Runtime Protection of Real-Time Critical Control Applications Against Known Threats

Muhammad Taimoor Khan
University of Greenwich

Martin Pinzger
Alpen-Adria University

Dimitrios Serpanos
University of Patras

Howard Shrobe
Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory (MIT CSAIL)

Editor's notes:
With increasing complexity, connectivity, and programmability of embedded CPS devices, the potential cyber-attack surface has also been increasing making the study of related cyber-security issues highly relevant and timely. This article provides an efficient verification method for runtime applications with tight performance specifications. By considering classification and the declarative and definitive properties of different types of attacks, it is shown that a runtime security monitor can be generated to assure the security of application execution against known threats given the performance requirements. The methodology is applied in a healthcare application of insulin pump to regulate blood sugar level.
—Farshad Khorrami, New York University

Conventional runtime verification-based security monitors assure that the execution of application is consistent with the functional specification of the application and raise an alarm when some inconsistency is detected [1]. To meet strict performance requirements of real-time applications, such verification methods either compute execution time bounds [e.g., worst-case execution time (WCET), best-case execution time (BCET), and average-case execution time (ACET)] of the applications and argue that monitoring overhead along with the execution bounds meets the performance requirements [2], [3] or propose such formalism for specification that can be translated into efficient and parallel hardware blocks, whose execution meets the performance requirements [4]. However, despite considerable progress achieved in the past [3], such verification methods are not directly applicable to real-time control applications mainly because they employ malware and intrusion detection mechanisms which suffer from high rate of false alarms, on the one hand, and do not consider cyber–physical process information to detect threats, on the other hand [5]. Such methods do not rigorously compare application execution with the known threats (which are approx. 600 attacks as reported by CAPEC—https://capec.mitre.org). Obviously, on-the-fly comparison of the execution of application with these threats in addition to the functional specification requires significant verification time, which may violate the real-time performance requirements of the application resulting in serious damages.

More recently, runtime verification performance has been improved by reducing the number of attacks to be monitored at runtime by considering runtime particularities of the application. For instance, developing an application in some language that assures security by design, that is, the program is free of certain class of vulnerabilities and threats, for example,

Digital Object Identifier 10.1109/MDAT.2020.3007729
Date of publication: 7 July 2020; date of current version: 25 November 2020.

Copublished by the IEEE CEDA, IEEE CASS, IEEE SSCS, and TTTC

developing web applications in the UrWeb language [6] assures that the application is free from code injection attacks and invalid HTML returns, to name a few. Furthermore, compiler operations have also been verified that assure that the compilation is free from being vulnerable, for instance, the C compiler (CompCert) [7]. Finally, verification of an implementation of various processor operations also assures the absence of some security threats, for instance, ARM [8]. All these developments assure the absence of certain class of vulnerabilities and security threats in application implementations, which significantly reduces runtime verification overhead. However, these results are application-, language-, and platform-specific, on the one hand, and consider specific software vulnerabilities, on the other hand. These results, however, are not sensitive to attacks to applications, for example, stealthy and insider attacks. Therefore, they cannot be adapted to critical control applications, for example, healthcare control applications, due to their strict performance and characteristics of their critical functionality and known attacks.

A critical application domain for real-time control applications with strict performance requirements is healthcare control systems [5]. Recent developments in wearable medical devices have improved diagnosis, monitoring, and therapy for a variety of medical conditions. In contrast to typical control systems, security threats to these devices have severe consequences and, therefore, require to be monitored and prevented with high assurance without compromising their performance. For instance, glucose monitoring and insulin pump are used for the treatment of diabetes. The components of such systems are wirelessly connected, e.g., the glucose monitor, insulin pump, and remote control, forming a real-time monitoring and feedback loop. Such systems are highly vulnerable due to wireless and sensor-based communication. An adversary can easily launch attacks to such control systems that can threaten the life of the patient, for instance, by sending incorrect blood glucose results to the insulin pump wirelessly, by compromising the command to the insulin pump remotely that stops the insulin injection, or by injecting insulin with undesired (i.e., very high) dose.

A security monitor for insulin pump [9] has been developed that focuses on handling threats arising from wireless communication of the pump. In another effort, Klonoff [10] has developed a security monitor that handles unauthorized access to the insulin pump. In contrast to the aforementioned detection mechanisms, we introduce a design methodology for monitoring real-time control applications based on the process (application) behavior and known attacks. The behavior of the application is the set of functional and nonfunctional (e.g., security and performance) characteristics of both the cyber and the physical components of the control process [11]. In addition to the behavior, the methodology also allows to specify a set of known threats to the application as shown in Figure 1. On the basis of ARMET [11], we assure that the design of critical control application is free of certain class of attacks and vulnerabilities. Furthermore, to assure that on-the-fly verification meets the performance requirements, we first classify the known attacks into computational, data integrity, and communication attacks. Then, we model declarative and definitive properties of each class. The former can be specified as a one big-step relation between initial (also known as preconditions) and final states (also known as postconditions). Finally, from the specification of declarative properties, we generate security monitor to ensure that the application execution is protected against known attacks respecting real-time performance requirements.

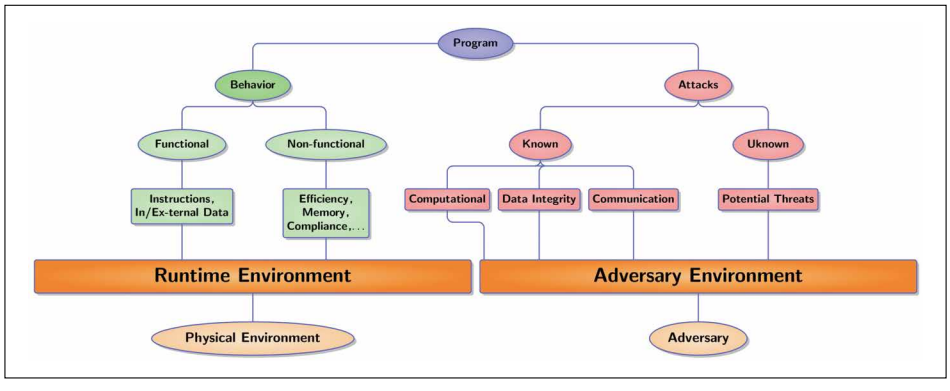

Figure 1. Full program view for CyberSecurity.

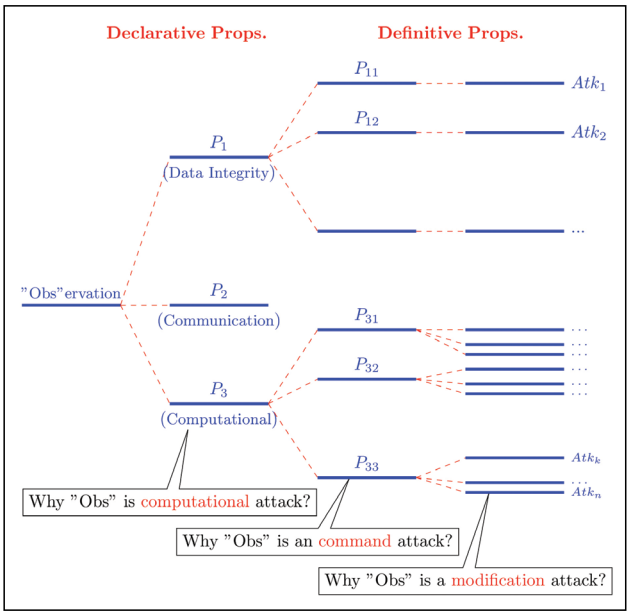

Figure 2. Attack model—command modification attack.

We have implemented a prototype monitor and demonstrate its efficient detection of declarative properties as compared to definitive properties.

Modeling known threats

Modeling the behavior of cyber–physical system applications and known threats models is complex due to fundamentally different characteristics of cyber and physical components. The complexity of the involved models significantly increases the runtime verification time. Therefore, to manage the complexity, we simplify the complex models by describing them at desired but pragmatic levels of abstraction which can be verified efficiently.

To model such threats (see Figure 2), we do the following:

- First, we classify the known attacks into
  - *computational attacks* that may change instructions or internal data of an application execution;
  - *data integrity attacks* that may modify the external input data to the system;
  - *communication attacks* that may access (respectively use) network resources by illegitimate means.
- Then, we model each attack class into its
  - *declarative properties* that specify sufficient conditions of an attack class as one big-step relation between an initial state and final state without considering the intermediate details;
  - *definitive properties* that specify each attack in the class uniquely as a composition of a sequence of small-step relations which corresponds to all intermediate states between initial and final states.

These properties can be viewed as big-step and small-step operational semantics of languages, respectively, where big-step is a unified model of the behavior of semantically similar language constructs, whereas small-step is a collection of models of the behavior of each construct. Furthermore, the declarative properties can be seen as a quick filter for attacks, whereas the definitive properties can be seen as a diagnostic engine [11]. In detail, declarative properties ensure the presence of an attack as efficient as real-time performance constraints of the application without considering intermediate details. Once identified, the system goes into a safe-operational mode and meanwhile the definitive properties diagnose the exact attack considering all intermediate details revealing all associated compromises of the attack.

We demonstrate the concept of declarative and definitive properties by modeling an insulin pump for managing sugar levels of diabetics.

Example

Consider a healthcare control application, which has a sensor that monitors glucose and an insulin (control) pump that receives and issues commands for the management of glucose monitor and insulin pump. The main task of an insulin pump is to inject an appropriate amount of insulin automatically as per the following algorithm:

1) Read the new glucose level through sensor (l_t at time t).
2) Compute the rate of glucose (r_t) based on new glucose 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 insulin dose (d_t) based on computed glucose rate (r_t), current glucose level (l_t), and previous (say k) injected insulin values ($D = \{d_{t-k}, d_{t-k-1}, \ldots, d_{t-1}\}$).
4) Inject the computed dose (d_t).

Each step of the algorithm has a corresponding function in the control application, which is specified as precondition, postcondition, and invariant. However, execution of the application has to be protected against variants of a command manipulation attack [5], which is classified as a computational attack. Launching a typical command manipulation attack requires the following steps (which correspond to high-level intermediate states):

- illegitimate access to the relevant resources;
- modify a command (i.e., instructions that implement the algorithmic steps) to achieve the malicious goal.

For a particular case, we consider the command that computes the new dose of insulin as per step 3 of the algorithm mentioned above. Now, we formalize this attack into declaration and definitive properties in the following sections.

1) *Declarative properties:* The declarative 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, runtime 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 runtime environment (i.e., a set of pairs of identifier and its types).

On the basis of the above descriptions, we formalize the example command manipulation attack as follows:

$$\frac{d_t = computeDose(r_t, l_t, D) \quad \alpha == \text{"}A\text{"} \Leftrightarrow \neg safe(d_t)}{\langle computeDose(r_t, l_t, D), s, re\rangle \curvearrowright \langle \alpha, s', re'\rangle}$$

where

$$safe(d_t) \Leftrightarrow d_t \leq max\_single\_dose \wedge today\_doses + d_t \leq max\_day\_dose \wedge l_t >= min\_safe\_lvl.$$

The rule states that execution of an application component ($computeDose(r_t, l_t, D)$) 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 insulin dose is greater than the maximum allowed single dose;
2) or sum of new computed insulin dose and cumulative dose for today is greater than the maximum allowed dose in a single day; or
3) when measured glucose level is critically low; this property can be both safety and security threat, however, we handle it as security threat.

The formalization assures that there is an attack without considering intermediate steps of the attack, namely either the output value d_t has updated or implementation of step 3 has been modified. This allows to verify within real-time constraints, if the application is under attack, if attack is detected, the application goes in a safe-operation mode and tries to investigate the exact attack The attack presented is simple and occurs only in one state, which implies that its initial and final states are the same.

2) *Definitive properties:* The goal of formalization of definitive properties is to identify 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, runtime environment ρ may yield a state σ', and environment ρ' with an attack α in many small-steps.

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

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

where

$$diagnose(cmd, cnd) : \text{INSTRUCTION} \cup \text{START} \cup \text{END}$$
$$inv(d_t) \Leftrightarrow \neg safe(d_t)$$

returns the point in a program (cmd) where the given condition (cnd) is violated. This point 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 $computeDose(r_t, l_t, D)$ in state s (and environment re) may yield state s' (and environment re') with an exact attack α s.t.

```
public enum Level {INJECT, NOTHING}
public enum Rate { DECREASING, NORMAL, INCREASING}
public enum Level {FALLING, NORMAL, RISING}

class InsulinPumpControllerSpec {
  private int gluc_lvl = -1;
  private ArrayList<int> insu_doses = new ArrayList<int>();
  private ArrayList<int> gluc_lvls = new ArrayList<int>();
  private constant int N = 2;
  ...

  public void readGlucose(int reading){
    if(abs(reading) > SENSOR_ACCURACY)
     if(gluc_lvl > 0)
     gluc_lvl = { (reading | True)}; }

  // computing effect of glucose
  public Level computeLevel(){
    Level lvl =
    { v | (v = FALLING ->  glucose_lvl < gluc_lvls[N])
     /\ (v = NORMAL ->  gluc_lvl == gluc_lvls[N])
     /\ (v = RISING ->  gluc_lvl > gluc_lvls[N]) };
     return lvl;
  }

  // computing rate of glucose change
  public Rate computeRate(){
    Rate rate =
    { r | (r = DECREASING ->
     (gluc_lvl-gluc_lvls[N])<(gluc_lvls[N]-gluc_lvls[N-1]))
     /\ (v = NORMAL ->
     (gluc_lvl-gluc_lvls[N])==(gluc_lvls[N]-gluc_lvls[N-1])
     )
     /\ (v = INCREASING ->
     (gluc_lvl-gluc_lvls[N])>(gluc_lvls[N]-gluc_lvls[N-1]))
      };
     return r;
  }

  // computing next dose of insulin
  public int computeDose(){

    // attack A1 specification
    gluc_lvl >= min_safe_lvl

    // computing insulin dose
    int insu_dose = 0;
    Level lvl = computeLevel();
    Rate r = computeRate();

    if(gluc_lvl <= max_safe_lvl){
     if(rate == DECREASING) insu_dose = 0;
     else if(rate == INCREASING)
      insu_dose = min_single_dose+insu_doses[N]/3;
    }else{
     if(lvl == RISING)
        insu_dose = min_single_dose+insu_doses[N]/3;
     else if(lvl == NORMAL || lvl == FALLING)
        insu_dose = min_single_dose;
     else insu_dose = 0;
    }

    // attack A3 specification
    insu_dose <= max_single_dose && insu_dose + today_doses
      <= max_day_dose
    return insu_dose; }

     // send command to pump for injecting given dose
    public Action injectDose(int insu_dose){
      // attack A2 specification
      insu_dose <= max_single_dose && insu_dose +
     today_doses <= max_day_dose
      Action act =
      { a | (a = INJECT -> insu_dose > 0)
      /\ (a = NOTHING -> insu_dose = 0)};
    }
}
```

Listing 1. Insulin pump controller + attack specification.

- if α is A_1, then measured glucose level has been compromised or threatened;
- if α is A_2, then result of the function (shared through sensor) has been compromised only;
- if α is A_3, then some instruction has been compromised/modified.

Note that if the invariant is violated at START, it also means that the input data sent to the pump via command signal has been compromised. If the invariant at any intermediate point (INSTRUCTION) of the execution is violated, this means that the value has been incorrectly modified by the execution of the program either by an attacker or due to an implementation error. If the invariant is violated at the END of the program, this also means that the program has computed incorrect value and returning an incorrect doze.

On the basis of the formalization mentioned above, we generate runtime security monitor, which is discussed in the following section.

Security monitor

The goal of the security monitor is to ensure that the healthcare control application is secure against known attacks (e.g., command manipulation) at runtime without hindering its performance requirement so as to ensure good health of the patient. To develop such an efficient security monitor, we have first formalized attacks at two different but practical levels of abstraction. Clearly, monitoring declarative properties consumes less amount of time since monitor only verifies initial state (i.e., preconditions) and final state (i.e., postconditions) of an attack class. While monitoring definitive properties consumes more time because it requires verification of all intermediate states (i.e., invariant) in addition to initial (i.e., preconditions) and final state (i.e., postconditions).

On the basis of deductive synthesis [11], [12], we derive security monitor through interactive step-wise refinement of the attack models (i.e., declarative properties) with respect to application specification. We start with an initial nondeterministic attack model with concealed performance requirements, that is, efficiency. Then, the model is synthesized through step-wise refinements, where each refinement optimizes some model statements at least such that no extra behavior is introduced that is beyond that of the optimized statements, on the one hand, and none of the security properties are violated, on the other

hand. As a final step, the model is refined into a fully deterministic implementation of a monitor that is not only correct with respect to attack class specification but is also obviously secure and efficient, respecting real-time performance constraints by employing efficient representations and algorithms as demonstrated in [11].

Example

To demonstrate our approach, we generate a security monitor for our running example of insulin control pump. For simplicity, we generate monitor in a familiar notation, that is, Javalike syntax as shown in Listing 3.

For our example, insulin pump controller specification (Listing 1), in each specific period of time (say 10 minutes), reads glucose level of the patient (see l.12) and then either the pump INJECTs some amount of insulin to the patient (see l.70), or does NOTHING (see l.71). Initially, the glucose level is undefined (see l.6). At any specific interval, the new glucose level is read, if the glucose 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 insulin pump controller either

- INJECTs insulin, that is, the controller first computes the appropriate amount of insulin such that glucose level becomes safe (see l.45–58) and then injects the desired amount of insulin to the patient (see l.70);
- or does NOTHING, that is, the controller concludes that the glucose level of the patient does not require any more insulin (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:

- *Data integrity attacks* (case A_1)—in which the command parameter is compromised or is critically low, that is, the insulin (sensor) dose gets compromised (see l.68) and (case A_2)—in which the sensor-based measured glucose level is compromised (see l.42).
- *Computational attack* (case A_3)—in which the command computational code is compromised, that is, the instruction(s) in the implementation is modified (see l.62) such that they now compute insulin values, which are either undesired for the patient or life threatening to her.

Based on the design decisions and synthesizing the specification (Listing 1), we 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

```
1  ...
2  class InsulinPumpControllerCode {
3    private int gluc_lvl = -1;
4    private ArrayList<int> insu_doses = new ArrayList<int>();
5    private ArrayList<int> gluc_lvls = new ArrayList<int>();
6    private constant int N = 2;
7    ...
8
9    public void readGlucose(int reading){ gluc_lvl = reading;
         }
10
11   // computing effect of glucose
12   public Level computeLevel(){
13    if(gluc_lvl < gluc_lvls[N])  return Level.FALLING;
14    else if(gluc_lvl == gluc_lvls[N]) return Level.NORMAL;
15    else if(gluc_lvl > gluc_lvls[N]) return Level.RISING;
16
17         return Level.NORMAL;
18   }
19
20   // computing rate of glucose change
21   public Rate computeRate(){
22   if((gluc_lvl-gluc_lvls[N])<(gluc_lvls[N]-gluc_lvls[N-1]))
23         return Rate.DECREASING;
24   else if((gluc_lvl - gluc_levels[2]) >= (gluc_levels[2] -
        gluc_levels[1]) )
25         return Rate.INCREASING;
26   else  return Rate.NORMAL;
27   }
28
29   // computing next dose of insulin
30   public int computeDose(){
31
32     // monitor attack A1, if sensor input is compromised
33     monitorComputeDosePre(gluc_lvl);
34
35     // computing insulin dose
36     int insu_dose = 0;
37     Level lvl = computeLevel();
38     Rate rate = computeRate();
39
40     if(gluc_lvl <= max_safe_lvl){
41      if(rate == DECREASING) insu_dose = 0;
42      else if(rate == INCREASING)
43         insu_dose = min_single_dose+insu_doses[N]/3;
44     }else{
45      if(lvl == RISING)
46         insu_dose = min_single_dose+insu_doses[N]/3;
47      else if(lvl == NORMAL || lvl == FALLING)
48         insu_dose = min_single_dose;
49      else insu_dose = 0;
50     }
51
52     // monitor attack A3, if computation is modified
53     monitorComputeDosePost(insu_dose);
54
55     return insu_dose; }
56
57     // send command to pump for injecting given dose
58     public Action injectDose(int insu_dose){
59       // monitor attack A2, if sensor input is compromised
60       monitorInjectDosePre(insu_dose);
61       if(insu_dose != 0) return INJECT;
62       else return NOTHING;
63     }
64
65 }
```

Listing 2. Insulin pump controller code.

```
1  class Monitor {
2
3     private int min_safe_lvl = 3;
4     ...
5
6     // monitoring attack scenario A1
7     // if measured glucose level is malicious
8     public bool monitorComputeDosePre(int gluc_lvl){
9          if(gluc_lvl < min_safe_lvl){
10          suspendExecution(); return false; }
11          return true;    }
12
13     // monitoring attack scenario A2,
14     // if command (injection amount) has been modified
15     public bool monitorInjectDosePre(int insu_dose){
16      if (insu_dose > max_single_dose ||
17       insu_dose + today_doses > max_day_dose){
18        suspendExecution(); return false; }
19       return true;
20    }
21
22    // monitoring attack scenario A3,
23    // if computations have been modified
24    public bool monitorComputeDosePost(int insu_dose){
25    if (insu_dose > max_single_dose ||
26     insu_dose + today_doses > max_day_dose){
27      suspendExecution(); return false; }
28      return true;
29    }
30 }
```

Listing 3. Security monitor code.

(see l.7 and l.23) and a computational attack—A_2 (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.

Simulation and results

We have implemented a prototype simulation in Java 9 on a MacBook Pro with a 2.6-GHz Intel Core i7 processor. In an actual deployment, the controller algorithm would typically run in an automatic pump while the monitor would run in a separate monitoring machine. This would incur more overhead since runtime observation (i.e., parameters values) would need to be exchanged from the controller to the monitor.

For demonstration, we assume that the controller conducts a control cycle on the insulin pump every 0.1 s (which is much faster than necessary for an actual insulin injection system in particular and for any such physical system in general). We have simulated system based on three different input files demonstrating three attack scenarios. For simplicity, we have simulated declarative and definitive properties of each attack scenario separately. The monitor was successful in detecting declarative properties (based attacks) approx. 1.16×10^{-5} faster than detecting their corresponding definitive properties, which is performance efficient indeed being compared to 0.1 per control cycle, on the one hand, and also demonstrates efficient detection of declarative properties as compared to declarative properties, on the other hand. In Figure 3, we show the overhead for CPU time and real-time of simulation for each of the attack scenarios described in declarative and definitive properties.

Though current results show the proficiency of the approach yet an approach needs to be tested against various practical challenges. For instance, to achieve higher scalability a careful investigation of relationship among declarative and definitive properties of those attacks is required that have distinctive effect in definitive properties. Moreover, the approach needs to support modeling of various emerging threats, for example, privacy and human-in-the-loop-based attacks. Furthermore, current approach assumes that the apparatus for the insulin pump (e.g., clock) works as desired. Current implementation also assumes that the monitor runs securely and remotely with negligible communication overhead because industrial control systems typically have dedicated internal networks.

THE EFFICIENCY OF runtime verification of an application execution w.r.t. the application model (also known as application's behavioral specification and associated known threats) depends on the complexity of the models. To make runtime verification more efficient, we have presented a method that reduces the complexity of threat models by describing them at pragmatic but different levels of abstraction that can be verified in significantly smaller amount of time. Furthermore, we have argued that our proposed attack models can be synthesized into runtime security monitor that can assure the security of

Per 1 x 10^{-1} Cycle	CPU Time			Real-time		
	Attack 1	Attack 2	Attack 3	Attack 1	Attack 2	Attack 3
Declarative properties	7.48 x 10^{-5}	7.22 x 10^{-5}	7.77 x 10^{-5}	7.55 x 10^{-5}	7.30 x 10^{-5}	7.80 x 10^{-5}
Definitive properties	8.71 x 10^{-5}	8.38 x 10^{-5}	8.87 x 10^{-5}	8.98 x 10^{-5}	8.53 x 10^{-5}	8.90 x 10^{-5}

Figure 3. Performance evaluation of the monitor.

application execution against known threats (i.e., data integrity or false data injection and insider attacks) strictly respecting application's efficiency requirements. We have demonstrated that declarative properties can be efficiently verified at runtime as compared to definitive properties. 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. ■

References

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

[2] D. Lo and G. E. Suh, "Worst-case execution time analysis for parallel runtime monitoring," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, Jun. 2012, pp. 421–429.

[3] I. Cassar et al., "A survey of runtime monitoring instrumentation techniques," 2017, *arXiv:1708.07229*. [Online]. Available: https://arxiv.org/abs/1708.07229

[4] D. Jin et al., "JavaMOP: Efficient parametric runtime monitoring framework," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*. Piscataway, NJ, USA: IEEE Press, Jun. 2012, pp. 1427–1430.

[5] C. Li, A. Raghunathan, and N. K. Jha, "Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system," in *Proc. IEEE 13th Int. Conf. e-Health Netw., Appl. Services*, Jun. 2011, pp. 150–156.

[6] A. Chlipala, "Ur/Web: A simple model for programming the web," *Commun. ACM*, vol. 59, no. 8, pp. 93–100, Jul. 2016.

[7] D. Kästner et al., "CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler," in *Proc. Embedded Real Time Softw. Syst. (ERTS2)*. Toulouse, France: 3AF, SEE, SIE, Jan. 2018. [Online]. Available: https://hal.inria.fr/hal-01643290

[8] A. Reid et al., "End-to-end verification of arm processors with ISA-formal," in *Computer-Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham, Switzerland: Springer, 2016, pp. 42–58.

[9] N. Paul, T. Kohno, and D. C. Klonoff, "A review of the security of insulin pump infusion systems," *J. Diabetes Sci. Technol.*, vol. 5, no. 6, pp. 1557–1562, Nov. 2011.

[10] D. C. Klonoff, "Cybersecurity for connected diabetes devices," *J. Diabetes Sci. Technol.*, vol. 9, no. 5, pp. 1143–1147, Sep. 2015.

[11] M. T. Khan, D. Serpanos, and H. Shrobe, "ARMET: Behavior-based secure and resilient industrial control systems," *Proc. IEEE*, vol. 106, no. 1, pp. 129–143, Jan. 2018.

[12] B. Delaware et al., "Fiat: Deductive synthesis of abstract data types in a proof assistant," in *Proc. 42nd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, Mumbai, India, Jan. 2015, pp. 689–700.

Muhammad Taimoor Khan is a Senior Lecturer of cyber security at the University of Greenwich, London, U.K., where he is a member of the Internet of Things and Security (ISEC) Research Group. His current research interest includes formal methods-based security of software systems. Khan has a PhD from Johannes Kepler University, Linz, Austria. He is a Member of IEEE.

Martin Pinzger is a Full Professor at the University of Klagenfurt, Klagenfurt, Austria, where he is heading the Software Engineering Research Group. His research interests include software evolution, mining software repositories, program analysis, software visualization, and automating software engineering tasks. He is a Member of ACM and a Senior Member of IEEE.

Dimitrios Serpanos is the Director of the Industrial Systems Institute (ISI), ATHENA, Patras, Greece, and a Professor of Electrical and Computer Engineering (ECE) with the University of Patras, Patras, Greece. His research interests include embedded Cyber Physical Systems (CPS) and computer architecture. Serpanos has a PhD in computer science from Princeton University, Princeton, NJ. He is a Senior Member of IEEE and a Member of ACM, American Association for the Advancement of Science (AAAS), and New York Academy of Sciences (NYAS).

Howard Shrobe is a Principal Research Scientist at the Massachusetts Institute of Technology (MIT) Computer Science and Artificial Intelligence Laboratory (MIT CSAIL), Cambridge, MA. He is a former Associate Director of CSAIL and a former Director of CSAIL's CyberSecurity@ CSAIL initiative. His research interests include AI, cyber security (particularly of control systems), and new computer architectures for inherently secure computing. Shrobe has a PhD from MIT.

■ Direct questions and comments about this article to Muhammad Taimoor Khan, University of Greenwich, London SE10 9JR, U.K.; m.khan@gre.ac.uk.