PRINCETON UNIVERSITY
Williams

CPS: Towards Secure, Privacy-Preserving, Verifiable Cyberphysical Systems

PIs: Esin Tureci and Kelly Shaw

CNS 1739701 (UR) and CNS 1739674 (Princeton), Oct. 1, 2017 - Sept. 30, 2021

Introduction

CPS/IoT systems are distributed over numerous sensing, actuation and computing devices, with a diverse set of specifications including

- physical and often safety-critical aspects to their operation and
- time sensitive requirements and energy constraints.

Challenges for achieving correctness and security include:

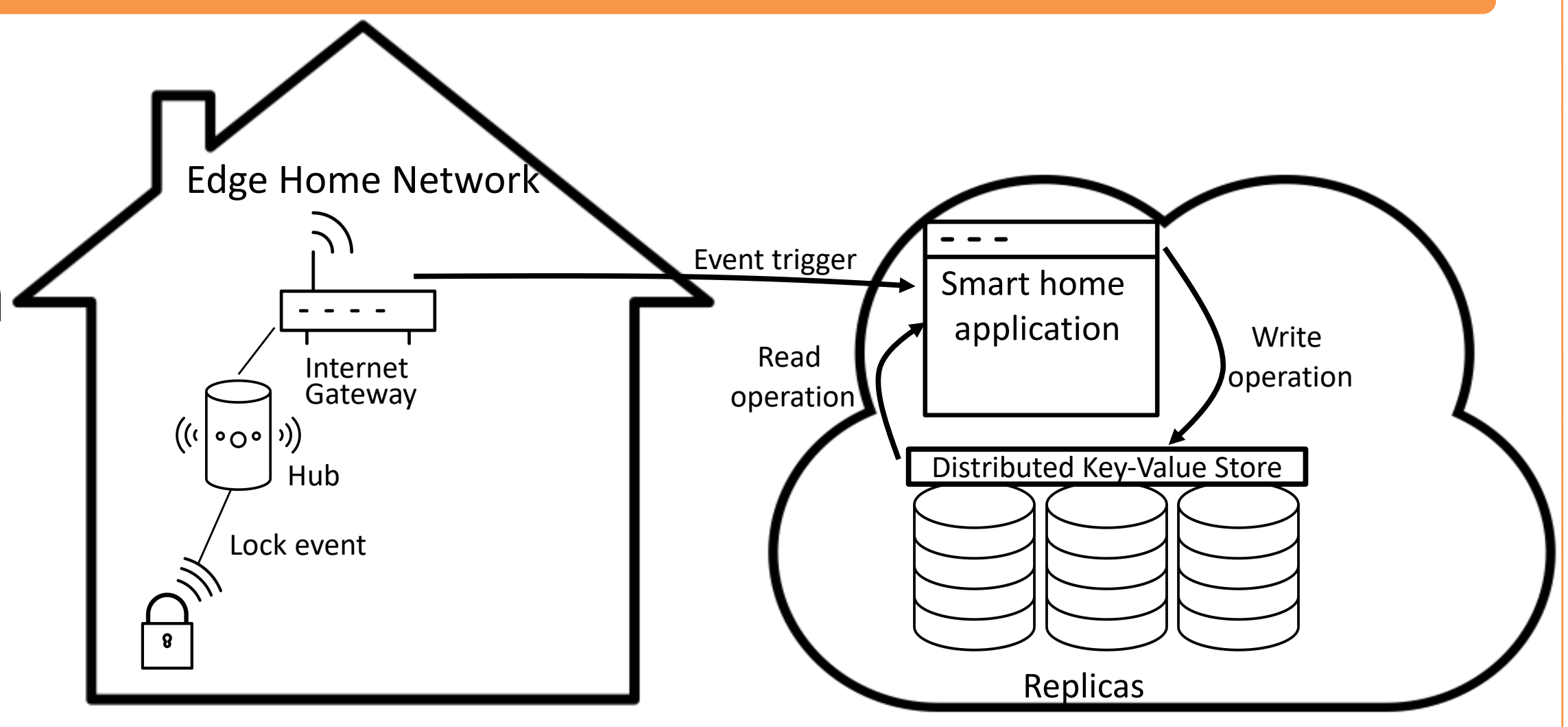
- increasing frequency of dynamic code compilation and deployment and
- heterogeneity across devices, systems, and within individual devices.

Proposed work:

- Develop formal, static verification techniques for correctness
- Develop hardware support for formal, dynamic system verification
- Open-source release of techniques and tools

Challenges and Opportunities in CPS/IoT

CPS/IoT devices communicate, update, and store state in distributed cloud storage. Data may be updated inconsistently in a variety of ways, creating potential correctness, security, and reliability flaws.



- Message delays or reorderings can leave device state inconsistent with data state stored in cloud.
- Interaction of different processing elements and applications in systems may introduce data inconsistencies and incorrect functionality.
- Lack of atomic read-modify-write operations in device programming interface can result in lost data updates or non-intuitive results.
- Weak data consistency models used by cloud infrastructure make it challenging for application developers to reason about correctly updating device and cloud state as order of updates is not guaranteed.
- Complexity in testing and deploying CPS/IoT systems and software is massive, due to the needs to ensure security and reliability.

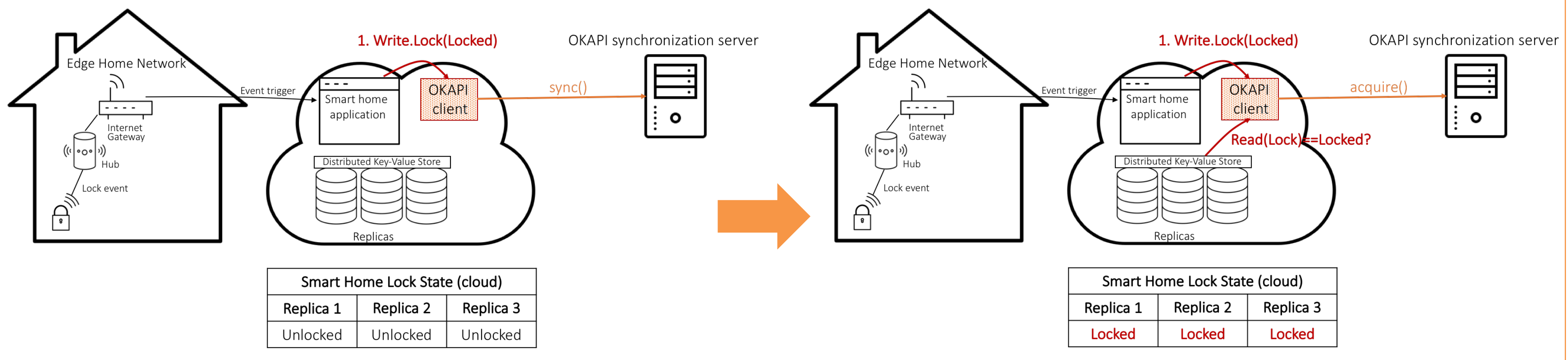
Case study: Smart Home Environments

Smart Home devices use cloud-based platforms to store application state, compute in response to sensor events, and trigger device actuation.

Problem 1: System support for cloud-based platforms is insufficient:

- Asynchronous events and distributed devices lead to event reordering.
- Concurrent event processing leads to race conditions due to non-existent read-modify-write atomic primitives.

Solution: OKAPI provides a platform synchronization service that orders events and provides synchronization mechanisms.



Smart Home Lock State (cloud)		
Replica 1	Replica 2	Replica 3
Unlocked	Unlocked	Unlocked

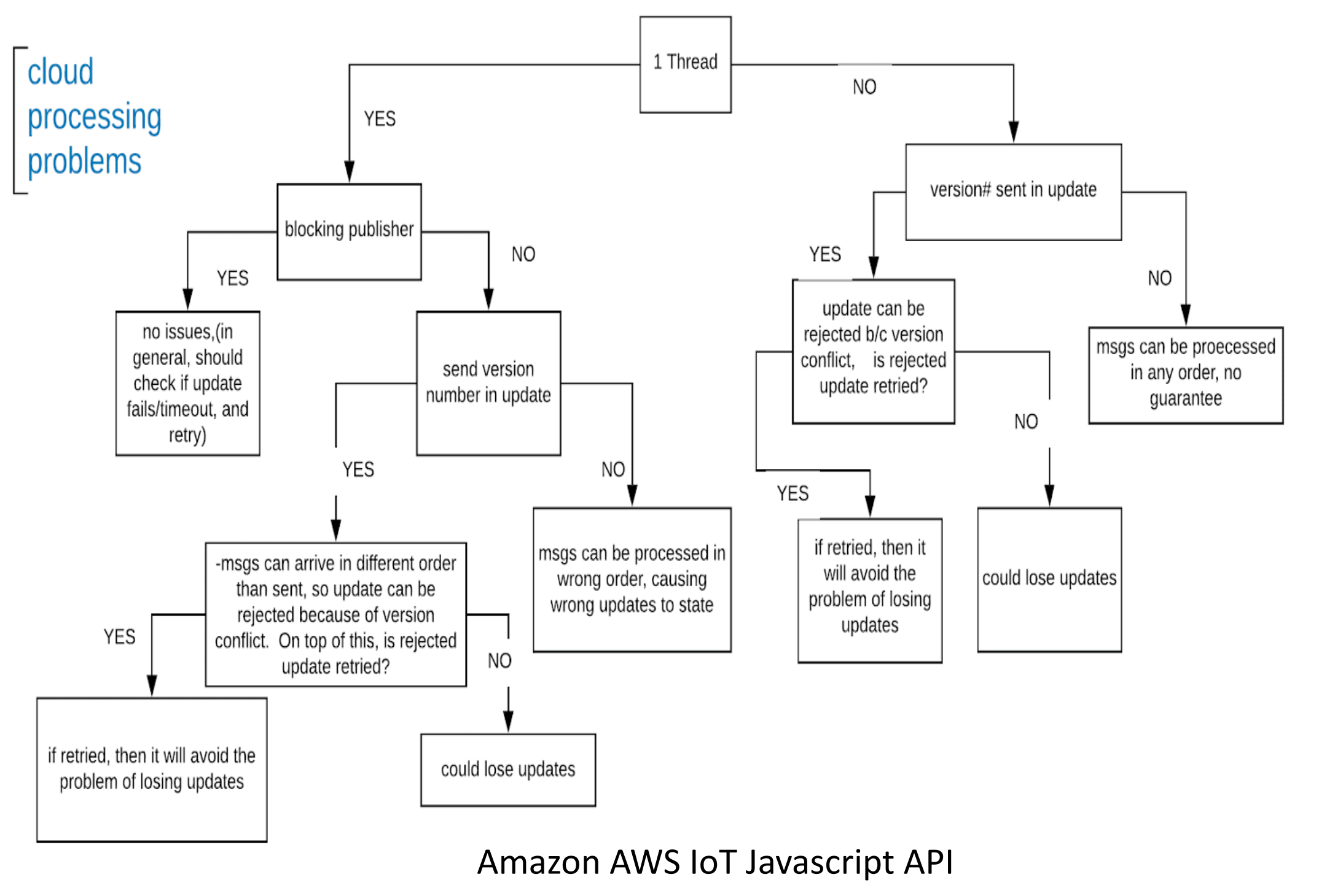
1.) Ask OKAPI for data lock

Smart Home Lock State (cloud)		
Replica 1	Replica 2	Replica 3
Locked	Locked	Locked

2.) Lock acquired, enabling atomic updates

Problem 2: Application developers do not factor eventual consistency and concurrent event processing into program development, resulting in incorrect applications.

Approach: Create static analysis tools to identify application code that may result in lost updates or reordered updates to cloud and device state.


cloud processing problems
1 Thread
YES
NO
blocking publisher
YES
no issues,(in general, should check if update fails/timeout, and retry)
NO
send version number in update
YES
-msgs can arrive in different order than sent, so update can be rejected because of version conflict. On top of this, is rejected update retried?
YES
if retried, then it will avoid the problem of losing updates
NO
could lose updates
NO
msgs can be processed in wrong order, causing wrong updates to state
version# sent in update
YES
update can be rejected b/c version conflict, is rejected update retried?
YES
if retried, then it will avoid the problem of losing updates
NO
could lose updates
NO
msgs can be proecessed in any order, no guarantee

Amazon AWS IoT Javascript API

Dynamic CPS/IoT Verification

Goal: Creating a full suite of verification tools for checking of state update orderings and atomicity in distributed systems.

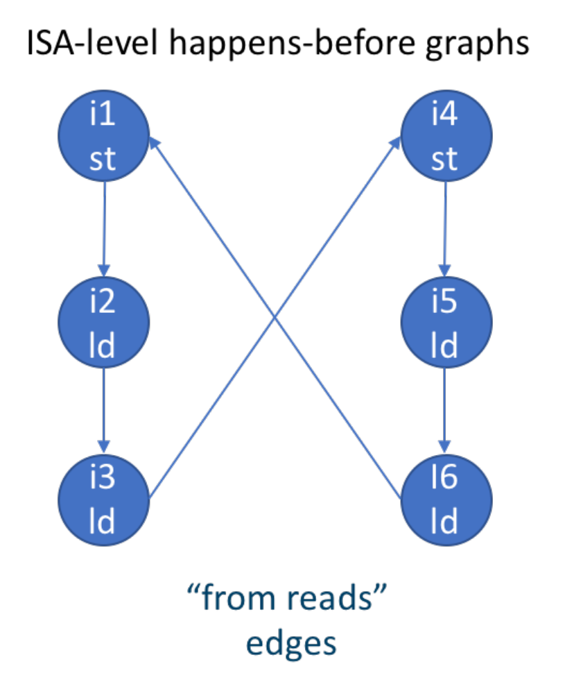
General approach: Apply computer architecture concept of “litmus tests” for verifying correctness of memory consistency implementations to CPS/IoT systems.

Cycle = Forbidden outcome, No Cycle = Allowed outcome

	Observable	Unobservable
Permitted	OK	OK (stricter than necessary)
Forbidden	BUG	OK

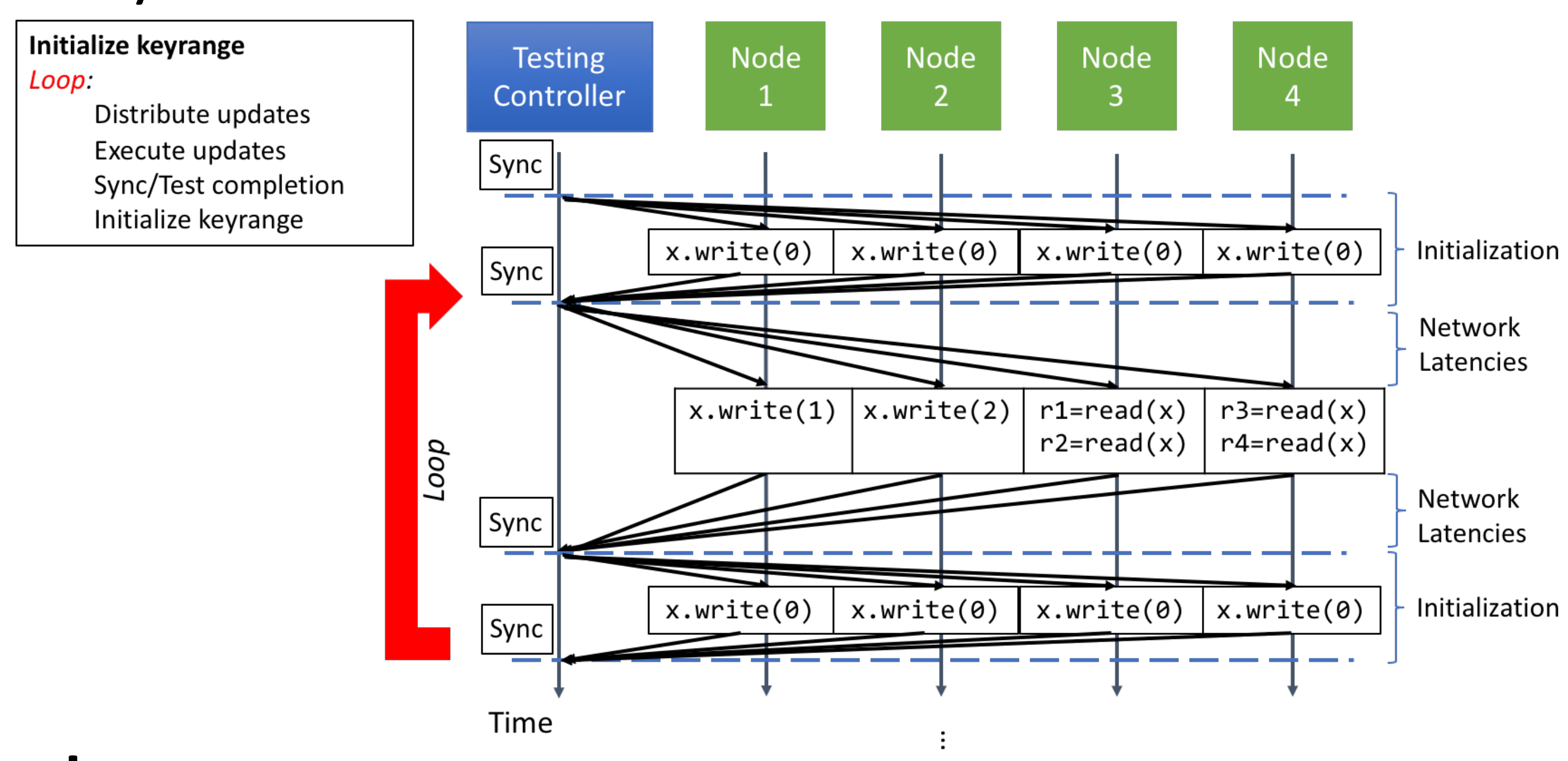
Core 1	Core 2
(i1) st[x] <- 1	(i4) st[y] <- 1
(i2) ld[x] -> r1	(i5) ld[y] -> r3
(i3) ld[y] -> r2	(i6) ld[x] -> r4
Forbidden Outcome: r1=1,r2=0,r3=1,r4=0	

Initially: [x]=[y]=0, We assume sequential consistency



Litmus tests are targeted tests that access data, looking for outcomes that are not allowed under specific guarantees. Repeated runs of the tests ensure an implementation is tested with many different timings.

We set out to develop a tool (Musli Tool) that can create and execute such distributed systems litmus tests.



Musli Tool:

- Adapts litmus tests to distributed key-value stores and distributed consistency models where execution and storage of a client are decoupled and updates are not local operations.
- Repeatedly executes tests, executing updates simultaneously and performing initialization before each new test.

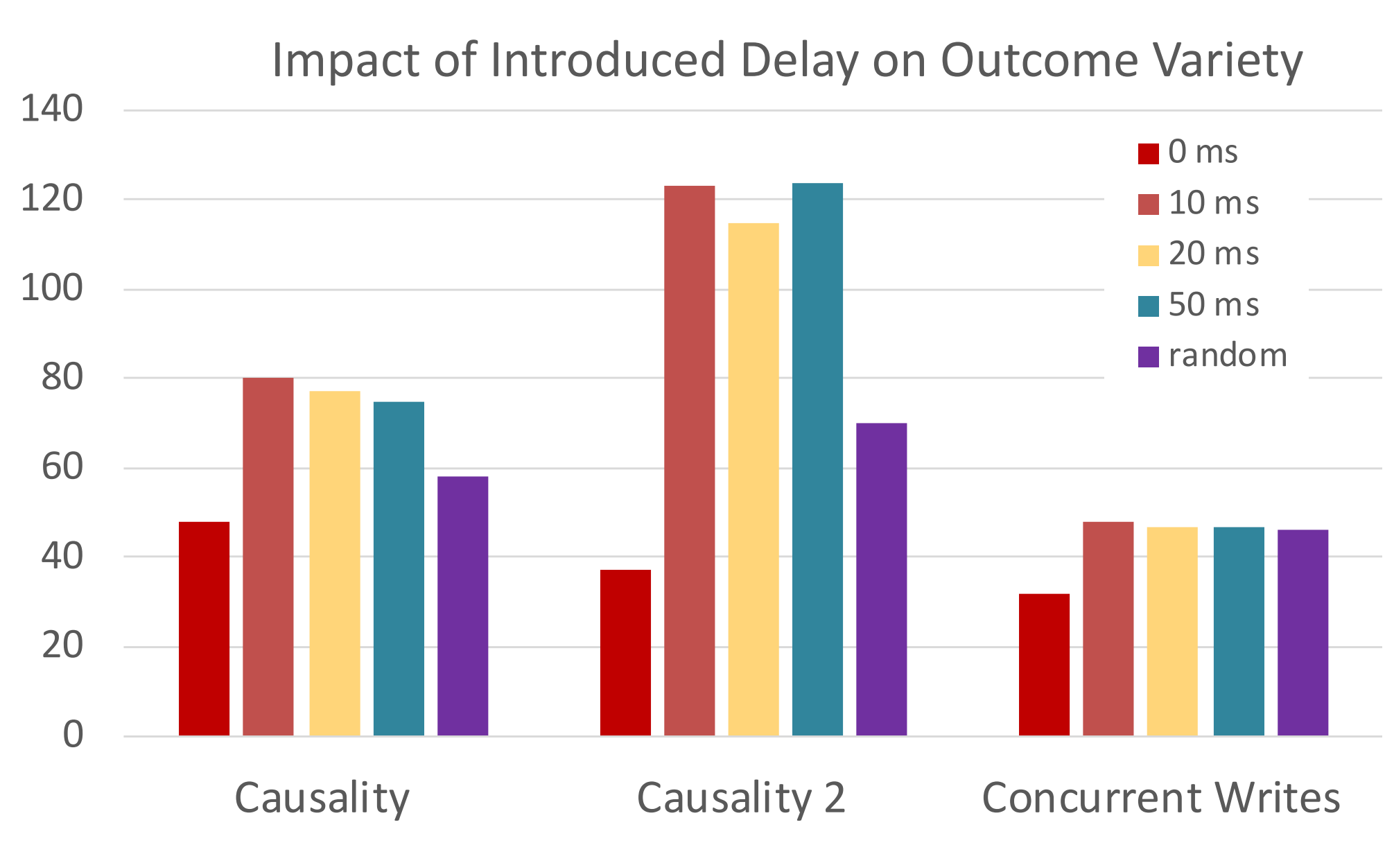
Created Distributed Litmus Tests:

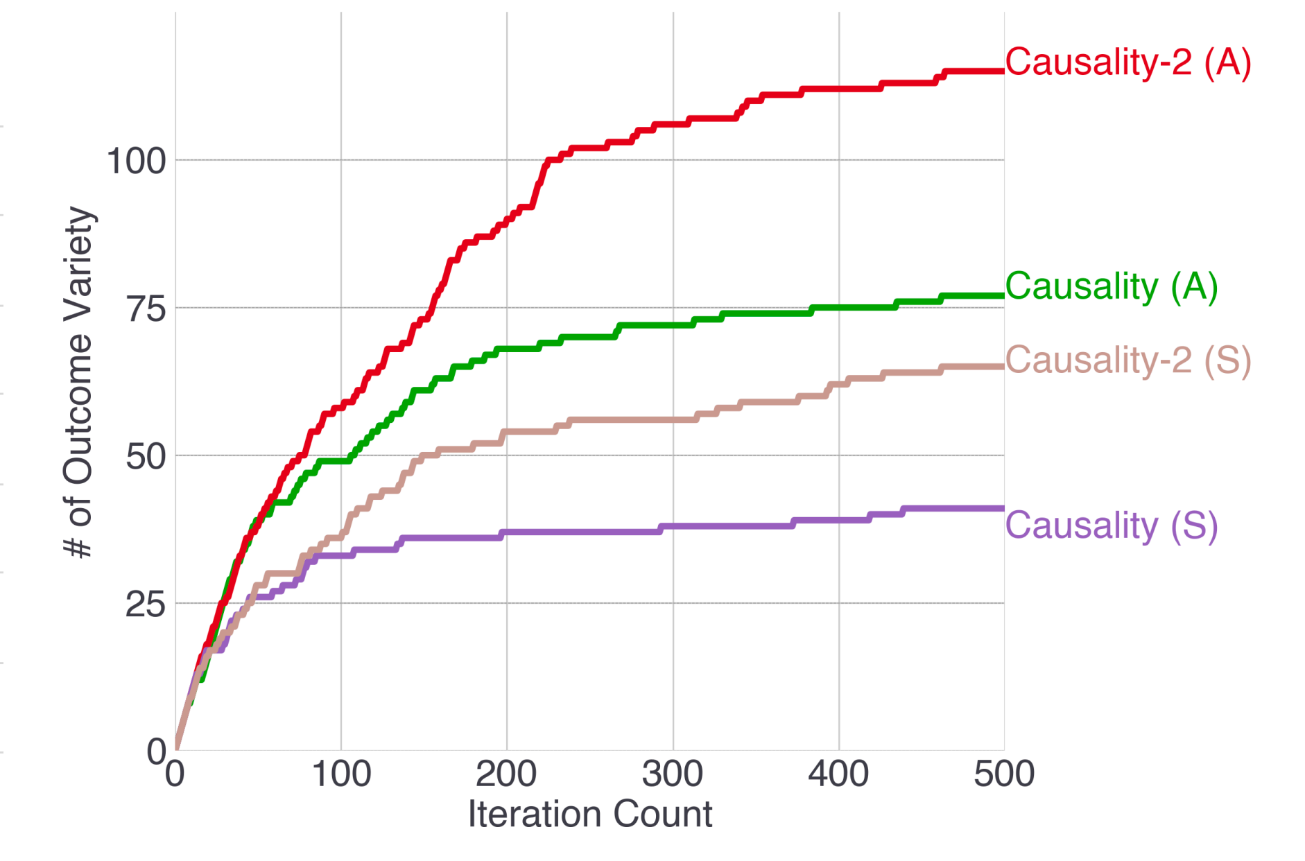
DLT Name	Sequential Consistency	Combinatorial
Causality	83	243
Causality 2	138	256
Concurrentwrites	47	81
Stale Reads	7	8
CO-MP	3	4
Load Buffering	3	4
Store Buffering	3	4
Message Passing	6	9
Monotonic Reads	1	9
Monotonic Reads 2	2	4
Read Your Own Writes	1	3

Execution Parameter	Parameter Values
Client Placement	local/non-local
RPC Request Type	synchronous/asynchronous
Delay	0ms/10ms/20ms/50ms/random
Cassandra Consistency Level	ONE/QUORUM

Left: 11 Distributed Litmus Tests were generated. Maximum number of outcome variety based on Sequential Consistency and Combinatorically are shown. **Above:** Client placement with respect to data nodes, RPC Request type, different Cassandra Consistency Levels as well as delays to data updates were introduced as possible features of a given system.

Execution and Analysis of the Litmus Tests:





Demonstrating the variability of outcome variety obtained through Musli Tool based on data update delay (left) and RPC Request type (right: synchronous (S), asynchronous (A)).

Conclusions

- Correctness and security are difficult to achieve in CPS/IoT and must rely on appropriate static and dynamic verification techniques, as security attacks often exploit design mistakes and their implications.
- Our work has created static and dynamic verification techniques to support these correctness and security goals.