A Virtualization Assurance Language for Isolation and Deployment

Sören Bleikertz Thomas Groß
IBM Research - Zurich
{sbl,tgr}@zurich.ibm.com

Abstract—**Cloud computing and virtualized infrastructures are often accompanied by complex configurations and topologies. Dynamic scaling, rapid virtual machine deployment, and open multi-tenant architectures create an environment, in which local misconfiguration can create subtle security risks for the entire infrastructure. This situation calls for automated deployment as well as analysis mechanisms, which in turn require a cloud assurance policy language to express security goals for such environments. Where possible, configuration changes should be statically checked against the policy prior to implementation on the infrastructure.**

We study security requirements of virtualized infrastructures and propose a practical tool-independent policy language for security assurance. Our policy proposal has a formal foundation, and still allows for efficient specification of a variety of security goals, such as isolation. In addition, we offer language provisions to compare a desired state against an actual state, discovered in the configuration, and thus allow for a differential analysis. The language is well-suited for automated deduction, be it by model checking or theorem proving.

I. Introduction

Cloud and large-scale virtualized infrastructures give rise to complex configurations. This complexity is a side-effect of highly dynamic scaling, rapid machine deployment and open multi-tenant systems. The complexity renders clouds challenging to administrate with respect to achieving all security requirements. This holds for cloud providers as well as their subscribers. The side effects of configuration changes to high-level security goals, such as isolation of tenants, are non-trivial at best. This is particularly true when performing local low-level configuration changes.

Indeed, research has established that configuration problems in complex environments are likely to result in security problems. A study of problems in large-scale Internet services by Oppenheimer et al. [18] highlights configuration problems as the major source of security issues. We conjecture that these results also apply to virtualized infrastructures and clouds, because these infrastructures are large-scale, interconnected, heterogeneous systems, as well. In addition, studies by Berger et al. [7] point out a complexity increase introduced by the virtualization of the infrastructure.

Given the impact of configuration problems on large-scale infrastructures, we suggest that global high-level security properties must be verified complementary to local low-level ones. This is because local security properties do not compose gracefully to fulfill goals for the entire topology. Let us exemplify this rationale in the case of isolation for a multi-tenant virtualized infrastructure. Even if an administrator configures all resources well with regard to local policy decisions, such as firewall policies for virtual machines or access control policies for virtual storage, there still may be information flow through the connections of the topology, be it by covert channels between virtual machines on the same hypervisor, inter-zone VLAN traffic, or shared physical storage areas.

The complexity of cloud configuration with respect to assuring high-level security goals is tantalizing. It calls either for infrastructure-wide access control and deployment mechanisms to enforce the security goals automatically or for verification mechanisms to check for breaches of the goals. In any case, we need a specification language for high-level assurance goals. Such a language plays a different role in the three cases mentioned: *First* in the access enforcement case, the security assurance language is an auxiliary input to the policy decision engine that has in turn the function to ensure that the high-level assurance goals are preserved by access requests. *Second* in the automated deployment case, the deployment mechanism establishes deployment patterns that maintain the high-level security goals. Best practices and deployment templates that incorporate some security targets are insufficient to fulfill high-level security goals for the entire topology, because a series of local configuration transitions, which fulfill a local-view security property, may still breach a topology-level security goal in a global view. *Third* in the verification case, the high-level security goals constitute the verification target, against which the actual infrastructure is evaluated.

There already exist specification languages for virtualized environments. These languages aim at provisioning (cf. [10], [15]), or network and reachability properties, e.g., firewall topology or distributed network access control [9]. In the former case, the specification languages are restricted to single resources, notably virtual machines, however do not have provisions for statements over the topology. In the latter case, the languages have provisions to model the topology and properties thereof, however they do not provide language primitives for expressing diverse security statements as needed in virtualized infrastructures.

We derived the following three categories of interesting security statements for virtualized infrastructures from existing research literature such as [7], [18], [20]: operational correctness, failure resilience, and isolation. *First*, operational correctness ensures that services are correctly deployed and that their dependencies are reachable. *Second*, failure resilience ensures that the effects of single component failures cannot cascade and affect many entities. *Third*, isolation ensures that different security zones are properly separated and that traffic between security zones is only routed through trusted guardians.

The goal of this paper is to study such high-level security properties of virtualized infrastructures and propose a policy language to express these as goals. We call the resulting language Virtualization Assurance Language for Isolation and Deployment (*VALID*).

A. Contribution

We contribute the first formal security assurance language for virtualized infrastructure topologies. More precisely, we model such an assurance language in the tool-independent Intermediate Format IF [4], which is well suited for automated reasoning. We lay the language's formal foundations in a set-rewriting approach, commonly used in automated analysis of security protocols, with access to graph analysis functions. In addition, we propose language primitives for a comparison of desired and actual states. As a language aiming at expressing topology-level requirements, it can express management and security requirements as promoted by [9]. Management requirements in the cloud context are, for instance, provisioning and de-provisioning of machines or establishing dependencies. Security requirements are, for instance, sufficient redundancy or isolation of tenants. To test soundness and expressibility of our proposal, we model typical high-level security goals for virtualized infrastructures. We study the areas deployment correctness, failure resilience, and isolation, and propose exemplary definitions for respective security requirements in *VALID*.

B. Outline

We structure this paper in a top-down way. We first propose infrastructure-level assurance goals for virtualized systems in Section II. These goals are a diverse sample of the language scope. In Section III, we specify our requirements on the cloud assurance language on a meta-level. We lay the language's formal foundations in Section IV, that is, we introduce its roots in the Intermediate Format IF [4] and our cloud-specific language primitives and syntax. In Section V, we propose formal specifications of checkable attack states for the assurance goals defined in Section II. Thereby, we exemplify the use of *VALID* in its application domain. We briefly discuss a virtualization assurance tool that would incorporate *VALID* in Section VI. We compare our cloud assurance language proposal to other policy language and virtualization security efforts in Section VII.

II. Virtualized Systems Security Goals

We distilled three categories of virtualized systems security goals based on common problems described in existing research literature: *Operational Correctness*, *Failure Resilience*, and *Isolation*. Furthermore, for each of these categories we identified specific goals that our language should be capable of capture and express efficiently. Figure 1 depicts a simple virtualized system example that we will use to illustrate the different security goals.

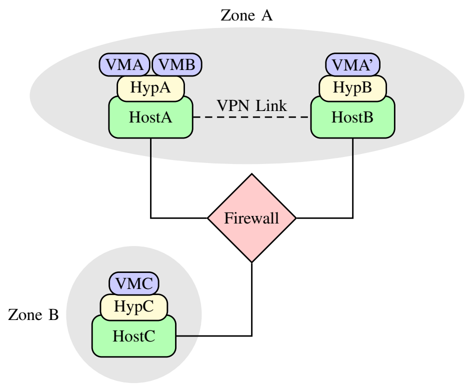

Figure 1. Virtualized System Example

A. Operational Correctness

Operational correctness describes that a service is both correctly deployed and reachable. It bears some similarity to the *Liveness* property introduced in [1], [14] and informally states that "good things" will eventually happen for a service. Configuration mistakes often lead to unavailability of services in traditional data center environments (cf. [18]) and is only intensified in virtualized environments due to their increasing complexity (cf. [7]).

Deployment correctness: means that an entity is deployed in correct operational conditions, which includes multiple factors: i. The geographic location of the host system can have legal and technical consequences, e.g., conflicts with privacy laws, or long end-to-end delay due to geographic disparity. ii. Properties of the host system such as capabilities and reliability can have a significant impact on the service. iii. Furthermore, the configuration of the host system and service has to be correct that the service can actually be run on the host.

Reachability: means that an entity is connected to all its operational dependencies. On one hand, these dependencies can be network reachability, i.e., the VM and physical host are actually reachable over the network from the client-side. On the other hand, these dependencies can be resource dependencies in general, e.g., that a VM is able to access services on other nodes. All such dependencies have to be fulfilled in order that the operational correctness of the service is given.

B. *Failure Resilience*

Failures of components in a computing environment are unavoidable, but a resulting failure of services, which are visible to the end users, can be mitigated. Such containment of component failures are pointed out in [18] and can be summarized as: failure compartmentalization due to *Independent failure*, and prevention of cascading failures and limitation of failure impact due to the *Redundancy*.

Independent failure: means that failures of an entity are well-contained and that dependencies of entities with the same function will fail independent from each other. This goal nurtures a diversity of the components deployed in the computing environment. A typical software stack in a virtualized system consists of a hypervisor, management operating system, virtual machine system, and the service application. A diversity in this stack, such as using different hypervisors from different vendors, will have an isolated failure in case of faults in one of these hypervisor implementations. Independent failure can be satisfied in the example scenario, in case the hypervisors *HypA* and *HypB* hosting the service VMs are provided by different vendors.

Redundancy: means that sufficient replication enforces that individual component failures will leave overall service availability unharmed. The necessary level of redundancy depends on the desired failure resilience for a service, which also depends on its criticality. Sufficient redundancy implies the absence of a single point of failure (SPoF). A SPoF exists in a system, if a dependency of a service is only satisfied by one entity in the whole system. The absence of such a SPoF entity will increase the failure resilience due to the limitation of a cascading failure effect on dependent services. In our example, the service running in *VMA* is replicated in *VMA'*, both running on different physical machine and interconnected with two independent network links.

C. *Isolation*

In virtualized environments, such as public infrastructure clouds, we see multi-tenancy in order to increase the utilization of the system. Isolation compares to *Safety* [1], [14] that undesired information flow do not happen. In [20], the problem of undesired information flow in public infrastructure clouds was exposed.

Isolation of zones: means that specified security zones are isolated from each other, either by correct association of machines to zones or by enforcement of flow isolation between any entity of different zones. A security zone can be any set of entities in the virtualized environment. For example, a zone in the case of tenant isolation is the set of resources used by a tenant, and zone isolation is given if the tenants do not have access to common resources. In the illustrated example, we have defined two security zones *Zone A* and *Zone B* that are disjoint, i.e., isolated of each other.

Guardian mediation: means that information flow between zones is allowed if, and only if, mediated by a trusted guardian. In case information flow is allowed between the two security zones defined in our example case, the *Firewall* guardian has to mediate the traffic between the zones.

Other goals regarding isolation are *Chinese wall policy* and *secure channels*.

III. Requirements

A. *Formal Foundations*

Virtualized environments can gain complexity beyond human oversight and therefore require tool-supported deployment and analysis. Thus, we expect the security assurance language to have formal rigor and be suitable for automated reasoning. This requirement implies a simple, mathematical structure with controllable state space.

B. *Expressibility*

There are many different security requirements imposed on virtualized infrastructures. Therefore, we require that the security assurance language needs to be able to efficiently express a wide range of security properties as discussed in Section II. *First*, the language needs to have *three expression layers*: i. statements about properties of resources, e.g., their IP address or functional classification, ii. set operations, such as membership in security zones, iii. graph operations, such as existence of an information flow or dependency path in a graph model of the topology. *Second*, the language needs to be *reflexive* and *self-contained*, that is, one can define new security goals with the existing terms of the grammar and without the need of auxiliary grammar.

As a corollary of this requirement, we propose that the security assurance language shall express *attack states*, that is, states in which a security property is violated, as well as *ideal states*, that is, states that assure a correct system behavior. Whereas the first approach is suitable for more efficient security analysis (model checking) without complete state exploration, the second approach is suitable for complete verification (theorem proving).

C. *Tool and Standard Independence*

Virtualized environments are still a young field without settled predominant standards. Therefore, we require the specification language to be independent from a specific vendor's tool or a specific standard.

D. *Desired State Comparison*

The validation of security properties of virtualized environments provides two different views on the state of such a virtualized infrastructure: a *desired* state or the *ideal world*, as specified in the policy, and an *actual* state or *real world*, i.e., the current configuration of the virtualized infrastructure. One specific goal of our assurance language is to express comparisons of a desired state and an actual state discovered in a configuration. Sometimes it is necessary to make statements about ideal elements as well as real elements in the very same policy statement. Consider the example that a VM should be hosted on a specific host. Or in other words, the goal is breached if the VM is hosted on a different machine than specified. This breach can be efficiently captured using both elements from the ideal and real world in one policy statement. We specify that we have an ideal machine hosting the VM and also a real machine hosting the same VM. In order to describe the placement breach, we say that these two machines do not correspond to each other, i.e., the real machine is not the same as the ideal one in terms of the given properties. Therefore, if such a statement holds, we observed a placement breach.

IV. Language Syntax

We propose a specification and reasoning language for security properties of virtualized environments based on set-rewriting and conditions over states.

VALID uses a subset of the AVISPA Intermediate Format IF [4] as its basis, a meta-language for automated deduction based on set manipulation and conditions over state expressions. We chose IF as the basis for our work because of its capability to efficiently express goals as stated in Section II, its natural extensibility to state transition formulations, its tool-independence, and its close relation to general-purpose automated deduction, which is given due to the strong formal foundation of IF, and its support by model checkers and theorem provers.

A. Term Algebra and Atomic Terms

We start from *atomic terms*, that is constants and variables. The value of a constant is fixed, e.g., the symbol for the type machine. We call the set of all constant terms *signature*. A variable can be matched against any value (of matching type). Atomic terms with different symbols have different values.

Definition 1 (Term Algebra). *We define a term algebra over a signature Σ and a variable set $\mathcal{V}$. Constants and variables are disjoint alphanumeric identifiers: constants start with a lower-case letter; variables start with an upper-case letter. We typeset IF elements in* $\mathsf{sans-serif}$.

The signature Σ contains a countable number of constant symbols that represent resource names, numbers and strings.

The atomic terms are typed (see Table I):

Definition 2 (Type System). *We have a set of basic types:*

$$\mathbb{T} := \{\mathsf{node}, \mathsf{machine}, \mathsf{host}, \mathsf{hypervisor}, \mathsf{machineOS}, \mathsf{hostOS}, \mathsf{network}, \mathsf{zone}, \mathsf{class}\}$$

We write $t : \tau$ for a term t having type τ. Variables can be untyped or typed. If a variable has a basic type, it can generally only be matched against a constant with matching type. The type symbol node *represents a super-type: variables of type* node *can match against types in the sub-set:*

$$\mathbb{T}_{\mathsf{N}} := \{\mathsf{machine}, \mathsf{host}, \mathsf{hypervisor}, \mathsf{machineOS}, \mathsf{hostOS}, \mathsf{network}\}$$

Table I
Basic type constants for virtualized infrastructures.

Type Symbol	Description
node	denotes the superclass of types in $\mathbb{T}_{\mathsf{N}}$.
machine	denotes a virtual machine.
hypervisor	denotes a hypervisor on a host or VM.
host	denotes a physical host.
machineOS	denotes an operating system of a virtual machine.
hostOS	denotes an operating system of a physical host.
network	denotes a network component
zone	denotes an isolation zone of an infrastructure.
class	denotes a functional class of similar components.

To analyze topologies, we model virtualized infrastructure configurations as graphs. Whereas the basic graph, called realization, is a unification of vendor-specific elements into abstract nodes, we introduce further graph transformations to model information flow and dependencies.

Definition 3 (Graph Types). *A* graph type *$G \in \{\mathsf{real}, \mathsf{info}, \mathsf{depend}\}$ is a constant identifier for a type of a graph model:*

- real *denotes a realization graph unification of resources and connections thereof.*
- info *denotes a realization graph augmented with colorings modeling topology information flow.*
- depend *denotes a realization graph augmented with colorings modeling sufficient connections to fulfill a resource's dependencies.*

B. Function Symbols and Dependent Terms

Definition 4 (Function Symbols). *Σ contains a finite set of fixed* function symbols.

- $\mathsf{pair}(A, B)$ *denotes a pair.*
- $\mathsf{contains}(S, E)$ *denotes a untyped set membership relationship of a set S and element E.*
- $\mathsf{matches}(I, R)$ *denotes the correspondence between an element of the ideal world I and the real world R. Both elements I and R must have the same type.*
- $\mathsf{edge}([G : \mathsf{real}]; A, B)$ *is a predicate, which denotes the existence of a single edge between A and B with respect to an (optional) graph type G.*
- $\mathsf{connected}([G : \mathsf{real}]; A, B)$ *is a predicate, denotes existence of a path between A and B, respect to an (optional) graph type G.*
- $\mathsf{paths}([G : \mathsf{real}]; A, B)$ *denotes the complete search of all paths between A and B, with respect to an optional graph type G. The resulting type of the function is a set of edge pair sets.*

The notation $[A : v]$ denotes an optional argument A with default constant value v.

Observe that the graph functions allow an optional graph type argument G (Definition 3), which specifies the graph type the function is applied to.

We introduce the notion of *dependent terms* to model access to resource properties, such as IP address $\mathsf{ipadr}(M)$ or image type $\mathsf{imagetype}(M)$ of a machine M.

Definition 5 (Dependent Term Function Symbols)**.** *A* dependent term *is a function symbol denoting the mapping of constant values to atomic terms. Σ includes a fixed set of constant symbols for dependent terms.*

C. Facts, State and Conditions

VALID aims at reasoning over secure and insecure states of a cloud topology, which we model as a set of known facts.

Definition 6 (Facts and State)**.** *A* Fact *represents a Boolean piece of knowledge: it can be either* true *or* false*. A state is a set of ground facts. We express such sets by a dot-operator ("."), that is, a commutative, associative, idempotent operator, which joins all elements of a state.*

Conditions restrict state terms with auxiliary predicates:

Definition 7 (Condition)**.** *A condition is an inequalities predicate over terms. We define the condition function symbols for equality* $\mathsf{equal}(Term, Term)$ *and less-or-equal* $\mathsf{leq}(Term, Term)$ *over terms as well as negation* $\mathsf{not}(Condition)$ *and conjunction operator* $\&\ Condition$ *over conditions with their natural semantics.*

D. State Transitions

In general, an IF specification consists of an initial state and a finite set of *transition rules*, defining a transition relation.

Definition 8 (Transition Rules)**.** *Transition rules have form*

$$PF.NF \quad C \quad =\!\![V]\!\!\Rightarrow \quad RF$$

where

- *PF and RF are sets of facts, NF is a set of* negated facts *of the form $not(f)$ where f is a fact,*
- *C is a set of conditions and*
- *V is a set of variables.*

We distinguish the left-hand side (LHS) defining the preceding state and the right-hand side (RHS) defining the result state. The variables V are existentially quantified in the rule to introduce fresh variables during transitions. RF defines the resulting facts. The variables of RF must be a subset of the variables of the positive facts PF and the existentially quantified variables V.[1]

[1]Note that this excludes the variables only occurring in negative facts and conditions.

Note that transition definition does not enforce transition determinism, that is, that result states are unambiguously defined from the preceding state. IF, being a formal language for model checking, focuses on exploring the state space and determining reachability of attack states, possibly following multiple routes.

The paper focuses on specification of security goals over static states and will only specify initial and goal states. We leave analysis of dynamic systems to future work.

E. Goals

We define *goals* by specifying an abstract state which constitutes attaining the goal. For an analysis we *pattern-match* a Fact set modeling the goals constrained by a conditions list against the actual analysis state.

Definition 9 (Goal)**.** *A* goal state *is a set of positive and negative facts constrained by a (potentially empty) condition list. It is specified with a unique identifier, an optional graph type G and a variable list as interface. It has the form:*

$$\mathsf{goal} \quad Identifier\ ([G : \mathsf{real}]; VariableList) := PF.NF \quad C$$

where PF and NF are positive and negative fact sets and C a condition list. The graph type G determines the graph type of unparametrized graph functions used in the goal.

Example 1 (Goal)**.** *Let us consider a simple isolation breach attack state, which matches against a state, in which disjoint zones* ZA *and* ZB *contain machines* MA *and* MB *respectively, and in which there exists an information flow path between these two machines. It is determined as information flow goal by the graph type* info*. Observe that the goal is defined over variables and can match against any state with constant zones and machines fulfilling this relation and that the matching values must be different.*

```
goal isolation_breach (info; ZA,ZB,MA,MB) :=
   contains(ZA,MA).contains(ZB,MB).
     connected(MA,MB)
```

F. Structured Specifications

Specification of our language consist of distinct sections: The *TypesSection* introduces all atomic terms that will be used throughout the analysis. The type section may have two subsections for real and ideal type declarations. The *InitsSection* specifies initial knowledge on entities. For instance, here one would specify properties of machines that can be used for identifying the machine, such as the machine's IP address as Condition over machine properties. Knowledge specified here can be about ideal and real entities. The *RulesSection* specifies the knowledge on the structure of the virtualized infrastructure. For instance, it specifies which machine elements are associated with which isolation zones. Note that the topology specified in this section is particularly important to model the system's ideal state. Finally, the

GoalsSection defines attack and assurance states which are matched against analysis results.

G. Dual Type System

We introduce the declaration of ideal and real types, that is a *dual type system*.

Definition 10 (Dual Types)**.** *For each constant or variable symbol, we explicitly declare symbol to be either universal or restricted to the ideal or real model. A declaration in the top-level of the TypesSection means universal, a declaration in the subsections* idealTypes *and* realTypes *restricts the declaration to the respective model. The* matches$(\cdot,\cdot)$ *fact denotes that two symbols of ideal and real world have a correspondence with each other.*

V. Attack State Definition

We model the security goals from Section II as abstract attack states. In case the state is reached, a tool will alert that the corresponding goal has been breached. This approach aims at security analysis by, for instance, model checking.

To facilitate an actual security analysis, one complements these abstract goals with specifications of the ideal state of the system in two areas: *First*, one defines the initial knowledge on entities (*InitsSection*), that is, properties modeled as dependent terms, such as IP address. *Second*, one defines the knowledge of the ideal structure of the topology (*RulesSection*) as initial state, that is, facts known on contains, matches or edge relations.

A. Operational Correctness

For the operational correctness from Section II-A, we model deployment breach as exemplary attack state.

1) Deployment Breach: Deployment breach considers in how far VMs are placed on an incorrect hypervisor or physical machine.

Definition 11 (Deployment Breach)**.** *A* deployment breach *is an attack state over some virtual machine* M *and two different hosts (*HA, HB*), in which* edge(HA, M)*, i.e.,* M *is hosted on* HA*, is a specified fact, but* edge(HB, M) *was observed.*

```
section types:
  M                          : machine
  subsection idealTypes:
    HA                       : host
  subsection realTypes:
    HB                       : host

section goals:
goal deploymentBreach (real; HA,HB,M) :=
  not(matches(HA,HB)).edge(HA,M).edge(HB,M)
```

After declaring Fact that HA does not match HB, the left-side of the statement contains the matched facts of the ideal world, that is, edge(HA, M), the right side of the statement the observed fact of the real world edge(HB, M).

2) Unreachability: Unreachability is an attack state that there does not exist a path between a machine and a dependent resource in the dependency graph.

Definition 12 (Unreachability)**.** *A* unreachability *is an attack state over some machine* M *and a resource set* $\{\mathsf{RA},\ldots,\mathsf{RN}\}$*, on which* M *depends. The attack state is triggered if no dependency path between* M *and at least one of the needed resources* RI *exists.*

B. Failure Resilience

1) Single Point of Failure:

Definition 13 (Single Point of Failure)**.** *A* single point of failure *is an attack state over any machine* M *and any two different resources (*RA, RB*) with equivalent function. A single point of failure exists if only* path(M, RA) *holds, but* not(path(M, RB)) *for any* RB.

In general, a single point of failure exists if there is only one dependency path between a resource and its dependencies. This requires knowledge what the dependencies of a certain resource (type) are and which other resources can fulfill the same function. For instance, for a network single point of failure, one may consider all network switches that connect to the Internet, independently from the ones connecting to the Intranet. We therefore define different attack state goals for different resource types and model the goals with functional classes of resources fulfilling the same purpose.

```
section types:
  M                              : machine
  NA, NB                         : network
  C                              : class

section goals:
goal singlePoF_Net (depend; NA,NB,M,C) :=
  contains(C,NA).contains(C,NB).connected(M,NA).
  not(connected(M,NB))
```

2) Interdependent Failure Behavior:

Definition 14 (Interdependent Failure Behavior)**.** Interdependent failure behavior *is an attack state over two different machines* (MA, MB) *with the same functional class* C *and* k *pairs of resource and associated class, i.e., a specific implementation, such as:*

$$(\{\mathsf{RA1},\ldots,\mathsf{RAN}\},\mathsf{CRA}),\ldots,(\{\mathsf{RK1},\ldots,\mathsf{RKN}\},\mathsf{CRK})$$

*We have an attack if for any two machines (*MA, MB*) of class* C*, there exists a resource of the same class they both have in their stack.*

C. Isolation

1) Zoning & Isolation Breach: We specify a simple isolation analysis over machines and zones. Machines can be recognized by their properties, for instance an IP or MAC address. By the contains rule, we express that a machine is associated with zone (i.e., that the zone contains the machine).

Definition 15 (Zoning Breach)**.** *A* zoning breach *is an attack state over a pair of machines (*MA, MB*) and zones (*ZA, ZB*), where either* MA *is declared to be in* ZA *and not present, or* MB *is declared not to be in* ZB*, but was found there in the real state.*

```
section types:
  MA, MB                       : machine
  subsection idealTypes:
    ZA, ZB                     : zone
  subsection realTypes:
    ZA0, ZB0                   : zone

section goals:
goal zoningBreach_Missing (info; ZA,ZA0,MA) :=
  matches(ZA,ZA0).contains(ZA,MA).
    not(contains(ZA0,MA))
goal zoningBreach_Unknown (info; ZB,ZB0,MB) :=
  matches(ZB,ZB0).not(contains(ZB,MB)).
    contains(ZB0,MB)
```

Isolation breach is more complex as it incorporates the existence of information flow paths between zones.

Definition 16 (Isolation Breach)**.** *An* isolation breach *is an attack state over any pair-wise different variable machines* (MA, MB) *and zones* (ZA, ZB)*,* MA *in* ZA *and* MB *in* ZB*, in which there exists a path between* MA *and* MB.

```
section types:
  MA, MB                       : machine
  ZA, ZB                       : zone

section goals:
goal isolationBreach (info; ZA,ZB,MA,MB) :=
  contains(ZA,MA).contains(ZB,MB).
    connected(MA,MB)
```

2) Guardian Circumvention: Guardian Circumvention is an attack state corresponding to Guardian Mediation from Section II. It means that there exist paths between machines that are not controlled by a trusted guardian.

Definition 17 (Guardian Circumvention)**.** Guardian circumvention *is an attack state over any pair-wise different variable machines* (MA, MB)*, guardian* G *and zones* (ZA, ZB)*,* MA *in* ZA *and* MB *in* ZB*, in which there exists a path between* MA *and* MB*, which does not contain the guardian* G*. The attack state naturally extends to a set of multiple guardians.*

VI. Virtualization Assurance Tool

We report that we have implemented a virtualized systems assurance tool, which is able to discover heterogeneous virtualized infrastructures, such as ones based on Xen and VMware, and build up a unified graph representation thereof. The tool provides mechanisms for graph operations and information flow analysis. *VALID* needs to be integrated into such a tool for diagnosis purposes, that is, matching the security goals against the currently deployed system. We built a parser for our language grammar using ANTLR[2], a Java parser generator. We aim at integrating the parser as well as *VALID*-specific analysis capabilities into the assurance tool as future work. To project a *VALID* policy onto native IF as well as use it with an existing IF tool, our assurance tool has to resolve graph function symbols (edge, connected, paths) into the set of all valid graph assignments. In addition, the tool needs to translate knowledge about the discovered real state into policy statements about real entity properties and topology.

VII. Related Work

Automated network infrastructure analysis: Narain et al. [17] analyze network infrastructures with regard to *single point of failure* using a formal modeling language. In contrast, our approach focuses on providing a generic language to express a variety of high-level security goals, among them the absence of single point of failure. Previous work has also analyzed network reachability in an automated way, for example, [22] for IP networks, [13] for VLANs, and [8] for cloud configurations. In terms of network manageability and configuration management, Ballani and Francis [5] propose a deployment language that overcomes the complexity of the low-level configuration. It allows the specification of high-level configuration goals to improve the manageability and was applied to network tunnels. Narain [16] proposes modeling a network configuration using a formal language and do automated reasoning on this formal model.

Formal languages for security policies and modeling: Ponder [9] is an object-oriented formal specification language for access control policies and role management in distributed systems. However, it does not aim at expressing high-level security goals for virtualized infrastructure topologies. Kagal et al. [12] present a policy language for pervasive computing, which is similar to cloud computing environments with regard to their dynamic behavior. It is to express entitlements on actions, services, or conversations of an entity, such as an agent or user. Their implementation is based on Prolog. *Alloy* [11] is a first-order logic modeling language, which is used, among other things, in network infrastructure modeling and analysis [16], [17]. Alloy can express structural properties as relations between objects as well as temporal aspects as dynamic models with states and allowed transitions. It has potential as suitable basis for our cloud assurance language, however we opted for IF as basis because of two reasons. *First*, IF has a strong formal foundation. *Second*, IF is supported by model checkers and theorem provers, such as AVISPA [2], SATMC [3], and OFMC [6] in combination with a fix-point evaluation exportable to Isabelle [19].

Virtualized systems specification languages: The *Open Virtualization Format* (OVF) [10] is a standardized specification language for the packaging and distribution of virtual

[2]www.antlr.org

machines. OVF is used to describe general information and virtual resource usage for an individual virtual machine or a virtual appliance consisting of multiple VMs, but not for an entire virtualized infrastructure as in our approach. *Virtual Machine Contracts* [15] are a policy specifications based on OVF that govern the security requirements of a virtual machine, e.g., to specify firewall rules. Similar to OVF, the objective of this language is linked to provisioning rather than expressing high-level security goals on the topological level. On the hypervisor level, *sHype* [21] is an implementation of access and isolation control for virtual machines, which uses a XML-based access control policy[3]. Again, the policy only applies to one entity in the virtualized system, i.e., the hypervisor hosting virtual machines.

VIII. Conclusion and Future Work

We studied virtualized systems security goals in the categories operational correctness, failure resilience, and isolation. We proposed a formal language to express such high-level security goals, which, unlike previous work, covers topological aspects rather than just individual virtual machines. We chose the *Intermediate Format* (IF) as formal foundation of our language because of its support by existing general-purpose model checkers and theorem provers. We demonstrated the ability of our language to efficiently express a diverse set of virtualized systems security goals by giving concrete specifications for a subset of the studied goals.

Further potential future work is to study dynamic models of virtualized infrastructures, in order to capture and analyze configuration changes and state transitions in general.

Acknowledgments

We would like to thank Stefan Berger, Dimitrios Pendarakis, Matthias Schunter and Ray Valdez for the interesting discussions on cloud security properties as well as for their valuable comments on approaches to cloud assurance languages. We are grateful to Sebastian Mödersheim for the enlightening discussions on the Intermediate Formate IF. This research has been partially supported by the TClouds project[4] funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement number ICT-257243.

References

[1] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. Technical report, Cornell University, Ithaca, NY, USA, 1986.

[2] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *CAV*, pages 281–285, 2005.

[3] Alessandro Armando and Luca Compagna. SATMC: A SAT-Based Model Checker for Security Protocols. In *Logics in Artificial Intelligence*, 2004.

[4] AVISPA. The Intermediate Format. Deliverable D2.3, Automated Validation of Internet Security Protocols and Applications (AVISPA), 2003. http://www.avispa-project.org/delivs/2.3/d2-3.pdf.

[5] Hitesh Ballani and Paul Francis. Conman: a step towards network manageability. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 205–216, New York, NY, USA, 2007. ACM.

[6] David Basin, Sebastian Mödersheim, and Luca Viganò. Ofmc: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, June 2005. Published online December 2004.

[7] Stefan Berger, Ramón Cáceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. Tvdc: managing security in the trusted virtual datacenter. *SIGOPS Oper. Syst. Rev.*, 42:40–47, January 2008.

[8] Sören Bleikertz, Matthias Schunter, Christian W. Probst, Dimitrios Pendarakis, and Konrad Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, pages 93–102, New York, NY, USA, 2010. ACM.

[9] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 18–38, London, UK, 2001. Springer-Verlag.

[10] DMTF. Open virtualization format specification. Technical report, DMTF, 2010.

[11] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002.

[12] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '03, pages 63–, Washington, DC, USA, 2003. IEEE Computer Society.

[13] Sunil D. Krothapalli, Xin Sun, Yu-Wei E. Sung, Suan Aik Yeo, and Sanjay G. Rao. A toolkit for automating and visualizing vlan configuration. In *SafeConfig '09: Proceedings of the 2nd ACM workshop on Assurable and usable security configuration*, pages 63–70, New York, NY, USA, 2009. ACM.

[14] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3:125–143, March 1977.

[15] Jeanna Matthews, Tal Garfinkel, Christofer Hoff, and Jeff Wheeler. Virtual machine contracts for datacenter and cloud computing environments. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 25–30, New York, NY, USA, 2009. ACM.

[16] Sanjai Narain. Network configuration management via model finding. In *Proceedings of the 19th conference on Large Installation System Administration Conference - Volume 19*, LISA '05, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[17] Sanjai Narain, Y.-H. Alice Cheng, Alex Poylisher, and Rajesh Talpade. Network single point of failure analysis via model finding. In *Proceedings of First Alloy Workshop*, 2006.

[18] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, Berkeley, CA, USA, 2003. USENIX Association.

[19] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994.

[20] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, New York, NY, USA, 2009. ACM.

[21] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, Stefan Berger, Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Doorn, John Linwood, and Griffin Stefan Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. Technical Report RC23511, IBM Research, 2005.

[22] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks, 2004.

[3] Xen User Manual, Section 10.3.

[4] http://www.tclouds-project.eu