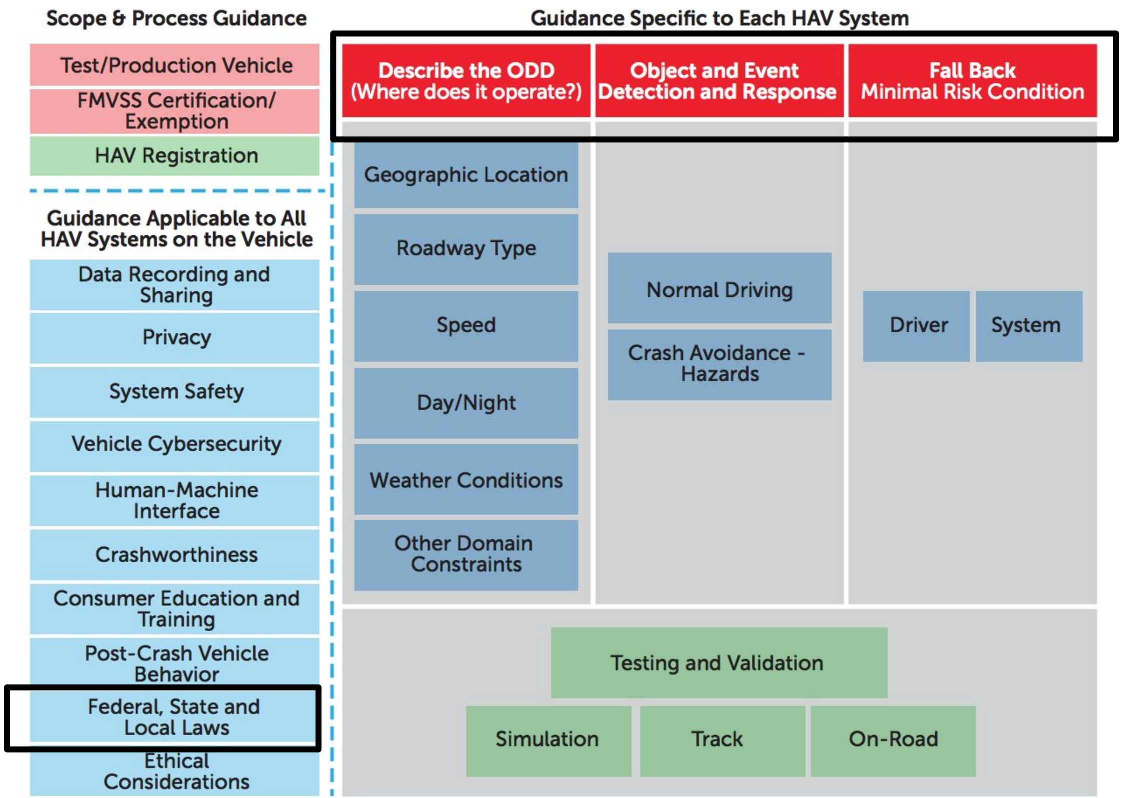
Conclusion (Part II)

- ***The first to design a static analysis framework*** for detecting and mitigating overprivilege in AV software systems
- Performed responsible disclosure to Baidu Apollo team and confirmed 3 attacks as valid

Part III: Verifying Self-Driving Safety Requirement Compliance for Autonomous Vehicle Systems

- A first driving safety verification framework for AV software
- Static analysis tool for systematic verification of safety rules

Safety requirements for AV software



Does AV software comply with the defined **object design domain (ODD)**, **object and event detection and response (OEDR)**, **minimal risk condition (MRC)**?

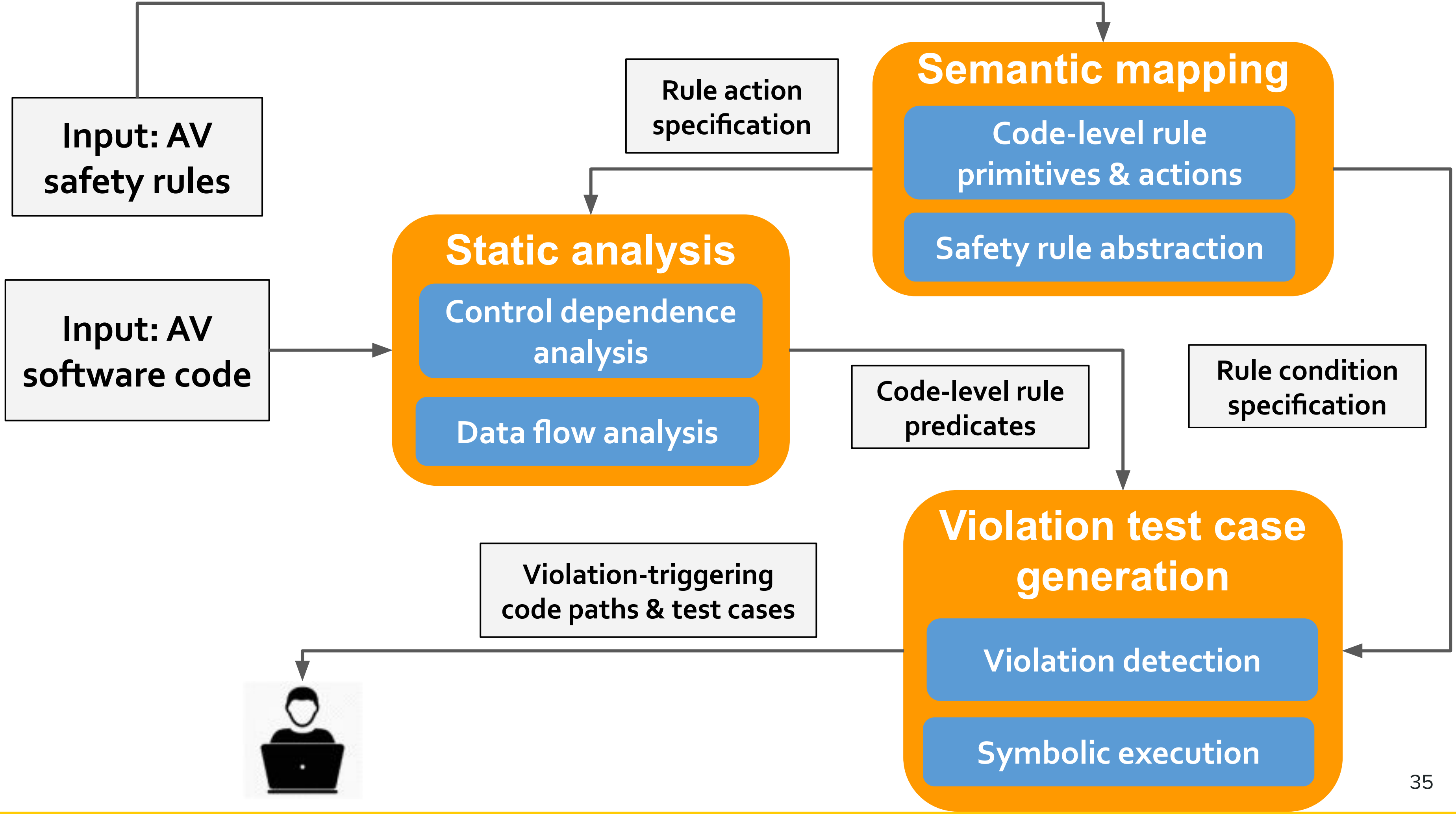
Does AV software generate self-driving decisions obeying **traffic law**?

Contribution

- Static program analysis ***incorporating self-driving semantics*** can lead to
  - Systematic detection of safety policy violation in the implementation of AV software
    - Built a safety compliance verification framework, ***AVerfier***, for AV software systems
    - Towards detecting policy violation with *zero false negative and low false positive*

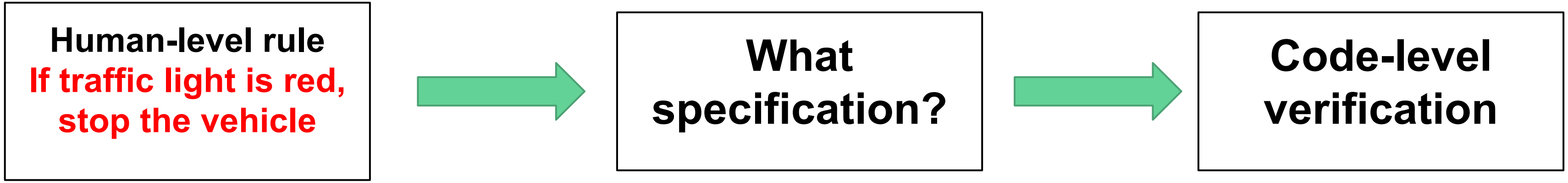
Related work & novelty

- Existing work in consistency checking of policy enforcement
  - Linux security policy & Android permissions
- Key difference: targeting at driving safety policies
  - Containing rich road traffic and driving semantics
  - Requiring specific formulation of driving safety policies to bridge the semantic gap between policy & code


Input: AV safety rules
Rule action specification
Semantic mapping
Code-level rule primitives & actions
Safety rule abstraction
Static analysis
Control dependence analysis
Data flow analysis
Input: AV software code
Code-level rule predicates
Rule condition specification
Violation test case generation
Violation detection
Symbolic execution
Violation-triggering code paths & test cases

Domain-specific challenge

- Definition of policy specification



- Solution
    - Policy specification composed by relevant APIs of the AV software

Safety policy specification example

- High-level policy
  - If ***traffic light is red***, ***stop*** the vehicle
- Specification
  - If ***signal.color() == TrafficLight::RED***, call ***BuildStopDecision***
- Validated generality on 35 safety rules of traffic laws

Towards completeness of rule verification

- Code-level rule predicate extraction
  - **Formulated as control dependencies**

A statement S2 is control dependent on S1 f and only if S2's execution is conditionally guarded by S1.

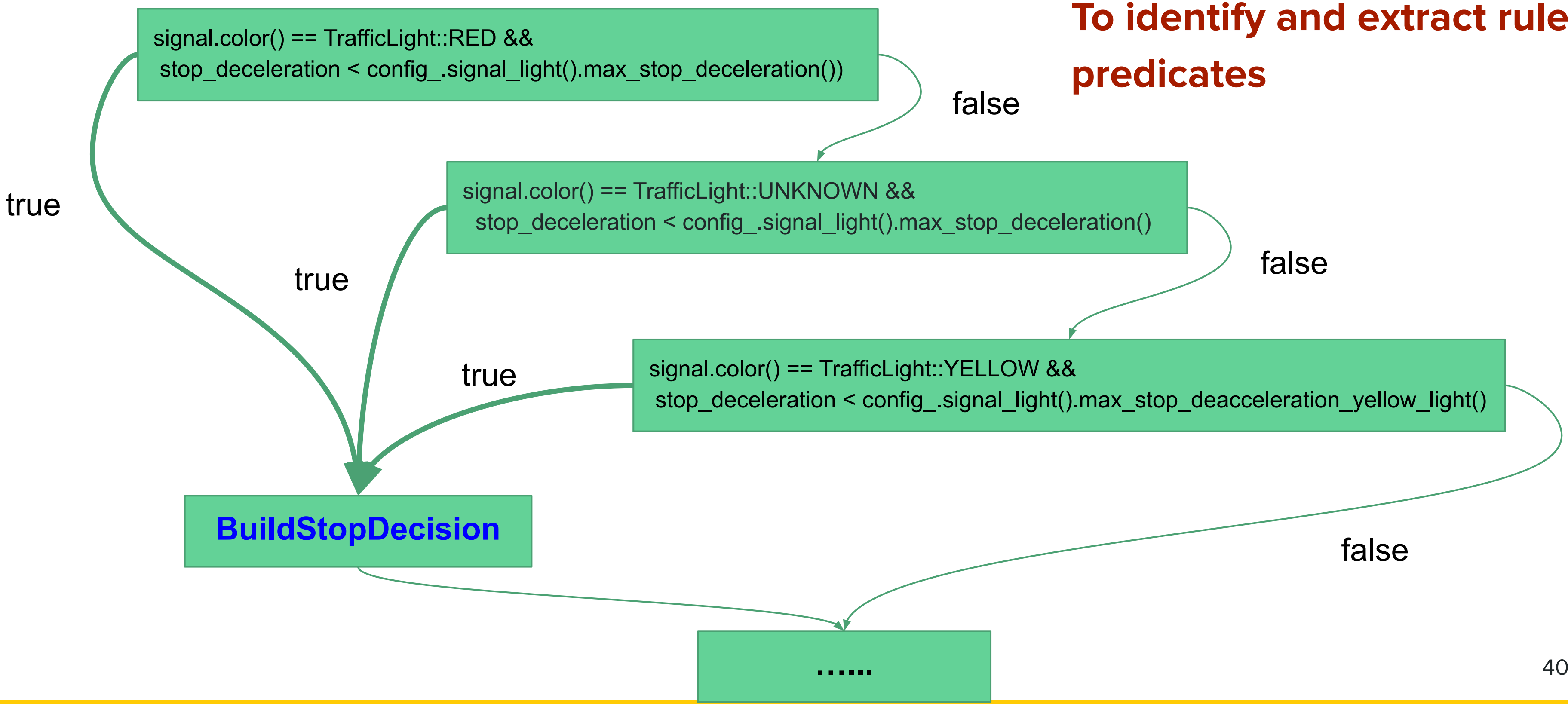
```
S1   if x > 2 goto L1
S2        y := 3
S3   L1: z := y + 1
```

Signal light case in Baidu Apollo

<table>
<tr><td>Action</td><td>SignalLight::ApplyRule</td></tr>
<tr><td>stop</td><td><pre>for (auto& signal_light : signal_lights_from_path_) {
      ……
      if ((signal.color() == TrafficLight::RED &&
         stop_deceleration < config_.signal_light().max_stop_deceleration()) ||
         (signal.color() == TrafficLight::UNKNOWN &&
         stop_deceleration < config_.signal_light().max_stop_deceleration()) ||
         (signal.color() == TrafficLight::YELLOW &&
         stop_deceleration < config_.signal_light().max_stop_deacceleration_yellow_light())) {
           ……
         if (BuildStopDecision(frame, reference_line_info, &signal_light)) {
           has_stop = true;
           signal_debug->set_is_stop_wall_created(true);
         }
       }
       ……
 }</pre></td></tr>
</table>

Control dependency analysis

To identify and extract rule predicates



Towards completeness of violation checking

```
  ......
  if (stop)
    pedestrians.push_back(obstacle_id);
}
if (!pedestrians.empty()) {
  // stop decision
  double stop_deceleration = util::GetADCStopDeceleration(reference_line_info, crosswalk_overlap->start_s,
      config_.crosswalk().min_pass_s_distance());
  if (stop_deceleration < config_.crosswalk().max_stop_deceleration())
      crosswalks_to_stop.push_back(std::make_pair(crosswalk_overlap, pedestrians));
}
}
for (auto crosswalk_to_stop : crosswalks_to_stop)
  BuildStopDecision(frame, reference_line_info, const_cast<hdmap::PathOverlap*>(crosswalk_to_stop.first),
                    crosswalk_to_stop.second);
```

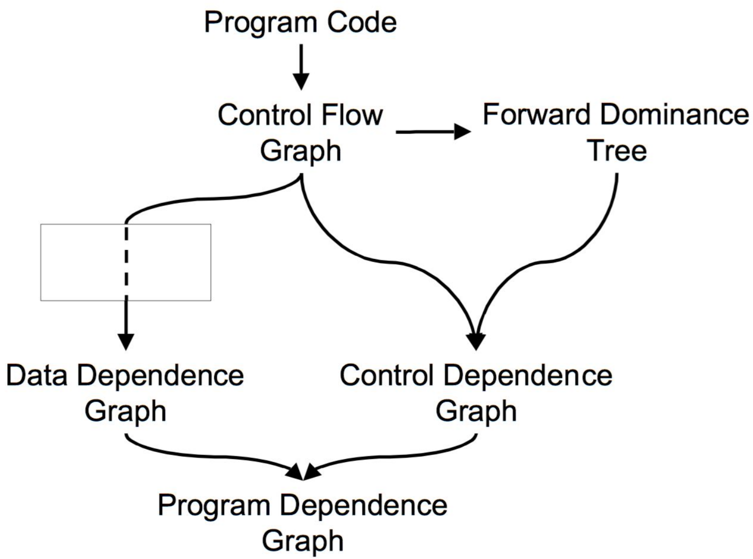
Implicit dependency (only if *pedestrians* is not empty)

Implicit dependency (only if *crosswalks_to_stop* is not empty)

Only one control-dependent predicate identified

Towards completeness of rule verification

- Code-level rule predicate extraction
  - **Program dependence analysis**



Policy inconsistency findings in Apollo

- Rule 1: Slow down to 15 mph when approaching a speed bump.
  - Found in Apollo v3.0 fixed in Apollo v3.5
- Rule 2: Do not pass if you are within 100 feet of an intersection.
  - Found in Apollo v3.0 fixed in Apollo v3.5

Towards low FP rate of violation detection

- Given a violation detected in policy checking, apply ***symbolic execution*** to systematically validate that a true violation exists
    - Symbolic execution gives proof of completeness
    - Engineering challenge with extending KLEE to run on AV software code base

Future research directions

- Systematic test case generation for violation
  - Preprocessing through flow analysis to prune irrelevant control flow paths
  - Only apply symbolic execution on relevant paths
- Semantic comparison
  - How to compare code-level predicates with specification
    - Inclusive, partial overlapping, etc.

Conclusion (Part III)

- ***The first to design a static analysis framework*** for driving safety compliance verification in AV software systems
- Proposed AV semantic mapping to enable flexible specification of driving safety policies with AV software code-level semantics

Conclusion

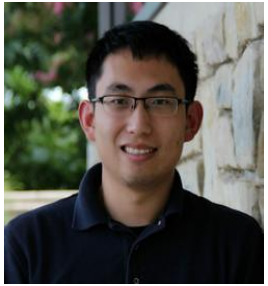
Performance, ***security*** and ***safety*** are key requirements for smart end systems.

We perform ***system-specific customization*** on systematic software analysis approaches for effective ***requirement testing and verification*** of smart system software.

Acknowledgement

RobustNet Research Group
University of Michigan

Z. Morley Mao



Qi Alfred Chen

Scott Mahlke

Florian Schaub

Other research and industry collaborators: Jason Flinn, Peter Chen, Harsha Madhyastha, Mark Gordon, Ashkan Nikravesh, John Kloosterman, Ze Zhang, Yulong Cao, Shengtuo Hu, Jiwon Joung, Brandon Nguyen, Yuqi Jin (UM), Scott Haseley, Geoffrey Challen (UIUC), Yadi Ma, Sujata Banerjee (HP Labs), Mahesh Ketkar, Michael Kishinevsky (Intel)

Conclusion

Performance, security and safety are key requirements for smart end systems

We incorporate system-specific knowledge to customize systematic software analysis approaches for effective requirement testing and verification of smart system software

Reference

[1] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, in OSDI'12, 2012.
[2] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: Event-based Tracing to Measure Mobile Application and Platform Performance. In Proc. of CODES+ISSS, 2013.
[3] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified, in Proc. of ACM CCS, 2011.
[4] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications, in IEEE Security & Privacy, 2016.
[5] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In Proc. of ACM CCS, 2002.
A. Edwards, T. Jaeger, and X. Zhang. Runtime verification of authorization hook placement for the Linux security modules framework. In Proc. of ACM CCS, 2002.
[6] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In Proc. of ACM CCS, 2003.
[7] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In Proc. of USENIX Security, 2008.
[8] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In Proc. of USENIX Security, 2011.
[9] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In Proc. of ACM CCS, 2013.
[10] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In Proc. of NDSS, 2012.
[11] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao, "Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework," in NDSS, 2016.