On Synthesis of Runtime Security Monitor for Real-time IoT Applications

Muhammad Taimoor Khan
School of Computing and Mathematical Sciences
University of Greenwich, London, UK
m.khan@gre.ac.uk

Dimitrios Serpanos
Industrial Systems Institute, RC-ATHENA and
ECE, University of Patras, Patras, Greece
serpanos@ece.upatras.gr

Howard Shrobe
CSAIL, MIT
Cambridge, USA
hes@csail.mit.edu

We report on the evaluation of an efficient and scalable verification method that assures runtime security of real-time Internet of Things (IoT) applications with strict performance requirements. Modeling known threats at abstract but practical level is key to assure security of efficient and scalable IoT applications. To this end, the method *classifies* set of known attacks into *computational*, *data integrity* and *communication* attacks. Then, each attack class is *decomposed* into *declarative properties* and *definitive properties*. A *declarative property* specifies an attack as a single big-step relation between initial and final state by ignoring intermediate states, while a *definitive property* specifies an attack as a sequence of numerous small-step relations considering all intermediate states between initial and final state. Finally, the declarative properties are translated into runtime security monitor that assures protection of the application execution against known threats without defying runtime application's performance requirements. in particular, we evaluate the methodology through its application to an energy management application of smart home.

Runtime assurance, smart home, energy management, computational attacks, data integrity attacks, communication attacks

1. INTRODUCTION

Modern IoT applications aim at providing on demand services in various critical application domains, e.g. smart homes, health-care and automotive vehicles, to name a few. Typically, these services are implemented as embedded software applications that remotely and automatically control the operations of IoT devices according to the demand as determined and monitored by the underlying (devices') sensors. The security and safety of such applications cannot be established mainly because of the unreliable sensors, which may provide incorrect data either due to their malfunctioning or due to an accidental (by privileged user) or intentional (by adversary) interference. Therefore, the incorrect sensor input data may identify an inaccurate demand, which may result in a serious threat or a significant loss, e.g. financial loss for incorrect energy usage - in smart home, threat to a patient's life - in health-care. To ensure safety and security of energy management IoT applications for smart home, current approaches either employ encryption based techniques Zhang (2018); Tiloca (2017) that assure correct data exchange by sensors or employ data analysis techniques that monitor sensor data and alarm when some unusual value is detected Soudan (2018). Alternatively, some applications employ improved access control mechanisms that ensure that controller decisions are consistent with sensor input data. However, such approaches fail to detect stealthy attacks that compromise system integrity, e.g. bad data (false data injection) and bad computations because they do not comprehend what the application or IoT device is trying to do. To this end, various run-time verification methods have been proposed.

Contemporary run-time verification based monitors assure that an application execution is consistent with design specification of the application and raises an alarm when an inconsistency is detected Khan (2016). However, despite considerable progress in this area Cassar (2017), these methods cannot be directly applied to real-time control applications mainly because they do not consider behavior information of an IoT process to detect the threats Li (2011). Furthermore, such methods do not compare application execution with the large number of known threats (approx. 588 attack patterns - CAPEC List (2019)). Clearly, comparison of the application execution with these large number of threats in addition to the design specification requires significant verification time, which may hinder the performance requirements of the application resulting in unpleasant incidents.

Recently, several approaches aimed at improving runtime verification performance by reducing number of attacks that have to be monitored at runtime by considering execution environment of the application. For instance, the program developed in a specific language is free of various classes of threats and vulnerabilities, e.g., a web application developed in the UrWeb language Chlipala (2016) is free from code injection attacks. In another effort Barrho (2018), compiler operations have been verified that assure that the compilation is vulnerability free, for instance, the C compiler (CompCert).

Lately, an implementation of various processor operations have been verified to show absence of security threats, e.g. ARM Chen (2016). These developments assure that certain classes of vulnerabilities and security threats cannot happen during application execution and thereby significantly reduce run-time verification overhead. However, these approaches handle specific vulnerabilities and threats on one hand and are language and platform specific on the other hand. Importantly, such approaches do not consider critical known attacks to applications, e.g., stealthy and insider attacks. Therefore, they cannot be directly applied to critical control applications, e.g. energy management and health-care control applications, due to their strict performance requirements and effects of their critical functionality when attacked.

We consider smart home as a critical control application domain with strict performance requirements Li (2011). Recent developments in IoT devices have improved security and safety of a variety of smart home applications. In contrast to typical control systems, security threats to these applications result in significant losses and, therefore, require to be monitored and prevented with high assurance without compromising their performance. For instance, energy monitoring devices are used for the management of energy in smart homes. The components of such systems are wirelessly connected, e.g., the energy monitor, energy distributor, and remote control that forms a real-time monitoring and feedback loop. Typically, these systems are vulnerable due to unreliable wireless and sensor based communication. An adversary can easily launch attacks to such applications that may result in significant financial loss, for instance, by sending incorrect value of required energy by various appliances wirelessly, by compromising the command to the energy distributor remotely that may stop the energy distribution, or by distributing energy with undesired (i.e., very high) current.

In contrast to above-mentioned threat detection mechanisms, we evaluate a verification method that ensures run-time security and safety of energy management application of an IoT based smart home. The method monitors the execution of control application based on the process (application) behavior and known attacks. The behavior of the application includes functional and nonfunctional (e.g., security, performance) properties of various components that manages the IoT control process Khan (2018). The methodology allows to specify set of known threats to the application alongside with functional specification. Based on ARMET Khan (2018), we assure that design of the control application is free of specific class of attacks and vulnerabilities. To reduce runtime verification overhead, the method classifies the known attacks into *computational*, *data integrity* and *communication* attacks. Next, we specify *declarative* and *definitive properties* of each class. In detail, declarative properties are specified as a one big-step relation between initial (i.e., pre-conditions) and final state (i.e. post-conditions) of an logically constituted operation. Finally, we argue that a security monitor can be generated from the specification of *declarative properties*. The generated monitor assures that the application execution is protected against known attacks without compromising real-time performance requirements of the application. Further details of the method are discussed in Khan (2019).

The rest of the paper is organized as follows: Section 2 elaborates modeling of known threat as discussed in Khan (2019), while Section 3 sketches generation of run-time security monitor. Importantly, the contents of the following sections are based on the material from the original approach Khan (2019).

2. MODELING KNOWN THREATS

In this section, we sketch the approach based on the material from Khan (2019). Modeling the behavior of IoT system applications and known threats is a complex task due to varying behavior of IoT devices and components. The complexity is directly proportion to the required run-time verification time. Therefore, to handle the complexity, we simplify the modelling task by specifying the threats in *declarative* and *definitive* properties, where the former specify an attack class as a single big-step relation between an initial state and final state of the operation (ignoring the intermediate details) and the latter specifies each attack uniquely as a composition of many small-step relations that correspond to intermediate states.

We demonstrate the concept of declarative and definitive properties by modeling the controller for managing various energy levels.

Example

Consider an energy control application of an IoT smart home, which has a sensor that monitors energy consumption and an energy controller that receives and issues commands for the management of energy monitor. The main task of the controller is to distribute desired amount of energy automatically as per following rules:

1. Get the new energy level through sensor (l_t at time t)
2. Compute the rate of distribution of energy (r_t) based on
   - new energy level (l_t) and some previous level (say k) values ($L = \{l_{t-k}, l_{t-k-1}, \ldots, l_{t-1}\}$)
3. Compute the new energy to be distributed (e_t) based on
   - computed rate of distribution (r_t), current distribution (l_t) and previously (say k) distributed energies ($E = \{e_{t-k}, e_{t-k-1}, \ldots, e_{t-1}\}$)
4. Distribute the computed energy (e_t).

The control application implements each of the above step as a module that is specified by corresponding pre and post-conditions and invariant. Our goal is to protect the application execution against variants of a command manipulation attack Li (2011) that is a *computational attack*. A typical command manipulation attack can be launched by performing the following two tasks:

1. illegitimate access to the relevant resources
2. modify a command (i.e. instructions that implement the rules) to achieve the malicious goal.

As an example, we consider the command that computes the new energy distribution as per rule 3 above. We formalize the attack into declaration and definitive properties in the following subsections.

2.0.1. Declarative Properties

The *declarative* properties of the attack can be formalized as follows

$$\langle [\![c]\!], \sigma, \rho\rangle \curvearrowright \langle \alpha, \sigma', \rho'\rangle$$

which says that an application (i.e., command sequence) execution $[\![c]\!]$ with the given state σ, run-time environment ρ may yield a state σ' and environment ρ' with an attack α in a single step. Here σ is a memory store (i.e., a set of pairs of variables and its values), ρ is a run-time environment (i.e., a set of pairs of identifier and its types).

Based on the above formulation, we formalize the example attack as follows:

$$\frac{\begin{array}{l} e_t = computeEnergy(r_t, l_t, D) \\ \alpha == \text{"}A\text{"} \Leftrightarrow \neg safe(e_t) \end{array}}{\langle computeEnergy(r_t, l_t, D), s, re\rangle \curvearrowright \langle \alpha, s', re'\rangle}$$

where

$$\begin{array}{rl} safe(e_t) \Leftrightarrow & e_t \leq max\_single\_energy \wedge \\ & today\_energy + e_t \leq max\_day\_energy \\ & l_t < max\_safe\_lvl \end{array}$$

The rule states that execution of an application component ($computeEnergy$) in state s (and environment re) may yields state s' (and environment re') with an attack class $\alpha(A)$ iff the critical conditions ($safe$) are violated, namely when

1. new computed energy distribution is greater than the maximum allowed single distribution (i.e. equals maximum load of wires in home) or
2. sum of new computed energy and cumulative energy for today is greater than the maximum allowed distribution of energy in a single day or
3. when measured distribution level is extremely high; This property can be both be safety and security threat.

The formalization assures detection of an attack without considering intermediate steps of the attack, namely either the output value e_t has updated or implementation of rule 3 has been modified. This allows to verify within real-time constraints, if the application is under attack. When an attack is detected, the application runs in fail-safe mode and attempts to determine the exact attack.

2.0.2. Definitive Properties

The definitive properties formalize an exact attack of the class (as identified by declarative properties). The *definitive* properties can be formalized as follows

$$\langle [\![c]\!], \sigma, \rho\rangle \curvearrowright_* \langle \alpha, \sigma', \rho'\rangle$$

which says that an application (i.e., command sequence) execution $[\![c]\!]$ with the given state σ, run-time environment ρ may yield a state σ' and environment ρ' with an attack α in many small-steps.

Based on the above formulation, we formalize the variants of example command manipulation attack as follows:

$$\frac{\begin{array}{c} e_t = computeEnergy(r_t, l_t, D) \\ p = diagnose(computeEnergy(r_t, l_t, D), inv(e_t)) \\ \text{IF } p == \text{START THEN } \alpha = "A_1" \\ \text{ELSE IF } p == \text{END THEN } \alpha = "A_3" \quad \text{ELSE } \alpha = "A_2" \end{array}}{\langle computeEnergy(e_t, l_t, D), s, re\rangle \curvearrowright_* \langle \alpha, s', re'\rangle}$$

where

$$diagnose(cmd, cnd) : \text{INSTRUCTION} \cup \text{START} \cup \text{END}$$

returns the position of an instruction in the program (cmd) which has violated the given condition (cnd). The position can be START, i.e. just before the method (body), END, i.e. just after the method (body) or any INSTRUCTION of the method body.

The rule states that an application execution of module $computeEnergy$ in state s (and environment re) may yield state s' (and environment re') with an exact attack α s.t.

1. if α is A_1, then measured distributed level has been compromised or threatened,
2. if α is A_2, then result of the function (shared through sensor) has been compromised only and
3. if α is A_3, then some instruction has been compromised/modified.

In the following subsection, we argue about method to generate run-time security monitor based on the formalization above.

3. SECURITY MONITOR

Based on the material from Khan (2019), in this section, we sketch the approach for synthesis of the security monitor from the previously developed specification. The monitor ensures that smart home control application is secure against known attacks (e.g., command manipulation in current example) at run-time without hindering its performance requirement so as to ensure negligible damage.

Based on deductive synthesis Khan (2018), we derive security monitor through refinement of the attack models (i.e., declarative properties) w.r.t. application specification. The refinement starts with an initial (non-deterministic) attack model with concealed performance requirements. Then, the model is synthesized through step-wise and interactive refinements, where each refinement optimizes certain model formulation in a way that no extra behavior is introduced on one hand, and none of the security properties are violated on the other hand. Finally, the model is translated into a fully deterministic implementation of a monitor that is correct w.r.t. specification of attack classes and is also secure and efficient, respecting real-time performance constraints by employing efficient representations and algorithms.

Example

To evaluate the methodology Khan (2019), we generate a security monitor for our running example of energy controller. For simplicity, we generate monitor in a familiar notation, i.e. Java-like syntax as shown in Listing 3.

For our example controller specification (Listing 1), in each specific period of time (say 10 seconds), reads energy level of the (see l.12) and then either the controller DISTRIBUTEs some amount of energy to the connected devices (see l.70), or does NOTHING (see l.71). Initially, the energy level is undefined (see l.6). At any specific interval, the new energy level is read, if the energy level (i.e. reading) is in the range of sensor accuracy (see l.13), then we either accept the value or any value (see l.15). After each specific interval of time (see l.69), the controller either

- DISTRIBUTEs energy, i.e. the controller first computes the desired amount of energy such that energy level becomes safe (see l.45-58) and then distributes the desired amount of energy to the devices (see l.70),
- or does NOTHING, i.e. the controller concludes that the energy level of the devices does not require any more energy (see l.45-58) and thus does NOTHING (see l.71).

Considering the command manipulation attack, the specification describes the following two possible attack scenarios:

1. *data integrity attacks* (case A_1) – in which the command parameter is compromised or is critically high, i.e. the energy (sensor) distribution gets compromised (see l.68) and (case A_2) – in which the sensor based measured energy level is compromised (see l.42)

```
1  public enum Action {DISTRIBUTE, NOTHING}
2  public enum Rate { DECREASING, NORMAL, INCREASING}
3  public enum Level {FALLING, NORMAL, RISING}
4
5  class EnergyControllerSpec {
6    private int enrgy_lvl = -1;
7    private ArrayList<int> enrgy_dists= new ArrayList
       <int>();
8    private ArrayList<int> enrgy_lvls= new ArrayList<
       int>();
9    private final int N = 2;
10   ...
11
12   public void readEnergy(int reading){
13     if(abs(reading) > SENSOR_ACCURACY)
14      if(enrgy_lvl > 0)
15      enrgy_lvl = { (reading | True)}; }
16
17   // computing effect of energy
18   public Level computeLevel(){
19     Level lvl =
20     { v | (v = FALLING ->  enrgy_lvl < enrgy_lvls[N
       ])
21      /\ (v = NORMAL ->  enrgy_lvl == enrgy_lvls[N])
22      /\ (v = RISING ->  enrgy_lvl > enrgy_lvls[N])
       };
23      return lvl;
24   }
25
26   // computing rate of energy distribution
27   public Rate computeDistributionRate(){
28     Rate rate =
29     { r | (r = DECREASING ->
30      (enrgy_lvl-enrgy_lvls[N])<(enrgy_lvls[N]-
       enrgy_lvls[N-1]))
31      /\ (v = NORMAL ->
32      (enrgy_lvl-enrgy_lvls[N])==(enrgy_lvls[N]-
       enrgy_lvls[N-1]))
33      /\ (v = INCREASING ->
34      (enrgy_lvl-enrgy_lvls[N])>(enrgy_lvls[N]-
       enrgy_lvls[N-1]))};
35      return r;
36   }
37
38   // computing next distribution of energy
39   public int computeEnergy(){
40
41     // attack A1 specification
42     enrgy_lvl < max_safe_lvl
43
44     // computing energy distribution
45     int enrgy_dist = 0;
46     Level lvl = computeLevel();
47     Rate r = computeRate();
48
49     if(enrgy_lvl <= max_safe_lvl){
50      if(rate == DECREASING) enrgy_dist = 0;
51      else if(rate == INCREASING)
52       enrgy_dist = min_single_dist+enrgy_dists[N
       ]/3;
53     }else{
54      if(lvl == RISING)
55         enrgy_dist = min_single_dist+enrgy_dists[N
       ]/3;
56      else if(lvl == NORMAL || lvl == FALLING)
57         enrgy_dist = min_single_dist;
58      else enrgy_dist = 0;
59     }
60
61     // attack A3 specification
62     enrgy_dist <= max_single_dist && enrgy_dist +
       today_dists <= max_day_dist
63     return enrgy_dist; }
64
65      // send command to controller for energy
       distribution
66     public Action distributeEnergy(int enrgy_dist){
67       // attack A2 specification
68       enrgy_dist + today_dist <= max_day_dist
69       Action act =
70       { a | (a = DISTRIBUTE -> enrgy_dist > 0)
71       /\ (a = NOTHING -> enrgy_dist = 0)};
72     }
73 }
```

Listing 1: Energy Controller + Attack Specification

2. *computational attack* (case A_3) – in which the command computational code is compromised, i.e. the instruction(s) in the implementation is modified (see l.62) such that they now compute energy values, which are either undesired for the devices or damaging to the network.

Based on various design choices, we synthesize the specification (Listing 1) to derive the Java implementation for the controller (Listing 2) and for the corresponding security monitor (Listing 3).

The controller implementation corresponds to the controller specification. Additionally, the controller implementation (Listing 2) has calls (see l.60 and l.53) to monitor (Listing 3) which eventually enables the security monitor to detect data attacks – A_1 and A_2 (see l.7 and l.23) and a computational attack – A_3 (see l.14). The deductive synthesis assures that the controller implementation is correct and secure by construction with respect to the specification of controller and attacks. Clearly, the monitor is capable of rigorously detecting any arbitrary data and computational attack in embedded software based controllers and thus highly assuring the safety and security of patients.

4. CONCLUSION

We have argued that the evaluated approach for runtime verification is effective. The approach allows to model known threats that can later be synthesized into runtime security monitor that assures the security of application execution against the threats (i.e. data integrity or false data injection and insider attacks) strictly respecting application's efficiency requirements. In future, we plan to build verification methods to detect unknown (i.e., hypothetical) attacks to critical control applications to assure more accurate medical diagnosis and other operations.

ACKNOWLEDGEMENT

Dimitrios Serpanos acknowledges that part of this work that was performed at ISI/ATHENA was supported by the project "I3T - Innovative Application of Industrial Internet of Things (IIoT) in Smart Environments" (MIS 5002434) which is implemented under the "Action for the Strategic Development on the Research and Technological Sector", funded by the Operational Programme "Competitiveness, Entrepreneurship and Innovation" (NSRF 2014-2020) and co-financed by Greece and the European Union (European Regional Development Fund).

```
...
class EnergyControllerCode {
  private int enrgy_lvl = -1;
  private ArrayList<int> enrgy_dists=new ArrayList<
     int>();
  private ArrayList<int> enrgy_lvls=new ArrayList<
     int>();
  private final int N = 2;
  ...

  public void readEnergy(int reading){ enrgy_lvl =
     reading; }

  // computing effect of energy
  public Level computeLevel(){
   if(enrgy_lvl < enrgy_lvls[N])         return
     Level.FALLING;
   else if(enrgy_lvl == enrgy_lvls[N]) return Level
     .NORMAL;
   else if(enrgy_lvl > enrgy_lvls[N]) return Level.
     RISING;

        return Level.NORMAL;
  }

  // computing rate of energy change
  public Rate computeRate(){
  if((enrgy_lvl-enrgy_lvls[N])<(enrgy_lvls[N]-
     enrgy_lvls[N-1]))
        return Rate.DECREASING;
  else if((enrgy_lvl - enrgy_levels[2]) >= (
     enrgy_levels[2] - enrgy_levels[1]) )
        return Rate.INCREASING;
  else  return Rate.NORMAL;
  }

  // computing next distribution of energy
  public int computeEnergy(){

    // monitor attack A1, if sensor input is
     compromised
    monitorComputeEnergyPre(enrgy_lvl);

    // computing energy distribution
    int enrgy_dist = 0;
    Level lvl = computeLevel();
    Rate rate = computeRate();

    if(enrgy_lvl <= max_safe_lvl){
     if(rate == DECREASING) enrgy_dist = 0;
     else if(rate == INCREASING)
        enrgy_dist = min_single_dist+enrgy_dists[N
     ]/3;
    }else{
     if(lvl == RISING)
        enrgy_dist = min_single_dist+enrgy_dists[N
     ]/3;
     else if(lvl == NORMAL || lvl == FALLING)
        enrgy_dist = min_single_dist;
     else enrgy_dist = 0;
    }

    // monitor attack A3, if computation is
     modified
    monitorComputeEnergyPost(enrgy_dist);

    return enrgy_dist; }

    // send command to controller for distributing
     given energy
    public Action distibuteEnergy(int enrgy_dist){
      // monitor attack A2, if sensor input is
     compromised
      monitorDistributeEnergyPre(enrgy_dist);
      if(enrgy_dist != 0) return DISTRIBUTE;
      else return NOTHING;
    }

}
```

Listing 2: Energy Controller Code

```
class Monitor {

   private int max_safe_lvl = 300;
   ...

   // monitoring attack scenario A1
   // if measured enrgy level is malicious
   public bool monitorComputeEnergyPre(int
     enrgy_lvl){
        if(enrgy_lvl >= max_safe_lvl){
         suspendExecution(); return false; }
         return true;   }

    // monitoring attack scenario A2,
    // if command (distribution amount) has been
     modified
    public bool monitorDistributeEnergyPre(int
     enrgy_dist){
     if (enrgy_dist + today_dists > max_day_dist){
       suspendExecution(); return false; }
       return true;
   }

   // monitoring attack scenario A3,
   // if computations have been modified
   public bool monitorComputeEnergyPost(int
     enrgy_dist){
   if (enrgy_dist > max_single_dist ||
    enrgy_dist + today_dists > max_day_dist){
     suspendExecution(); return false; }
     return true;
   }
}
```

Listing 3: Security Monitor Code

REFERENCES

W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu (2018) *Homonit: Monitoring smart home apps from encrypted traffic*, in Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '18. New York, USA: ACM, pp. 1074–1088.

M. Tiloca, K. Nikitin, and S. Raza (2017, *Axiom: Dtls-based secure iot group communication*, ACM Trans. Embed. Comput. Syst., vol. 16, no. 3, pp. 66:1–66:29.

M. B. Soudan, H. M. Al Rifaie, T. M. Asmar, and S. Majzoub (2018) *Smart home energy management system: An exploration of iot use cases*, in Advances in Science and Engineering Technology International Conferences (ASET), pp. 1–5.

M. T. Khan, D. Serpanos, and H. Shrobe (2016) *A rigorous and efficient run-time security monitor for real-time critical embedded system applications*, in IEEE 3rd World Forum on Internet of Things (WF-IoT), pp. 100–105.

I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir (2017) *A survey of runtime monitoring instrumentation techniques*.

N. Li, A. Raghunathan (2011) *Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system*, in IEEE 13th International Conference on e-Health Networking, Applications and Services, pp. 150–156.

CAPEC, Online (2019). Available: `https://capec.mitre.org`

A. Chlipala (2016) *Ur/web: A simple model for programming the web*, Commun. ACM, vol. 59, no. 8, pp. 93–100.

D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, and S. Blazy (2018) *CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler*, in ERTS2 2018 - Embedded Real Time Software and Systems. Toulouse, France.

A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi (2016) *End-to-end verification of arm processors with isa-formal*, in Computer Aided Verification, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, pp. 42–58.

M. T. Khan, D. Serpanos, and H. Shrobe (2018), *Armet: Behavior-based secure and resilient industrial control systems*, Proceedings of the IEEE, vol. 106, no. 1, pp. 129–143.

M. T. Khan, M. Pinzger, D. Serpanos, and H. Shrobe (2019) *Runtime protection of real-time critical control applications against known threats IEEE Design & Test*, no. In Review.

B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala (2015) *Fiat: Deductive synthesis of abstract data types in a proof assistant* in Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, Mumbai, India, pp. 689–700.