Combining Secure System Design with Risk Assessment for IoT Healthcare Systems

Florian Kammüller
Middlesex University London, UK
f.kammueller@mdx.ac.uk

***Abstract*—In this paper, we show how to derive formal specifications of secure IoT systems by a process that uses the risk assessment strategy of attack trees on infrastructure models. The models of the infrastructure are logical models in the Isabelle Infrastructure framework. It comprises actors, policies and a state transition of the dynamic evolution of the system. This logical framework also provides attack trees. The process we propose in this paper incrementally uses those two features to refine a system specification until expected security and privacy properties can be proved. Infrastructures allow modeling logical as well as physical elements which makes them well suited for IoT applications. We illustrate the stepwise application of the proposed process in the Isabelle Insider framework on the case study of an IoT healthcare system.**

I. Introduction

Secure systems are a moving target in the literal sense since they are targeted by attackers but also for system engineers: they need development methods that allow for dynamic change to make up for continuously arising new vulnerabilities of systems previously believed (and maybe even proved) to be secure. System models need to be concise which is achieved by omission of details; refinement into concrete systems adds details not present in the abstract model. Systems may be proved to be secure on the abstract specification and yet attacks may arise that exploit details added by those refinements. In short, attacks unforeseen by security proved system specifications come from outside the model. A real challenge worthwhile to be master-minded is to build a dynamic development process that pre-meditates unforeseen vulnerabilities. Such a process must integrate good engineering practice of co-designing the system together with the attacker's possibilities: a process that interleaves secure system design methods with security risk assessment methods. Established industry-strength methods for secure system design as well as security risk assessment exist: for example formal system specification, quantitative model checking and attack tree analysis. Distributed systems based on the Internet of Things (IoT) seem to allow building more flexible human centered systems. However, a malicious attacker can easily exploit IoT devices to build botnets, lock them with ransomware, or use them as a bridgehead into less accessible networks.

Dynamic risk assessment using attack formalism, like attack graphs, has recently found great attention, e.g. [3]. However, usually, the focus of the process lies on attack generation and response planning while we address the design of secure systems. Rather than incident response, we intend to use early analysis of system specification to provide a development of secure systems. This includes physical infrastructure, like IoT system architecture, as well as organisational policies with actors. The means to enable us to integrate security at such an early phase at design time is that we use powerful logical specification of high-level system designs and machine assisted proof of security properties to enhance the process.

The contributions of this paper are: (a) we present an iterative process of system specifications with attack tree analysis that incrementally refines a system specification; (b) we illustrate and thus validate the process on an IoT healthcare example from our CHIST-ERA project SUCCESS [2].

We first summarise the Isabelle Infrastructure framework and the IoT healthcare case study (Section II). Next, we present the Refinement-Risk-Loop (RR-Loop) and an overview of its application to the case study (Section III). Then we provide technical details of the loop's application by the stepwise system refinement steps triggered by attacks (Section IV and V) before we conclude in Section VI.

II. Background

A. Isabelle Infrastructure Framework

Isabelle is a generic Higher Order Logic (HOL) proof assistant. Its generic aspect allows the embedding of so-called object-logics as new theories on top of HOL. There are sophisticated proof tactics available to support reasoning: simplification, first-order resolution, and special macros to support arithmetic amongst others. Object-logics are added to Isabelle using constant and type definitions forming a so-called *conservative extension*. That is, no inconsistency can be introduced: new types are defined as subsets of existing types; properties are proved using a one-to-one relationship to the new type from properties of the existing type. The use of HOL has the advantage that it enables expressing even the most complex application scenarios, conditions, and logical requirements. Isabelle enables the analysis of meta-theory, that is, we can prove theorems *in* an object logic but also *about* it.

This allows the building of telescope-like structures in which a meta-theory at a lower level embeds a more concrete "application" at a higher level. Properties are proved at each level. Interactive proof is used to prove these properties but the meta-theory can be applied to immediately produce results. Figure 1 gives an overview of the Isabelle Infrastructure framework with its layers of object-logics – each level below embeds the one above.

978-1-5386-9151-9/19/$31.00 ©2019 IEEE

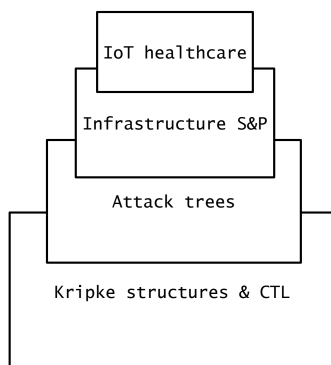

Fig. 1. Generic framework for infrastructures embeds applications.

The Isabelle Infrastructure framework has been created initially for the modeling and analysis of Insider threats [12]. Its use has been validated on the most well-known insider threat patterns identified by the CERT-Guide to Insider threats [1]. More recently, this Isabelle framework has been successfully applied to realistic case studies of insider attacks in airplane safety [7] and on auction protocols [8]. These larger case studies as well as complementary work on the analysis of Insider attacks on IoT infrastructures, e.g. [9], have motivated the extension of the original framework by Kripke structures and temporal logic as well as a formalisation of attack trees [4]. Recently, GDPR compliance verification has been demonstrated [5].

In the course of this extension, the Isabelle framework has been restructured such that it is now a general framework for the state-based security analysis of infrastructures with policies and actors. Temporal logic and Kripke structures build the foundation. Meta-theoretical results have been established to show equivalence between attack trees and CTL statements. This foundation provides a generic notion of state transition on which attack trees and temporal logic can be used to express properties.

We apply Kripke structures and CTL to model state based systems and analyse properties under dynamic state changes. Snapshots of systems are the states on which we define a state transition. The temporal logic CTL is then employed to express security and privacy properties. The meta-theory can be used to navigate between CTL and attack trees establishing attacks. In this paper, we present an integrated process showing how refinements of the system specification can be interleaved with attack analysis to iterate until security properties can be proved in Isabelle.

B. IoT Healthcare System

The example of an IoT healthcare systems is from the CHIST-ERA project SUCCESS [2] on monitoring Alzheimer's patients. Figure 2 illustrates the system architecture where data collected by sensors in the home or via a smart phone helps monitoring bio markers of the patient. The data collection is in a cloud based server to enable hospitals (or scientific institutions) to access the data which is controlled via the smart phone.

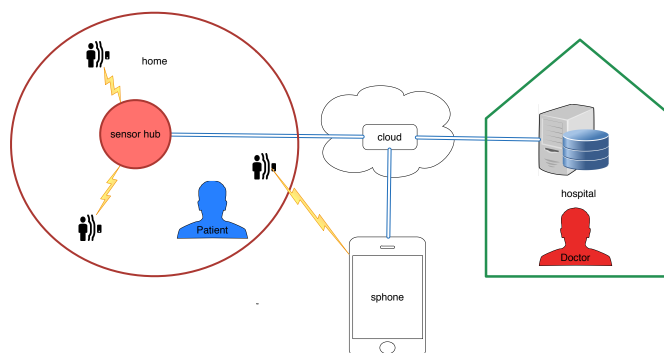

Fig. 2. IoT healthcare monitoring system for SUCCESS project

III. The Refinement-Risk-Loop for Secure IoT System

We first introduce the iterative process of refinement and attack tree analysis (the "Refinement-Risk-Loop") and give a tabular overview of the steps taken for the case study. How the models are expressed and refined in each iteration as well as the attack trees that exhibit vulnerabilities, is discussed afterwards.

As an initial step, we propose the Fusion/UML method for developing a system architecture from early requirements. This system architecture is translated into the Isabelle Infrastructure framework: actors in UML become Isabelle Infrastructure actors, UML system classes are represented by locations in the infrastructure graph, and the class attributes and pre- and postconditions of methods are formalised in the local and global policies. The identification of attacks, using for example invalidation [11], can then reveal paths of state transitions through the system model where the global security policy is violated. In an iteration, these attack paths provide details useful for refining the system specification by adding security controls, for example, access control, privacy preservation, or blockchain. The addition of detail, however, may in turn introduce new vulnerabilities that lead to new iterations of the process. Security properties may be proved at each level of the iteration. They are true for this abstraction level of the system model. However, as is known in the "security paradox": attacks mostly come from outside the model. Attacks may be found despite proved security. If these attacks undermine the proven properties, it is because they use information not present in the model. But this yields the key to finding a refinement: introducing a level of detail that enables a formal or computational representation of the details used in the attack incarnates the next refinement. This process is graphically depicted in Figure 3.

Following the RR-Loop, we have modelled and analysed the IoT healthcare application in four iterations summarised in the following table. The technical details of these steps are discussed next.

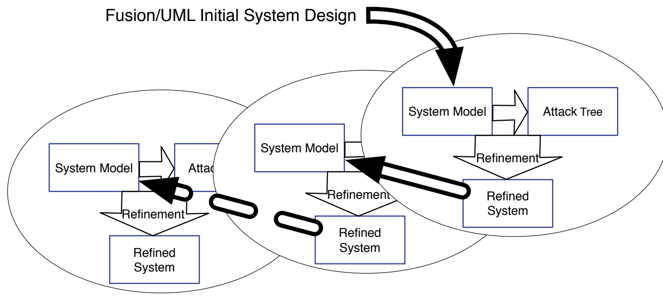

Fig. 3. Refinement-Risk-Loop iterates design, risk analysis, and refinement

System	Attack
Initial Fusion system home-cloud-hospital	Eve can perform action get at cloud
Refinement-Risk-Loop Iteration 1	
Access control by DLM labels	Eve can perform action eval at cloud; changes label to her own
Refinement-Risk-Loop Iteration 2	
Privacy preserving functions type `label_fun`	Eve puts Bob's data labelled as her own
Refinement-Risk-Loop Iteration 3	
Global blockchain	Eve is an insider impersonating the blockchain controller
Refinement-Risk-Loop Iteration 4	
Consensus (for example Nakamoto) blockchain	no attack known yet

IV. Applying RR-Loop to IoT Healthcare Example

A. Initial Step: Fusion/UML for System Architecture

We have used the Fusion/UML process for object oriented design and analysis to derive a system design for the application scenario. For reasons of conciseness, we omit here the details presenting just one of the main outcomes of the analysis process: the system class model as depicted in Figure 4. Note that, within the security perimeter, we place only the cloud server and the connected hospital (or other client institutions). The smartphone and the home server feature as data upload devices and the smartphone additionally as a control device that is included in some of the use cases. This is a consequence of the GDPR [5] and immediately settled in the initial architecture. Another result of the Fusion/UML analysis along with this system architecture is a set of operation schemas based on the system class model, additional use cases and object collaborations. For details see [10].

A major observation in the system architecture depicted in Figure 4 is that the security perimeter stretches over two separate distributed systems.

B. Infrastructures, Policies, and Actors

The Isabelle Infrastructure framework supports the representation of infrastructures as graphs with actors and policies attached to nodes. These infrastructures are the *states* of the Kripke structure.

The transition between states is triggered by non-parametrized actions `get`, `move`, `eval`, and `put` executed by actors. Actors are given by an abstract type `actor` and a function `Actor` that creates elements of that type from identities (of type `string` written ''s'' in Isabelle). Actors are contained in an infrastructure graph constructed by `Lgraph` – here the IoT healthcare case study example.

```
ex_graph ≡ Lgraph
   {(home, cloud), (sphone, cloud), (cloud,hospital)}
   (λ x. if x = home then {''Patient''} else
         (if x = hospital then {''Doctor''} else {}))
   ex_creds ex_locs
```

This graph contains a set of location pairs representing the topology of the infrastructure as a graph of nodes and a function[1] that assigns a set of actor identities to each node (location) in the graph. The last two graph components `ex_creds` and `ex_locs` are here abbreviated only (for the definitions see [6]). The function `ex_creds` associates actors to a pair of string sets by a pair-valued function whose first range component is a set describing the credentials in the possession of an actor and the roles the actor can take on; `ex_locs` defines the data residing at the component. Corresponding projection functions for each of the components of an infrastructure graph are provided; they are named `gra` for the actual set of pairs of locations, `agra` for the actor map, `cgra` for the credentials, and `lgra` for the data at that location.

Infrastructures contain an infrastructure graph and a policy. There are projection functions `graphI` and `delta` when applied to an infrastructure return the graph and the policy, respectively.

Policies specify the expected behaviour of actors of an infrastructure. They are given by pairs of predicates (conditions) and sets of (enabled) actions. They are defined by the `enables` predicate: an actor `h` is enabled to perform an action `a` in infrastructure `I`, at location `l` if there exists a pair `(p,e)` in the local policy of `l` (`delta I l` projects to the local policy) such that the action `a` is a member of the action set `e` and the policy predicate `p` holds for actor `h`.

```
enables I l h a ≡ ∃ (p,e) ∈ delta I l. a ∈ e ∧ p h
```

The function `local_policies` gives the policy for each location `x` over an infrastructure graph `G` as a pair: the first element of this pair is a function specifying the actors `y` that are entitled to perform the actions specified in the set which is the second element of that pair.

```
local_policies G x ≡
case x of
  home ⇒ {(λ y. True, {put,get,move,eval})}
| sphone ⇒ {((λ y. has G (y, ''PIN'')),
                 {put,get,move,eval})}
| cloud ⇒ {(λ y. True, {put,get,move,eval})}
| hospital ⇒ {((λ y. (∃ n. (n  @_G hospital) ∧
                 Actor n = y ∧ has G (y, ''skey''))),
                 {put,get,move,eval})}
```

[1] We use the common λ-abstraction, e.g. `λx. True`, to define functions with parameters, here the function returning True for any input x.

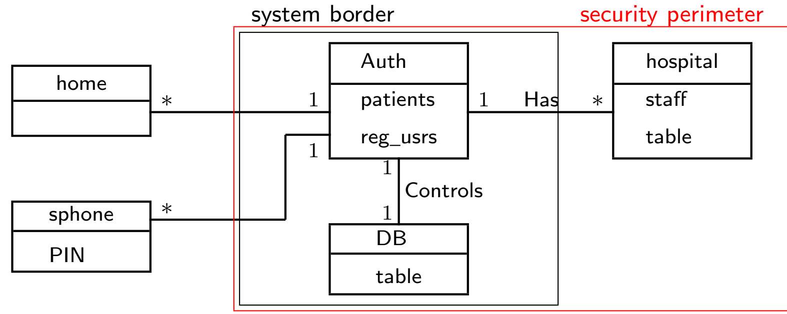

Fig. 4. System class model for IoT healthcare system

```
| _ ⇒ {})
```

The global policy is 'only the patient and the doctor can access the data in the cloud':

```
global_policy I a ≡  a ∉ hc_actors ⟶
                     ¬(enables I cloud (Actor a) get)
```

C. Infrastructure State Transition

The state transition relation uses the syntactic infix notation `I → I'` to denote that infrastructures `I` and `I'` are in this relation. To give an impression of this definition, we show here just one of several rules that defines the state transition for the action get because this rule will be adapted in the process of refining the system specification. Initially, this rule expresses that an actor that resides at a location `l` and is enabled by the local policy in this location to "get" can change the state of that location to the string value `s` representing data stored in location `l'`.

```
get_data: G = graphI I ⟹ h @G l ⟹
enables I l' (Actor h) get ⟹ s ∈ lgra G l' ⟹
I' = Infrastructure
      (Lgraph (gra G)(agra G)(cgra G)
       lgra G (l := lgra G l ∪ {s}))
      (delta I)
⟹ I →n I'
```

D. Attack: Eve can get data

How do we find attacks? The key is to use invalidation [11] of the security property we want to achieve, here the global policy. Since we consider a predicate transformer semantics, we use sets of states to represent properties. The invalidated global policy is given by the following set `shc`.

```
shc ≡ {x. ¬ (global_policy x ''Eve'')}
```

The attack tree calculus [4] exhibits that an attack is possible.

```
hc_Kripke ⊢ EF shc
```

V. Entering the Loop

A. First Refinement Iteration: Adding DLM Access Control

The Decentralised Label Model (DLM) [14] allows labelling data with owners and readers. We adopt it for our model. Labelled data is given by the type `data × dlm` where `data` can be any data type. We provide functions `owns` and `readers` that enable specifying when an actor may access a data item.

```
has_access G l a d ≡ owns G l a d ∨ a ∈ readers d
```

In the first refinement of the model in the RR-Loop, we thus use labeled data to adapt the infrastructures. Also the state transition now implements access control. The refined rule `get_data'` checks the labels for the data item stored in a location `l'` and only gives access if – in addition to get being enabled for an actor `h` – also this actor is among the readers or is the owner. In this case the data item including the label can be copied to the location `l` where `h` resides.

```
get_data':
G = graphI I ⟹ h @G l ⟹
enables I l' (Actor h) get ⟹
(n, (Actor h', hs)) ∈ (lgra G l') ⟹
Actor h ∈ hs ∨ h = h' ⟹
I' = Infrastructure
     (Lgraph (gra G)(agra G)(cgra G)
      lgra G (l := lgra G l ∪ {(n, (Actor h', hs))}))
      (delta I)
⟹ I →n I'
```

B. Attack: Eve can change labels

The above "get" attack is still valid in the refined model but this does not matter any more since the global policy changes. We cannot quite express the new policy yet before refining the model but we can already observe another attack. Eve can also process data using the eval action at the cloud: we can prove there is a path (EF) in the system leading to the corresponding attack state.

```
hc_Kripke ⊢
        EF {I. enables I cloud (Actor ''Eve'') eval}
```

Using the Completeness theorems for the attack tree calculus [4] we can thus derive that an attack exists: Eve can tamper with the access control labels by processing labeled data. We need to prove privacy preservation, i.e. that labels are preserved. As a countermeasure to this attack, the next iteration of the refinement loop thus enforces label preserving functions.

C. Second Iteration: Privacy Preservation

The labels of data must not be changed by processing. This invariant can be formalized in our Isabelle model by a type definition of functions on labeled data that preserve their labels.

```
typedef label_fun = {f :: data × dlm ⟹ data × dlm.
                       ∀ x. snd x = snd (f x)}
```

We also define an additional function application operator ⇕ on this new type. Then we can use this restricted function type to implicitly specify that only functions preserving labels

may be applied in the definition of the system behaviour in the state transition rules.

Furthermore, we can prove now that only entitled users (owners and readers) can access data: privacy is preserved by the use of label preserving functions. We can prove that processing preserves ownership for all paths globally (expressed using the CTL quantifier AG), That is, in all states of the Kripke structure and all locations of the infrastructure graph we have that the ownership in the initial state `hc_scenario` will persist.

```
theorem priv_pres: h ∈ hc_actors ⟹
  l ∈ hc_locations ⟹
  owns (Igraph hc_scenario) l (Actor h) d ⟹
  hc_Kripke ⊢ AG {x. ∀ l ∈ hc_locations.
                  owns (Igraph x) l (Actor h) d }
```

D. Attack: Eve can simply put data

When trying to prove a theorem to express that different occurrences of the same data in the system must have the same labels, we fail. The reason for this is the following attack.

```
hc_Kripke ⊢
        EF {I. enables I cloud (Actor ''Eve'') put}
```

Eve could learn the data by other means than using the privacy preserving functions and using the action put to enter that data as new data to the system labelled as her own data. As a countermeasure, we need a concept to guarantee consistency across the system: blockchain.

E. Third RR-Loop Iteration: Blockchain Consistency

One major achievement of a blockchain is that it acts like a distributed ledger, that is, a global accounting book. A distributed ledger is a unique consistent transcript keeping track of protected data across a distributed system. In our application, the ledger must mainly keep track of where the data resides for any labelled data item. We formalize a ledger thus as a type of functions that maps a labelled data item to a set of locations. In this type, we further constrain each data to have at most one valid data label of type `dlm`. This is achieved by stating that there exists a unique (∃!) label l for which the location set `ld(d, l)` assigned to by the ledger is *not* empty – unless it is empty for all labels for `d`.

```
typedef ledger = { ld :: data × dlm ⇒ location set.
                   ∀ d. (∀ l. ld (d, l) = {}) ∨
                        (∃! l. ld (d, l) ≠ {}) }
```

The addition of `set` makes the range of the ledger a set of sets of locations which allows for none (empty set) or a number of locations to be assigned to a data item.

F. Ledger enables Data Protecting State Transition

The set of rules for defining the state transition of infrastructures needs to be adapted to the refined model. The refinement by a ledger is incarnated into the system specification to guarantee consistency across distributed units.

1) The get data rule: now requires that the ledger be updated by noting that the data item also resides in the new location `l`. This is achieved by unifying the existing set of locations `L` for this data item with the new location `l`. The existing set of locations `L` is simply retrieved by applying the ledger `ledgra G` to the data item `n` and its label `(Actor h', hs)`. The update of the ledger at the position `ledgra G (n, (Actor h', hs))` of this data item uses the operator `:=` to change the ledger to contain the new list of locations `L ∪ {l}`.

```
get_data'': G = graphI I ⟹ h @G l ⟹
enables I l' (Actor h) get ⟹ Actor h ∈ hs ∨ h = h'
⟹ ledgra G (n, (Actor h', hs)) = L ⟹ l' ∈ L ⟹
I' = Infrastructure
      (Lgraph (gra G)(agra G)(cgra G)(lgra G)
           (ledgra G (n, (Actor h', hs)) := L ∪ {l})
      (delta I)
⟹ I →n I'
```

2) The put data rule: assumes an actor `h` residing at a location `l` in the infrastructure graph `G` and being enabled the `put` action. If infrastructure state `I` fulfils those preconditions, the next state `I'` can be constructed from the current state by adding the data item `n` with label `(Actor h, hs)` at location `l`. The addition is given by updating (using `:=`) the existing ledger `legdra G`. The ledger is set for this labelled data item `(n, (Actor h, hs))` initially as the singleton set `{l}` containing just this location. Note that the first component `Actor h` marks the owner of this data item as `h`. The other components are the reader list `hs`, and the actual data `n`.

```
put: G = graphI I ⟹ h @G l ⟹
enables I l (Actor h) put ⟹
I' = Infrastructure
      (Lgraph (gra G)(agra G)(cgra G)(lgra G)
              (ledgra G (n, (Actor h, hs)) := {l}))
      (delta I)
⟹ I →n I'
```

In the extended Infrastructure of the refined system the infrastructure graph needs to be extended by the ledger.

```
ex_graph ≡ Lgraph
   {(home, cloud), (sphone, cloud), (cloud,hospital)}
   (λ x. if x = home then {''Patient''} else
         (if x = hospital then {''Doctor''} else {}))
   ex_creds ex_locs ex_ledger
```

The data and its privacy access control definition is given by the parameter `ex_ledger` specifying that the data 42, for example some bio marker's value, is owned by the patient and can be read by the doctor and is currently only contained in location `home`.

```
ex_ledger ≡ (λ (d, l).
if d = 42 ∧ l = (Actor''Patient'',{Actor''Doctor''})
then {home} else {})
```

G. Ledger Guarantees Consistent Data Ownership

We can now prove that data protection is consistent across the infrastructure. If in any two locations the same data item `n` resides, then the labeling must be the same. That is, the owner and set of readers are identical.

```
theorem Ledger_con: h, h' ∈ hc_actors ⟹
  l, l' ∈ hc_locations ⟹
  l ∈ ledgra G (n, (Actor h, hs)) ⟹
  l' ∈ ledgra G (n, (Actor h', hs')) ⟹
  (Actor h, hs) = (Actor h', hs')
```

This property immediately follows from the invariant property of the type definition of the type ledger (see Section V-E) and privacy preservation given by the label function type (see Section V-A). This means that the corresponding interactive proofs that we have to provide to Isabelle are straightforward and largely supported by its automated tactics (see the Isabelle source code [6] for details).

H. Attack and Fourth RR-Loop: Eve can overwrite blockchain

Despite the above proved theorem, there is yet another aspect – as usual outside the model – that leads to an attack. In the abstract specification of a ledger, we have omitted the implementation of a blockchain. We could have a centrally controlled blockchain in which one part signs the entire blockchain to guarantee consistency. Eve could be an insider impersonating the blockchain controller. In that case, she could just overwrite the entry made by Bob and add his data as her own. Formally, we can re-use the put attack of the previous level using the rule put above to overwrite Bob's entry by Eve's.

As a refinement for the RR-Loop, we need to consider a consensus algorithm, like Nakamoto's used in Bitcoin, between the participants in the distributed system to chose a different leader for each blockchain commitment to avoid the attack. Adding a refinement with a Nakamoto consensus to our model is possible but rather complex. However, we can simply specify the effect of this refinement in the system specification by adding `∀a as. ledgra G (n, (Actor a, as)) = {}` as a precondition to the rule put, that is, the data item must not yet be assigned to anyone in the ledger in order to allow a put action.

VI. Conclusion

In this paper, we have presented the Refinement-Risk-Loop as a method that interleaves formal system specification in the Isabelle Infrastructure framework with attack tree analysis thereby refining the security of a system. The method is particularly useful for IoT systems since it allows modeling physical as well as logical realities. We have illustrated the RR-Loop process on an IoT healthcare example running four iterations adding access control, privacy preservation, and a ledger for global consistency.

Compared to other verification techniques, like Modelchecking, Isabelle requires user interaction. However, Modelchecking is restricted to finite models and first order logic. Isabelle enables the use of higher order quantification and induction necessary for invariant proofs (like Theorem `priv_pres` in Section V-C).

The use of a distributed ledger, also known as a blockchain, is new for formal system specification and verification. There are currently many attempts to formalize blockchains but most of them are very close to technical implementations, e.g. [13], thus obliterating the possibility to provide clear specification of legal requirements like we do with respect to the GDPR [5]. Moreover, to our knowledge, none of these formal models has been produced in Isabelle or similar Higher Order Logic tools. Our formalization uses a generic notion of a ledger that may simply control consistency in the distributed application.

Risk assessment loops exist for secure systems, e.g. [3]. There the process generates attacks in order to plan incident responses. By contrast, we use the risk assessment to improve the design of secure systems by refinement. The novelty of our approach is to integrate the dual of risk assessment with attack trees into a constructive development of system specifications. This approach clearly demands a certain level of familiarity with logical specification. However, abstract system specifications can be provably refined and finally code can be extracted to major programming languages, e.g. Scala.

References

[1] D. M. Cappelli, A. P. Moore, and R. F. Trzeciak. *The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud)*. SEI Series in Software Engineering. Addison-Wesley Professional, 1 edition, Feb. 2012.

[2] CHIST-ERA. Success: Secure accessibility for the internet of things, 2016. http://www.chistera.eu/projects/success.

[3] G. Gonzalez-Granadillo, S. Dubus, A. Motzek, J. Garcia-Alfaro, E. Alvarez, M. Merialdo, S. Papillon, and H. Debar. Dynamic risk management response system to handle cyber threats. *Future Generation Computer Systems*, 83:535–552, 2018.

[4] F. Kammüller. Attack trees in isabelle. In *20th International Conference on Information and Communications Security, ICICS2018*, volume 11149 of *LNCS*. Springer, 2018.

[5] F. Kammüller. Formal modeling and analysis of data protection for gdpr compliance of iot healthcare systems. In *IEEE Systems, Man and Cybernetics, SMC2018*. IEEE, 2018.

[6] F. Kammüller. Isabelle infrastructure framework with iot healthcare s&p application, 2018. Available at https://github.com/flokam/IsabelleAT.

[7] F. Kammüller and M. Kerber. Investigating airplane safety and security against insider threats using logical modeling. In *IEEE Security and Privacy Workshops, Workshop on Research in Insider Threats, WRIT'16*. IEEE, 2016.

[8] F. Kammüller, M. Kerber, and C. Probst. Towards formal analysis of insider threats for auctions. In *8th ACM CCS International Workshop on Managing Insider Security Threats, MIST'16*. ACM, 2016.

[9] F. Kammüller, J. R. C. Nurse, and C. W. Probst. Attack tree analysis for insider threats on the IoT using Isabelle. In *Human Aspects of Information Security, Privacy, and Trust - Fourth International Conference, HAS 2015, Held as Part of HCI International 2016, Toronto*, Lecture Notes in Computer Science. Springer, 2016. Invited paper.

[10] F. Kammüller, O. O. Ogunyanwo, and C. W. Probst. Using fusion/uml for iot architecures for healthcare applications. *arXiv*, https://arxiv.org/abs/1901.02426, 2018.

[11] F. Kammüller and C. W. Probst. Combining generated data models with formal invalidation for insider threat analysis. In *IEEE Security and Privacy Workshops (SPW)*. IEEE, 2014.

[12] F. Kammüller and C. W. Probst. Modeling and verification of insider threats using logical analysis. *IEEE Systems Journal, Special issue on Insider Threats to Information Security, Digital Espionage, and Counter Intelligence*, 11(2):534–545, 2017.

[13] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy*, pages 839–858. IEEE, 2016.

[14] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 1999.