Automated Verification of Virtualized Infrastructures*

Sören Bleikertz
IBM Research – Zurich
sbl@zurich.ibm.com

Thomas Groß
University of Newcastle upon Tyne
thomasgross@acm.org

Sebastian Mödersheim
DTU Informatics
samo@imm.dtu.dk

ABSTRACT

Virtualized infrastructures and clouds present new challenges for security analysis and formal verification: they are complex environments that continuously change their shape, and that give rise to non-trivial security goals such as isolation and failure resilience requirements. We present a platform that connects declarative and expressive description languages with state-of-the art verification methods. The languages integrate homogeneously descriptions of virtualized infrastructures, their transformations, their desired goals, and evaluation strategies. The different verification tools range from model checking to theorem proving; this allows us to exploit the complementary strengths of methods, and also to understand how to best represent the analysis problems in different contexts. We consider first the static case where the topology of the virtual infrastructure is fixed and demonstrate that our platform allows for the declarative specification of a large class of properties. Even though tools that are specialized to checking particular properties perform better than our generic approach, we show with a real-world case study that our approach is practically feasible. We finally consider also the dynamic case where the intruder can actively change the topology (by migrating machines). The combination of a complex topology and changes to it by an intruder is a problem that lies beyond the scope of previous analysis tools and to which we can give first positive verification results.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security,Verification

CCSW'11, October 21, 2011, Chicago, Illinois, USA.

Keywords

virtualization, cloud, isolation, information flow, assurance, formal specification, system verification

1. INTRODUCTION

Virtualized infrastructures and clouds form complex and rapidly evolving environments that can be impacted by a variety of security problems. Manual configuration as well as security analysis often capitulate in face of these ever-changing complex systems. The need for automated security assurance analysis is immediate. Given the volatility of virtualized infrastructure configurations as well as the diversity of desired security goals, specialized analysis tools—even though having performance advantages—have limited benefits.

As a general approach, we propose to first specify abstract security goals as *desired state* for a virtualized infrastructure in a formal language. For instance, goals can be in the areas *operational correctness* (e.g., "Are all VMs deployed on their intended clusters?"), *failure resilience* (e.g., "Does the infrastructure provide enough redundancy for critical components?") or *isolation* (e.g., "Are VMs of different security zones isolated from each other?"). Second, we employ a generic analysis tool to evaluate the *actual state*, i.e., the virtualized infrastructure configuration, against this desired state. Thus, we obtain an automated analysis mechanism that can check the configuration—and configuration changes—against a high-level security policy.

Such an automated analysis can cover two scopes: in the *static* case, we analyze a single state of a virtualized infrastructure against the desired properties. In the *dynamic* case, we consider the actual configuration as a start state and consider transitions that can change this configuration. In our example, we consider in particular changes that an intruder can make to the network (within the limits of his access rights), e.g., by migrating VMs to other security zones. The question is whether we can reach an attack state in this way, i.e., a current configuration of the system that violates the required security properties. The dynamic case is a generalization of the static case that can only be handled by the model-checking tools.

From engagements with customers running large-scale virtualized infrastructures, we learned that they are interested in a broad range of security goals. Specialized tools can be applied to a subset of these security goals, as we already demonstrated in previous research (cf. [7]) for security zone isolation. However, a general approach is desired that can cover this broad range of security requirements.

Our goal is to establish general-purpose verification methods as an automated tool for security assurance of virtualized infrastructures. We present a platform that connects declarative and expressive description languages with state-of-the art verification methods. With such a platform we can benefit from the variety of existing methods and recent advances such as those in the field of SMT solving. As desired state specification, we take security assurance goals in the formal language *VALID* [6] as inputs. As actual state, we lift the configuration of a heterogeneous virtualized infrastructure to a unified graph model. For this, we employ a security assurance analysis tool called SAVE [7], which also computes graph coloring overlays, that model, e.g., information flow. We develop a translator that connects these descriptions with the various state-of-the art verification tools. The translation involves adapting the verification problem to the domain of the respective tool, and property-preserving simplifications and abstractions to support the verification. In particular, the translation does not add false positives or false negatives to the model.

In this paper we demonstrate that model-checking of cloud infrastructures is in general possible, and we exemplify our approach by studying three examples: *zone isolation*, *secure migration*, and *absence of single point of failure* on the network level. The first example is a static case, which asks whether machines from different security zones are somehow connected in an information flow graph. The relevancy of this case was confirmed in a case study with a financial institution. The second example is of dynamic nature, and asks whether an intruder with rights to migrate VMs can reach an attack state, either by migrating the machine through an insecure network (thereby modifying the VM) or to a physical machine he controls. Secure migration as an example is used to show our first result in verifying dynamic problems, which are in our future interest. The last example belongs to the static case, and we consider that between certain machines we must have redundant network links. Studying a broader range of scenarios using our proposed general approach is left as future work.

1.1 Contributions

We are the first to apply general-purpose model-checking for the analysis of general security properties of virtualized infrastructures. We propose the first analysis machinery that can check the actual state of arbitrary heterogeneous infrastructure clouds against abstract security goals specified in a formal language. Our approach covers static analysis as well as dynamic analysis and uses a versatile portfolio of problem solver back-ends to benefit from their different solution strategies (fix-point evaluation, resolution, etc.).

We believe that our experiments with different analysis strategies (Horn clauses, transition rules) are of independent interest, because the problem instances for security assurance of virtualized infrastructures are structured differently than traditional application domains of model checkers, notably security protocols. In addition, we gained some insights on the complexity relations of different problem classes.

As a case study, we successfully model checked a sizable[1] production infrastructure of a global financial institution against the zone isolation goal. We have previously analyzed this infrastructure extensively with specialized tools and found the same problems with this generic approach. We report that our different optimizations allowed us to improve the performance by several orders of magnitude: whereas the unoptimized problem instances did not terminate within several hours, the optimized problem instances completed the analysis in the order of seconds.

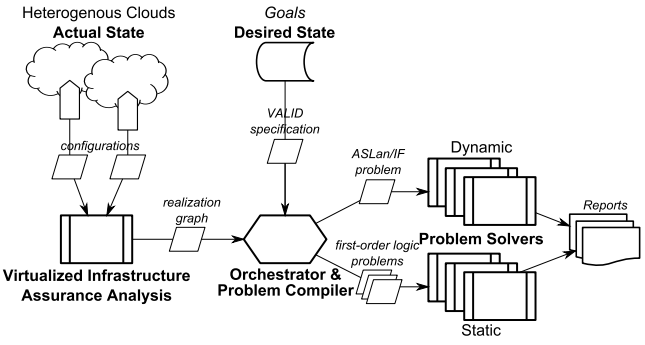

Figure 1: Architecture for model checking of general security properties of virtualized infrastructures.

We note that some of the authors have already made contributions in this area, upon which this work builds. Bleikertz et al. [7] introduced an analysis system for virtualized infrastructures, called *SAVE*, which models configurations in a graph representation and runs graph-coloring based information flow analysis on this representation. Bleikertz and Groß introduced a domain-specific language, called *VALID*, which allows us to specify security goals for virtualized infrastructures in formal terms. This work makes distinct new contributions by introducing analysis based on general-purpose model checking on SAVE's graph representation against VALID specifications. The system presented in this paper goes far beyond information flow analysis of [7] by enabling validation of a wide range of security goals.

1.2 Architecture

We aim at the evaluation of an actual state against a desired state, for which we employ a tool architecture illustrated in Figure 1. To specify a *desired state*, we formulate general security goals in *VALID* [6], a language for security assurance of virtualized infrastructures.

To obtain the *actual state* of a virtualized infrastructure, we employ a tool for *assurance analysis of virtualized infrastructures*, called SAVE [7]. It comes with discovery probes for heterogeneous clouds such as VMware, Xen, pHyp, etc. and takes their proprietary configuration data as inputs. It lifts the configuration data to a unified graph representation of the virtualized infrastructure (the *realization model*) and computes transitive closures over a graph coloring model for information flow tracing. SAVE outputs its graph representations as the actual state of our analysis.

We use and compare several state-of-the-art tools for automated verification. The first is the AVANTSSAR platform[2]; it consists of three verification backends, OFMC [4], CL-Atse [18], and SAT-MC [1], that all have the common input language ASLan (AVANTSSAR Specification Language). We have focused here on OFMC and made initial experiments with the other two; due to lack of source code availability and lack of support of Horn clauses in current SAT-MC, we could not run CL-AtSe and SAT-MC on the large scale case

[1]approximately 1,300 VMs, 25,0000 nodes and 30,000 edges

[2]`http://www.avantssar.eu/`

study through their web-interface. The particular strength of AVANTSSAR is that we can model a dynamic network and check whether a property holds in all reachable states of the network. For the simpler case of analyzing a static network, a broader range of tools is applicable as we can express verification as deducibility problems in first-order Horn clauses. We consider here the automatic first-order theorem prover SPASS [19][3] and the protocol verifier ProVerif [5][4]. We also made initial experiments with the SuccinctSolver [14][5]. In general, our hope is that tools based on different methods can have complementary strengths and connecting them allows us to benefit from all advances of the tools.

As the key component for the actual/desired state analysis, we develop a *compiler* that takes the graph representation of the actual state and the desired state specification in *VALID* as inputs and compiles problem instances for the solver back-ends. It refines the graph representation (e.g., by abstracting from nodes that cannot affect the analysis goal or by introducing "fast lanes"), compiles a term algebra from it, and enhances this problem instance with an analysis strategy (such as, Horn clauses or intruder transition rules) and the goal specified in *VALID*[6].

2. INFORMATION FLOW ANALYSIS TOOL PRELIMINARIES

We use an analysis tool for information flow analysis of virtualized infrastructures [7] to discover the actual configuration of a virtualized infrastructure, abstract it into a unified graph model and determine potential information flow by transitive closure over graph coloring. This tool will provide the bases for the term algebra to describe the actual state. It proceeds in the following phases[7]: The first phase of building a graph model is realized using a discovery step that extracts configuration information from heterogeneous virtualized systems, and a translation step that unifies the configuration aspects in one graph model. For the subsequent analysis, we apply the graph coloring algorithm defined in [7] parametrized by a set of traversal rules and a zone definition. The resulting colored graph model is the actual state input for the compiler to be verified against the desired state security polices.

Discovery.

The goal of the discovery phase is to retrieve sufficient information about the configuration of the target virtualized infrastructure. For this matter, platform-specific data is obtained through APIs such as VMware VI, XenAPI, or libVirt, and then aggregated in one discovery XML file. It contains information, among others, about the virtual machines, virtual networks, and storage in a platform-specific representation. The target virtualized infrastructure, for which we will discover its configuration, is specified either as a set of individual physical machines and their IP addresses, or as one management host that is responsible for the infrastructure. Additionally, associated API or login credentials need to be specified. For each physical or management host given in the infrastructure specification, we will employ a set of discover probes that are able to gather different aspects of the configuration.

We illustrate the discovery procedure with VMware as example. Here, the discovery probe connects to *vCenter* to extract all configuration information of the managed resources. It does so by querying the VMware API with the retrieveAllTheManagedObjectReferences() call, which provides a complete iteration of all instances of ExtensibleManagedObject, a base class from which other managed objects are derived. We ensure completeness by fully serializing the entire object iteration into the discovery XML file, including all attributes.

Transformation into a Graph Model.

We translate the discovered platform-specific configuration into a unified graph representation of the virtualization infrastructure, the *realization model* (cf. [7] for the formal specification of the graph model). It expresses the detailed configuration of the various virtualization systems and includes the physical machine, virtual machine, storage, and network details as vertices. We generate the realization model by a translation of the platform-specific discovery data. This is done by so-called *mapping rules* that obtain platform-specific configuration data and output elements of our cross-platform realization model. Our tool then stitches these fragments from different probes into a unified model that embodies the fabric of the entire virtualization infrastructure and configuration.

Again, we illustrate this process for a VMware discovery. Each mapping rule embodies knowledge of VMware's ontology of virtualized resources to configuration names, for instance, that VMware calls storage configuration entries storageDevice. We have a mapping rule that maps VMware-specific configuration entries to the unified type and, therefore, establishes a node in the realization model graph. We obtain a complete iteration of elements of these types as graph nodes. The mapping rules also establish the edges in the realization model. In the VMware case, the edges are encoded implicitly by XML hierarchy (for instance, that a VM is part of a physical host) as well as explicitly by Managed Object References (MOR). The mapping rules establish edges in the realization model for all hierarchy-links and for all MOR-links between configuration entries for realization model types.

This approach obtains a complete graph with respect to realization model types. Observe that configuration entries that are not related to realization model types are not represented in the graph. This may introduce false negatives if there exist unknown devices that yield further information flow edges. To test this, we can introduce a default mapping rule to include all unrecognized configuration entries as dummy node and all respective MOR links as edges.

Coloring Through Graph Traversal.

The graph traversal phase obtains a realization model and a set of information source vertices with their designated colors as input. The graph coloring outputs a colored realization model, where a color is added to a node if permitted by an appropriate traversal rule. We apply a first-matching algorithm to select the appropriate traversal rule for a given pair of vertices.

[3] `http://www.spass-prover.org/`

[4] `http://www.proverif.ens.fr/`

[5] `http://www.imm.dtu.dk/cs_SuccinctSolver/`

[6] The AVANTSSAR tools accept *VALID* goals out of the box, we translate the goals for the other tools

[7] For further details about this process we refer the reader to [7].

We use the following three type of traversal rules that are stored in a ordered list. *Flow rules* model the knowledge that information can flow from one type of node to another if an edge exists. E.g., a VM can send information onto a connected network. We represent these rules by "follow". *Isolation rules* model the knowledge that certain edges between trusted nodes do not allow information flow. E.g., a trusted firewall is known to isolate, i.e., information does not flow from the network into the firewall. We represent these rules by "stop". *Default rule* Whenever two types are not covered by an isolation or flow rule, then we default to "follow". In order to be on the safe side, we assume that flow is possible along this unknown type of edges. An example of a set of traversal rules can be found in [7].

Output Actual State.

As actual state formulation for our subsequent analysis, the tool outputs different kinds of unified graph models of the infrastructure. We call the graph with the topology of the entire infrastructure realization model; as we will see in Section 3, this models the graph type real of the desired state specification in *VALID*. In addition, we obtain subgraphs of the topology that are reachable by a color in the information flown analysis; they model the graph type info of the desired state specification.

3. LANGUAGE PRELIMINARIES

We give a brief overview of the languages *VALID* and ASLan which are at the core of our formal models. ASLan stands for *AVANTSSAR Specification Language* [2], a set-rewriting based formalism for specification of infinite state transition systems dedicated to security of distributed systems. ASLan is an extension of the AVISPA Intermediate Format [3]; one of the key extensions of ASLan is the integration of Horn-clauses that allow for complex evaluations within every state of the transition system.

The Virtualization Assurance Language for Isolation and Deployment (*VALID*) [6] is a formal language building upon ASLan/IF for specifying security assurance goals of virtualized infrastructures. It is a domain-specific extension and customization of ASLan, in particular introducing a typing system tailored to the needs of virtualized infrastructures and graph-based analysis. Being close to ASLan, it is relatively well-suited for the connection with the model-checking tools of the AVANTSSAR platform.

We describe the two languages along-side (as their structure and meaning is very similar) and highlight the differences.

Term Algebra.

At the core of ASLan and *VALID* is a term algebra over a signature Σ and variable symbols $\mathcal{V}$. In concrete syntax, all constant and function symbols of Σ are alphanumeric identifiers that start with a lower-case letter, while variable symbols of $\mathcal{V}$ start with upper-case letters. We typeset ASLan/*VALID* elements in sans−serif. We interpret terms in the *free algebra*, i.e., syntactically different terms represent different values. In particular, two different constants always represent different entities. We use standard notions of terms such as *ground* (terms without variables), *substitution*, and *matching*.

In ASLan, terms represent usually (cryptographic) messages where constants can be the identifiers of participants, cryptographic keys, etc., and functions represent cryptographic operations like symmetric encryption. In *VALID*, in contrast, the constants denote the elements of the virtualized infrastructure such as virtual and physical machines, switches, and zones. We rarely deal with composed terms, and use only constants and variables.

VALID's Type System.

All constants in *VALID* have a type, according to the following type system:

DEFINITION 1 (TYPE SYSTEM). *We have a finite set of* type symbols *(disjoint from variable and constant symbols) that include for this paper the following:*

$$\mathbb{T} := \{\mathsf{node}, \mathsf{machine}, \mathsf{host}, \mathsf{network}, \mathsf{zone}\}$$

We also have an acyclic subtype relation between types; here, all the types machine, host, *and* network *are subtypes of type* node.

A valid specification must contain one type declaration *for every used constant symbol, and at most one for every used variable. All variables without type declaration are* untyped. *Compound terms are always untyped. A typed variable can only be matched against a constant of the same type or of a subtype. A type declaration that term t has type τ is denoted by $t : \tau$.*

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 2 (GRAPH TYPES). *A* graph type $G \in \{\mathsf{real}, \mathsf{info}, \mathsf{depend}, \mathsf{net}\}$ *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.*
- net *denotes a realization graph augmented with colorings modeling network topology information flow.*

Facts and States.

The next layer of ASLan and *VALID* are *facts* (aka *predicates*) expressing relationships between terms. We use in this work the following (untyped) signature of facts symbols (disjoint from constant, variable, and type symbols) with their intuitive meaning:

- contains(Z, M) where typically Z and M are constant or variable symbols of type zone and machine, respectively. This denotes that machine M belongs to zone Z.
- 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.
- connected$([G : \mathsf{real}]; A, B)$ is a predicate, which denotes existence of a path between A and B, respect to an (optional) graph type G.

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

There are further predicate symbols used in *VALID* that we do not discuss here for brevity, such as paths used to iterate over all paths, and matches used to relate ideal and real nodes.

A *state* is a finite set of ground facts that hold true in the state (and all other facts are false). We denote states using the an enumeration of facts separated by "." (which technically can be regarded as a commutative, associative, and idempotent operator).

Rules and Goals.

An ASLan specification consists of an *initial state*, a set of *rules* that give rise to a transition relation, and a set of *goals* that describe a set of states, usually the violations of the security properties.[8] The security analysis shall then determine whether a goal state is reachable from the initial state by using the rules. Moreover, one may add Horn clauses to specify immediate consequences within a single state which we discuss in more detail below.

The rules have the form $PF.NF.C \Rightarrow RF$ where PF and RF are sets of facts, NF is a set of negative facts (denoted using the ASLan operator $\mathsf{not}(\cdot)$), and C is a set of inequalities on terms. The variables of RF must be a subset of the variables of PF. Such a rule is interpreted as follows: we can make a transition from state S to state S' if S contains a match for all "positive" facts of PF, does not contain any instance that can match a negative fact of RF, and the inequalities of C do hold under the given match. (More formally, the variables of PF are thus existentially quantified, and the ones that only occur in NF and C are universally quantified.) The successor state is obtained by removing the matched positive facts of PF and adding the RF under the matching substitution.

For example, the following rule expresses that, if an intruder resides at a node N and there is an edge from N to another node M and M is not contained in a particular zone z, then the intruder can move to M:

$$\mathsf{intruderAt(N).edge(N,M).not(contains(z,M))} \Rightarrow \mathsf{edge(N,M).intruderAt(M)}$$

Upon this transition, the fact intruderAt(N) is deleted (because it is not repeated on the right-hand side); the fact edge(N, M) remains in the graph because it *is* repeated on the right-hand side.

Goals are quite similar to rules in that have the form $PF.NF.C$ (like a rule without right-hand side) and by the same semantics as rules characterize a set of states, usually "bad" states for state-based safety properties. In *VALID*, goal specifications are also labeled with a graph type G.

Horn Clauses.

ASLan introduced the specification of Horn clauses to the transition system to allow for specifying immediate consequences within a state. One of the main application is the formalization of access control policies: access rights can be expressed as a direct consequence of other facts that express for instance that an employee is a member of particular group. Horn clauses and the state transition system can mutually interact. First, a transition can change the facts that currently hold (e.g., an employee changes to another group) which has immediate consequences for the access rights via the Horn clauses. Second, the fact representing the (current) access decision can be the condition of another transition rule (where an employee requests access to a resource). In our context, we can also use the Horn clauses to formalize properties of the current graph. E.g., to formalize that connected() is the symmetric transitive closure of the edge() predicate we can simply specify:

$$\mathsf{connected(A,B)} :- \mathsf{edge(A,B)}$$
$$\mathsf{connected(B,A)} :- \mathsf{edge(A,B)}$$
$$\mathsf{connected(A,C)} :- \mathsf{edge(A,B).connected(B,C)}$$

Introducing or removing edges upon transitions would automatically change the connected() relation.

[8]Alternatively, ASLan allows for specifying goals also as LTL properties, a feature that we do not use in this paper, however.

4. PROBLEM CLASSES

During our analysis, we found that the analysis goals for virtualized infrastructures can be structured into orthogonal problem classes, and that different problem classes exhibit consistent complexity tendencies for the solver-backends. We consider problem classes with respect to three (syntactical) criteria on attack states and intruder rules: locality, positivity, and dynamics.

4.1 Local vs. Global

Definition 3 (Locality). *We call an attack state* local *if it only exhibits state facts that will be part of the initial state, e.g.,* edge() *and* contains(). *We call an attack state* global *if it exhibits state facts that must be derived by an evaluation over the topology (e.g.,* connected()*). We use these terms for the corresponding problem instances, as well.*

Secure migration—in the sense that the intruder cannot reach a state in which he controls the physical host to which a VM was migrated—is a local problem, because the attack state will be formulated on the edge() statement between these components.[9] Zone isolation mentioned in the introduction is an example of a global problem, because it needs to consider the connections through-out the topology.

All other factors equal, we conjecture that local problems can be consistently checked more efficiently than global problems. We also conjecture a positive performance correlation between Horn clause based models and problem solvers with local problems, and between transition based models and problem solvers with global problems.

4.2 Positive vs. Negative Attack States

Attack states formulated in *VALID* can contain positive as well as negative facts.

Definition 4 (Positivity). *We call an attack state* positive *if it exclusively contains positive state facts. We call an attack state* negative *if it contains at least one negative state fact. We use these terms for the corresponding problem instances, as well.*

[9]Another example for a local problem is machine placement specified in [6], the question whether each VM in the actual state has an edge to the physical host specified in the desired state.

The secure migration and zone isolation examples are positive problems. A negative attack state is, for instance, the guardian mediation introduced in [6], which is fulfilled if there exists any connection between a machine and a network that is *not* mediated by a guardian (firewall).

All other factors equal, we conjecture that positive problems can be checked more efficiently than negative problems.

4.3 Static vs. Dynamic

We consider problems that are statically checking whether the actual state fulfills a desired state. By introducing additional transition rules we can allow the intruder to transform the virtualized infrastructure to reach an attack state, and therefore introduce dynamics.

Definition 5 (Dynamics). *We call a problem instance* static *if its transition rules and Horn clauses only include topology traversal over the initial state. We call a problem instance* dynamic *if it contains transition rules or Horn clauses that model intruder capabilities to change the initial state.*

Many example problems presented in [6] (machine placement, zone isolation, guardian mediation) are static in first instantiation. As soon as we extend the intruder rules/clauses with rights to, e.g., start, stop or migrate machines or to reconnect networks/storage, we obtain dynamic problems. Secure migration introduced above is a dynamic problem.

All other factors equal, we conjecture that static problems can be checked more efficiently than dynamic problems. In the static case, it is more efficient to check for several attack states than in the dynamic case. We conjecture that first-order logic models and tools will have an advantage at static problems, whereas transition based models and tools will have an advantage at dynamic problems.

5. COMPILING PROBLEM INSTANCES

This section discusses how we compile problem instances for the solver back-ends, thus, explains the compiler which is a key component of the architecture in Section 1.2. The compiler receives the following inputs: the realization model or derivatives as a graph representation of the actual state and a *VALID* policy as representation of the desired state.

The success and efficiency of the solver back-ends are largely determined by the initial size of the problem instance, by solution strategies that limit the search tree complexity, and by problem formulations that match the solvers' capabilities. Therefore, the compiler must strive for a significant complexity reduction while maintaining generality. Because we target sizable real-world infrastructures, the initial problem size may easily be in the order of tens of thousands of nodes and the compiler's pruning prove crucial.

The compiler works in multiple phases:

- *Graph Transformation:* Reducing the complexity of the graph and representing it as term algebra facts.
- *Strategy Amendment:* Introducing sensible analysis strategies into the problem instance that match the solver's strengths'.

5.1 Graph Transformation

A (colored) realization model input consists of high-level nodes, such as machine, and low-level nodes, such has ipInterface, as well as edges that model the connections between these components. In general, we aim at representing the edges of this graph as edge() facts in term algebra and give the problem solvers means to derive graph facts, notably connected().

Real-world virtualized infrastructures consist of tens of thousands low-level components and similarly many edges, an initial complexity that could easily overwhelm the solver back-ends. Therefore, we support the solver back-ends in traversing these graphs efficiently by either abstracting from low-level nodes not impacting the analysis or introducing "fast lanes" into the graph:

Definition 6 (Optimization: Graph Refinement). *For all adjacent high-level components, such as* machine *or* host*, connected though a sub-graph with low-level components with degree smaller than three, we replace the subgraph by edges maintaining the same connectivity. Similarly, "fast lanes" added to the graph allow the solver back-ends to reach other segments of the graph with fewer steps.*

The graph refinement maintains analysis generality if the pruned node types do neither occur in attack states nor in intruder or topology transformation rules/clauses.

5.2 Strategy Amendment

Graph Traversal.

A major part of the solver's strategy will depend on how the graph traversal is modeled, which we express by connected() facts derived from the edge() facts:

$$\mathsf{edge}(A,B) \Rightarrow \mathsf{edge}(A,B).\mathsf{connected}(A,B).\mathsf{connected}(B,A)$$

$$\mathsf{edge}(A,B).\mathsf{connected}(B,C) \Rightarrow \mathsf{edge}(A,B).\mathsf{connected}(B,C).\mathsf{connected}(A,C)$$

Observe that this formulation to compute the connected() relation does not change the graph, i.e., edge() facts are neither introduced or removed by these rules. While this is a necessity for all evaluations of the graph in the dynamic case, in the static case, we can formalize evaluation procedures that *do* change the graph, for instance rules that remove edges from the graph as soon as they were visited by the evaluation. Our benchmarks show that such changes can improve the performance of our zone isolation example, however only slightly. Moreover, such "graph-consuming" strategies are helpful for formalizing some advanced properties below.

We propose an additional translation, which reduces the state complexity significantly. In this case, we imagine an intruder tries to traverse the topology from some start-point and "obtain" nodes he has access to. This avoids the binary fact connected and instead uses a unary fact intruderHas to represent all members of the largest connected subgraph that contains the intruder start point.

The transition rules are as follows:

$$\mathsf{intruderHas}(A).\mathsf{edge}(A,B).\mathsf{not}(\mathsf{intruderHas}(B)) \Rightarrow \mathsf{intruderHas}(A).\mathsf{intruderHas}(B).\mathsf{edge}(A,B)$$

$$\mathsf{intruderHas}(A).\mathsf{edge}(B,A).\mathsf{not}(\mathsf{intruderHas}(B)) \Rightarrow \mathsf{intruderHas}(A).\mathsf{intruderHas}(B).\mathsf{edge}(B,A)$$

For large graphs, the restriction of analyzing such chunks rather than the full connected-relation means substantial savings: roughly speaking, the number of derivable facts is in the worst case linear for the intruderHas strategy, while the number of connected facts is quadratic. This optimization requires, however, that we have to select one start point for

the intruderHas() computation and thus get the verification of isolation from other zones only for that selected start point. In case a connected(A, B) fact is used in a security goal, we can translate it to intruderHas(A).intruderHas(B) for certain goals (e.g. zone isolation).

Depending on the used solver or back-end, the evaluation which nodes the intruder can obtain can either be expressed by a means of transition rules (as above) or as first-order Horn clauses (omitting the not(intruderHas(B)) condition on the left-hand sides).

Dynamic Problems.

In addition to the graph analysis model, we need to introduce intruder rules for the dynamic analysis to model his capabilities to modify the infrastructure. They are highly dependent on the scenario, but can easily be modeled by introducing new facts as well as transition rules or Horn clauses. We exemplify this by modeling the secure migration problem in Section 6.

Encoding Static Problems into FOL.

In case of static problems, such as zone isolation, we do not need to consider transition systems but can rather encode the problem into "static" formalisms like first-order logic (FOL) and alternation-free least fixed point logic (ALFP) for which mature tools exists. We now show that we can effectively use such tools as an alternative to the model-checking approach in the static case. We study the use of the SuccintSolver for (ALFP) [14], the FOL theorem prover SPASS [19] and the protocol verifier ProVerif [5].

The example of zone isolation can be expressed as an initial set of facts representing the graph structure, a set of Horn clauses expressing the graph traversal as shown previously, and a predicate that an intruder can reach a machine in another different security zone.

The SuccintSolver [14] is an effective tool for computing the least fixedpoint (i.e., all facts that are derivable by the ALFP clauses from the given facts) of an ALFP specification. (This fixedpoint is in our case always finite.)

The next tool we use is the generic first-order theorem prover SPASS [19] which is based on resolution. The problem is here that we want a Herbrand model of the symbols (e.g., different constants always represent different elements) which cannot be enforced directly. We thus formulate as a proof goal that an isolation breach can be reached; such a proof exists if and only if this is true in the Herbrand model. For the inequality of zones, we need to specify this as axioms for zones to properly handle the negation w.r.t. the "Herbrand-trick".

We finally consider the ProVerif tool [5] which is also based on resolution but dedicated to security problems formulated by Horn clauses, and therefore often faster than SPASS. Like in SPASS, we have to axiomatically introduce here the inequality of zones, albeit using an uninterpreted predicate symbol (because negation is not possible in ProVerif).

Static Problems beyond FOL.

We now discuss static problems that are beyond the expressiveness of FOL. Consider the goal of the absence of single point of failures for network links, i.e., that a network contains sufficient redundancies, so that failure of a single node does not disrupt communication.[10] More formally, let us consider a network and the dependability constraint $depend(n_1, n_2)$ between two nodes n_1 and n_2. Then we require that there is are at least two disjoint paths (using disjoint nodes) in the network from n_1 to n_2. Even in the static case (when the network topology cannot change) this problem is beyond the expressiveness of first-order logic (as a consequence of the Löwenheim-Skolem theorem, see e.g. [10]).

As a consequence, we cannot use the solvers SPASS, ProVerif, and Succinct Solver. Also, the standard approach to specify the security property as a set of *VALID* or ASLan goals (even using Horn clauses to evaluate the graph) is not applicable, because that would also be FOL expressible relations. However, we can specify a transition system in ASLan to express a game that has a solution (expressed as a set of goal states) if and only if there exists no single point of failure.

We demonstrate this game for the absence of single point of failure in Section 6.

6. MODEL-CHECKING A VIRTUALIZED INFRASTRUCTURE

In this section, we study three example problems, namely zone isolation, secure migration, and absence of single point of failure, and demonstrate how these problems can be analyzed using model-checking. We apply model-checking on small infrastructure examples to demonstrate the approach, and we will analyze a large-scale infrastructure with regard to zone isolation in Section 7.

We structure this section analogously to the architecture Section 1.2 where for each example problem, we first specify the desired state in *VALID* or ASLan goals along with the required language primitives. Second, we introduce the actual state, that is the infrastructure examples we analyze. Third, we discuss specialties of compiling the corresponding problem instance, the problem solvers employed and their output for the analysis.

6.1 Zone Isolation

We consider the following scenario to illustrate the zone isolation security goal: an enterprise network consists of three security zones, namely a *high* security zone containing confidential information, a *base* security zone for regular IT infrastructure, and a *test* security zone. Any machine in one zone should not be able to communicate with a machine from a different zone, and network isolation is realized using VLANs.

Desired State.

To have the solvers check violations of zone isolation, we define an attack state isolation_breach, which asks the question whether any two machines of any two different security zones are connected.

Definition 7 (Goal: Zone Isolation). *The isolation breach attack state matches if any two 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.

[10] [6] considers another goal for single point of failure that can be expressed as a goal state: when a node depends on a particular resource, then it is connected to more than one node to provide that resource.

```
goal isolation_breach(info;ZA,ZB,MA,MB):=
  contains(ZA,MA).contains(ZB,MB).
    connected(MA,MB) & not(equal(ZA,ZB))
```

Furthermore, the *VALID* policy requires a specification of the membership of machines to specific zones. For example, contains(high, vm1) denotes that *vm1* is part of the *high* security zone.

Actual State.

As discussed in Section 2, SAVE discovers the given infrastructure and captures all low-level configuration details and resource associations. SAVE performs an information flow analysis with the different security zones as information sources and produces an information flow graph for the infrastructure.

Model-Checking.

Based on the actual state provided by SAVE, our compiler will generate a representation of the (potentially refined) information flow graph in edge() facts and node constants. Since we are dealing with a static problem, we use the efficient intruderHas modeling for graph traversal, and transform the goal accordingly. The output is ASLan for OFMC and a variety of first-order logic languages used by the static problem solvers.

Suppose the VLAN identifier of a machine *vm2* in the *test* zone was misconfigured and is identical to the VLAN ID of a machine *vm1* from the *high* security zone. OFMC will provide us with such an attack state (reduced for brevity) indicating a zone isolation breach:

```
SUMMARY
  UNSAFE
PROTOCOL
  zone_isolation.if
GOAL
  isolation_breach

% contains(zone(high),node(machine(vm1)))
% contains(zone(test),node(machine(vm2)))
% intruderHas(node(machine(vm1)),i)
% intruderHas(node(machine(vm2)),i)
```

6.2 Secure Migration

Secure migration is a problem often encountered in practise which was also highlighted by Oberheide et al. [15]. Secure Migration is an interesting problem as its very nature requires a dynamic modeling. However, we do not claim to solve it completely with this work, as is a complex endeavor in which many factors (network and storage connections, VLAN associations, correct configuration of VMs, machine contracts, etc.) need to be considered. Still, we want to demonstrate the principles of dynamic analysis with a simplified example of this problem class. We leave a full-scale analysis of secure migration of a production system for future work.

We consider the topology depicted in Figure 2 for our scenario: five hosts, where one is controlled by a malicious administrator, are connected to two networks. The malicious administrator can migrate virtual machines between hosts as indicated by the *migrate* edges. There is one VM running on host *HostA*.

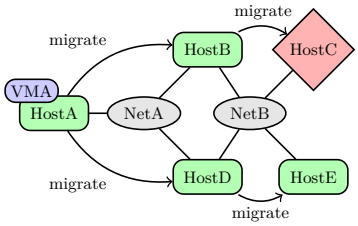

Figure 2: Migration Scenario Topology

Desired State.

We study two exemplary instantiations of the problem of secure migration. The attack state vm_breach asks whether the intruder can migrate a virtual machine from a secure environment to a physical host to which he has root access (in order to perform attacks demonstrated by Rocha et al. [17]). The attack state insecure_migration asks whether an intruder can migrate a VM through an insecure network in order to manipulate the VM (cf. attacks demonstrated by Oberheide et al. [15]).

We define these goals in *VALID*, for which we introduce the unary facts intruderAccess() and root(), and the binary fact migrate(). These model the intruder's access capability set of root access (typically to a given host) and machine migration between two hosts. These facts have the following signature:

$$\begin{aligned}&\mathsf{intruderAccess} : \mathsf{fact} \rightarrow \mathsf{fact}\\ &\mathsf{migrate} : \mathsf{host} * \mathsf{host} \rightarrow \mathsf{fact}\\ &\mathsf{root} : \mathsf{node} \rightarrow \mathsf{fact}\end{aligned}$$

The fact intruderAccess() models the set of all access rights the intruder has, that is, it has the semantic that any term enclosed by the fact belongs to the intruder's access capabilities. The fact root() models administrator rights on the enclosed node.

We model virtual machine migration in the following way.

Definition 8 (Migration). *The capability of migrating a VM* MA *from host* HA *to* HB *is expressed as Horn clause* canMig *that incorporates the intruder access to migrate between these two hosts, that both hosts are connected to the same network* NA*, and one host is running the VM.*

Migration is a transition rule that removes the association of a VM MA *to a host* HA*, and adds an association to a new host* HB *in case fact* canMig *matches.*

$$\mathsf{canMig}(\mathsf{MA},\mathsf{HA},\mathsf{HB},\mathsf{NA}) :- \mathsf{edge}(\mathsf{HA},\mathsf{MA}).\mathsf{edge}(\mathsf{HA},\mathsf{NA}).\mathsf{edge}(\mathsf{HB},\mathsf{NA}).\mathsf{intruderAccess}(\mathsf{migrate}(\mathsf{HA},\mathsf{HB}))$$

$$\mathsf{edge}(\mathsf{MA},\mathsf{HA}).\mathsf{canMig}(\mathsf{MA},\mathsf{HA},\mathsf{HB}) \Rightarrow \mathsf{edge}(\mathsf{MA},\mathsf{HB})$$

The goals are defined in *VALID* in the following way:

Definition 9 (Goal: VM Security). *The VM breach attack state matches if there is a* root() *fact on a host* HA *in the intruder's access capability set and a VM* MA *being connected to the host.*

```
goal vm_breach(real;HA,MA):=
  intruderAccess(root(HA)).
  edge(MA, HA)
```

Definition 10 (Goal: Secure Migration). *The attack state for insecure migration is the following. The intruder can migrate a VM* MA *from host* HA *to* HB*, and he*

has root *access to a host* HC *that is connected to the same network.*

```
goal insecure_migration(net;HA,HB,HC,MA,NA):=
  canMig(MA, HA, HB).
  intruderAccess(root(HC)).
  edge(HA, NA).edge(HB, NA).edge(HC, NA)
```

Actual State.

We model the access capabilities of the intruder for our scenario in the following way.

- intruderAccess(root(hostC))
- intruderAccess(migrate(hostA, hostB))
- intruderAccess(migrate(hostA, hostD))
- intruderAccess(migrate(hostB, hostC))
- intruderAccess(migrate(hostD, hostE))

The network information flow graph for the scenario is generated by SAVE.

Model-Checking.

Unlike in the previous static example, we had to explicitly model the dynamic behavior of the intruder, i.e., machine migration, and its effects on the infrastructure. We modeled that as transition rules with restrictions based on access privileges of the intruder. Since we are dealing with a dynamic problem, we have to use a tool from the AVANTSSAR tool chain, for instance OFMC.

OFMC found the following attack states (reduced for brevity) for our scenario.

```
INPUT
  migration.if
SUMMARY
  ATTACK_FOUND
GOAL: vm_breach

% Reached State:
%
% intruderAccess(root(node(host(hostC))),i)
% edge(node(machine(vma)).node(host(hostC)),i)
```

OFMC finds this attack state for *vm_breach* due to the migration of *VMA* to *HostB*, and then to *HostC*.

```
INPUT
  migration.if
SUMMARY
  ATTACK_FOUND
GOAL: insecure_migration

% Reached State:
%
% canMig(node(machine(vma)).node(host(hostD)).node(↩
    host(hostE)),i)
% intruderAccess(root(node(host(mc))),i)
% edge(node(host(hostD)).node(network(netB)),i)
% edge(node(host(hostE)).node(network(netB)),i)
% edge(node(host(hostC)).node(network(netB)),i)
```

This attack state for *insecure_migration* is reached by the migration of *VMA* to *HostD*, then to *HostE* and intercepted by *HostC* due to the connection to the same network *NetB*.

6.3 Absence of Single Point of Failure

We consider the topology illustrated in Figure 3 for our scenario to demonstrate the absence of single points of failure for network links. We have two hosts that are depended on each other and connected through a combination of three networks.

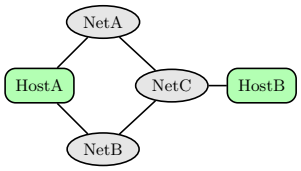

Figure 3: Single Point of Failure Scenario Topology

Desired State.

The goal of the absence of single point of failure for network links is not expressible in FOL, or *VALID* or ASLan goals. Therefore, we construct a game using transitions in ASLan that has a solution if and only if there exists no single point of failure.

This game works as follows for a single dependency constraint $depend(n_1, n_2)$ (if there are several such constraints, one must start each as a separate game). We have two phases in which sets S_1 and S_2 of nodes are collected. In the first phase we start with $S_1 = \{n_1\}$ and non-deterministically follow edges from a member of S_1 to a non-member that we then add, until we have reached n_2 and start the second phase. We begin similarly with $S_2 = \{n_1\}$ and non-deterministically follow an edge from a member of S_2 to a node that is not part of *either* S_1 and S_2 that we add to S_2 until we have reached n_2. Then S_1 and S_2 represent nodes for two disjoint (except for start and end nodes) paths from n_1 to n_2. Since the transition system allows to non-deterministically choose the edge to follow, the goal state $n_2 \in S_2$ is reachable if and only if such disjoint paths exist.

In the following are the transition rules modeling this game. The first one starts the first phase, the second one traverses nodes in the first phase, and the third one terminates the first phase and starts the second phase. The fourth rule traverses nodes in the second phase.

$$\text{not}(\text{round1}).\text{not}(\text{round2}).\text{depend}(A, B) \Rightarrow \text{round1}.\text{depend}(A, B).\text{inS1}(A)$$

$$\text{round1}.\text{depend}(A, B).\text{inS1}(X).\text{edge}(X, Y).\text{not}(\text{inS1}(Y)).\text{not}(\text{equal}(Y, B)) \Rightarrow \text{round1}.\text{depend}(A, B).\text{inS1}(X).\text{inS1}(Y)$$

$$\text{round1}.\text{depend}(A, B).\text{inS1}(X).\text{edge}(X, B) \Rightarrow \text{round2}.\text{depend}(A, B).\text{inS1}(X).\text{inS1}(B).\text{inS2}(A)$$

$$\text{round2}.\text{depend}(A, B).\text{inS2}(X).\text{edge}(X, Y).\text{not}(\text{inS1}(Y)).\text{not}(\text{inS2}(Y)).\text{not}(\text{equal}(Y, B)) \Rightarrow \text{round2}.\text{depend}(A, B).\text{inS2}(X).\text{inS2}(Y)$$

Here we use special facts round1 and round2 to separate the different phases and inS1 and inS2 to denote the members of S_1 and S_2.

The following goal is reached when the second phase terminates, and thereby identified a second disjoint path between A and B.

```
section goals:
    attack_state spof_absence (A,B,X) :=
      round2.depend(A,B).inS2(X).edge(X,B)
```

Edge symmetry is not handled by the previously shown transitions and the goal, and has to be modeled explicitly with another set of transitions and a goal.

For our scenario, we also have to specify the dependency between *HostA* and *HostB* using the depend term.

Actual State.

The network information flow graph for the scenario is generated by SAVE.

Model-Checking.

Since we are dealing with a static problem that cannot be encoded in first-order logic, we modeled this goal in such a way that an attack state is actually a satisfaction of the goal, namely there are no single point of failures. This is contrary to the previous two examples, where an attack state always denoted a breach of a security goal.

For our scenario, the model-checker OFMC will not reach an "attack state", therefore the infrastructure contains a single point of failure. Now we consider connecting *HostB* also to *NetB*, therefore we get a second disjoint path from *HostA* to *HostB*. OFMC produces the following output showing the two disjoint paths (reduced to inS1 and inS2 facts, and re-ordered):

```
INPUT
  spof.if
SUMMARY
  ATTACK_FOUND
GOAL: spof_absence

% Reached State:
%
% inS1(node(host(hostA)),i)
% inS1(node(network(netB)),i)
% inS1(node(host(hostB)),i)
% inS2(node(host(hostA)),i)
% inS2(node(network(netA)),i)
% inS2(node(network(netC)),i)
```

7. CASE STUDY FOR ZONE ISOLATION

In this section, we analyze a real and large-scale production environment of a global financial institution. The infrastructure consists of approximately 1,300 VMs and its realization model modeling all networking and storage resources consists of approximately 25,000 nodes and 30,000 edges. The infrastructure is divided into several security zones, each containing multiple clusters, and models networking up to Layer 2 separation on VLANs and storage providers up to separation on file level. We have already analyzed this virtualized infrastructure extensively with specialized tools and know which attack states to expect. Given the large initial size of the actual state, this case study provides a suitable test environment for the subsequent performance analysis.

Whereas our compiler translates the problem instances to the different static and dynamic problem solvers introduced in Section 1.2, we focus the performance evaluation on three tools SPASS and ProVerif for the static case and OFMC for both the static and dynamic case. We have also performed initial experiments with SAT-MC, CL-AtSe, and Succint-Solver, but could not apply them to the large case study. We analyze various optimization and modeling techniques introduced in Section 5 to establish their effects in practice. We are focusing in this evaluation on two specific clusters (we call them *Cluster1* and *Cluster2*) and their corresponding information flow graphs, for which we know that *Cluster1* has an isolation problem and *Cluster2* is safe.

Graph Refinement.

We first measure the simplification of the information flow graphs for the different clusters in terms of the number of edges and nodes. The information flow graph of *Cluster1* consists of 14386 nodes and 17817 edges. We achieve a reduction of the graph by 13428 nodes and 16860 edges, resulting in a graph with only 958 nodes and 957 edges. The algorithm performs this simplification in 0.18 seconds. *Cluster2* has a smaller information flow graph with 6218 nodes and 7543 edges. The graph reduction completes within 0.06 seconds and results in a graph with 359 nodes and 358 edges.

Zone Isolation.

We are now evaluating the analysis of the zone isolation goal for the large-scale infrastructure. For our evaluation, we consider all analysis cases for the following parameters: attack/safe, simplified/non-simplified graph, and different graph traversal models. *Attack* denotes an isolation breach and *Safe* denotes secure isolation.

For the graph traversal modeling using connected() in form of Horn clauses or transition rules, all tools we are evaluating either run out of memory (OFMC) or do not terminate within our time limit of 4 hours. We therefore focus our detailed performance analysis on our intruderHas graph traversal model with the following analysis cases.

- *Simplified Graph:* Attack **1**, Safe **2**
- *Non-Simplified:* Attack **3**, Safe **4**

The time measurements of the analysis cases for the different tools are depicted in Figure 7.

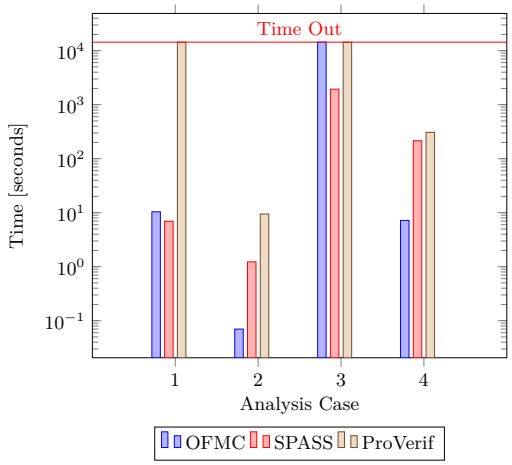

Figure 4: Time measurements (on logarithmic scale) for analysis cases of zone isolation.

The measurements show that ProVerif is only able to analyze the *Safe* configuration, because in the other case it does not terminate within our time frame of 4 hours. Since ProVerif is based similarly on resolution as SPASS (which terminates within the time limit for all problems), we suspect that the pre-processing of rules in ProVerif may be the cause.

OFMC yields good performance results and is very fast for analyzing such a large-scale infrastructure. We noticed

a problem in analyzing the vulnerable cluster with the non-simplified graph, that is OFMC runs out of memory. SPASS terminates for all analysis cases and is faster for case 1 as OFMC.

Discussion.

The analysis of a large-scale infrastructure with regard to the zone isolation goal gave us insights into the efficiency of our modeling and the employed problem solvers. We learned that our initial modeling of connected() facts using Horn clauses or transitions were only applicable for small infrastructures and not for such real-world scenarios. Therefore, we developed the more efficient modeling of using intruderHas() facts for graph traversal, which made the analysis in a reasonable time frame possible. The complexity of this graph traversal is only linear to the number of edges, whereas the graph traversal using connected() yields a quadratic complexity.

Furthermore, we learned that problem solvers were overwhelmed by the detailed modeling of the infrastructure in form of our realization model. In case of security goals concerned with graph connectivity, we developed a graph refinement algorithm that simplifies the realization graph, but preserves its connectivity properties. The combination of efficient graph traversal modeling and graph simplification yielded results in the order of seconds for the analysis of our scenario infrastructure.

In terms of employed problem solvers, SPASS and OFMC performed best for our scenario.

8. RELATED WORK

Virtual systems introduce several new security challenges [9]. Two important drivers that inspired our work is the increase of scale as well as the transient nature of configurations that render continuous validation with a variety of security goals more important.

Narain et al. [13] analyze network infrastructures with regard to *single point of failure* using a formal modeling language. In contrast, our approach focuses on a variety of high-level security goals, among them the absence of single point of failure, that can be evaluated using general-purpose model-checkers. Previous work has also analyzed network reachability in an automated way using specialized tools, for example, [20] for IP networks, [11] for VLANs, and [8] for Amazon cloud configurations. Narain [12] proposes modeling a network configuration using a formal language and do automated reasoning on this formal model. We are extending this concept by considering the entire virtualization infrastructure, not just networking resources. Ritchey et al. [16] employ model-checkers to check for vulnerabilities in networks.

The main differentiation of our work to previous ones is two-fold. First, we have a generic way to specify and verify security goals for virtualized infrastructures rather than specialized analysis. Second, our framework includes the modeling of a dynamic infrastructure, in particular one where the intruder can influence the topology (for instance by migrating machines) to mount an attack. This paper is the first to formally verify security properties of virtualized infrastructures with this dynamic behavior.

9. CONCLUSION AND FUTURE WORK

In this paper we demonstrated our novel approach for the automated verification of virtualized infrastructures. We are able to specify a variety of security goals in a formal language and validate heterogeneous infrastructure against them. We are the first to employ general-purpose model-checker and theorem provers for this matter.

We studied three examples of static and dynamic problems, namely zone isolation, secure migration, and single point of failure. For each problem, we showed how to specify goals in the formal languages and proposed efficient modeling strategies. We successfully demonstrated the automated verification of these examples against small infrastructures. Finally, we also validated a large-scale infrastructure against the zone isolation secure goal and showed the practical feasibility of our approach.

Future work includes the further study of dynamic problems in virtualized infrastructure and their efficient analysis on large-scale infrastructures.

Acknowledgments

Thomas Groß contributed to this research while being at IBM Research – Zurich. This research has been partially supported by the TClouds project[11] funded by the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement number ICT-257243. We thank Piotr Filipiuk for support on the SuccintSolver.

10. REFERENCES

[1] Armando, A., and Compagna, L. Sat-based model-checking for security protocols analysis. *International Journal of Information Security 7* (2008), 3–32.

[2] AVANTSSAR. ASLan final version with dynamic service and policy composition. Deliverable D2.3, Automated Validation of Trust and Security of Service-oriented Architectures (AVANTSSAR), 2010. `http://www.avantssar.eu/pdf/deliverables/avantssar-d2-3.pdf`.

[3] 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`.

[4] Basin, D. A., Mödersheim, S., and Viganò, L. OFMC: A symbolic model checker for security protocols. *Int. J. Inf. Sec. 4*, 3 (2005), 181–208.

[5] Blanchet, B. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)* (Cape Breton, Nova Scotia, Canada, June 2001), IEEE Computer Society, pp. 82–96.

[6] Bleikertz, S., and Gross, T. A Virtualization Assurance Language for Isolation and Deployment. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2011)* (2011). to appear.

[7] Bleikertz, S., Gross, T., Schunter, M., and Eriksson, K. Automated information flow analysis of virtualized infrastructures. In *16th European*

[11] `http://www.tclouds-project.eu`

Symposium on Research in Computer Security (ESORICS'11) (Sep 2011), Springer.

[8] Bleikertz, S., Schunter, M., Probst, C. W., Pendarakis, D., and Eriksson, K. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop* (New York, NY, USA, 2010), CCSW '10, ACM, pp. 93–102.

[9] Garfinkel, T., and Rosenblum, M. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2005), USENIX Association, pp. 20–20.

[10] Huth, M., and Ryan, M. D. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004.

[11] Krothapalli, S. D., Sun, X., Sung, Y.-W. E., Yeo, S. A., and Rao, S. G. A toolkit for automating and visualizing vlan configuration. In *SafeConfig '09: Proceedings of the 2nd ACM workshop on Assurable and usable security configuration* (New York, NY, USA, 2009), ACM, pp. 63–70.

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

[13] Narain, S., Cheng, Y.-H. A., Poylisher, A., and Talpade, R. Network single point of failure analysis via model finding. In *Proceedings of First Alloy Workshop* (2006).

[14] Nielson, F., Nielson, H. R., and Seidl, H. A succinct solver for ALFP. *Nordic J. of Computing 9* (December 2002), 335–372.

[15] Oberheide, J., Cooke, E., and Jahanian, F. Exploiting Live Virtual Machine Migration. In *BlackHat DC Briefings* (Washington DC, February 2008).

[16] Ritchey, R. W., and Ammann, P. Using Model Checking to Analyze Network Vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2000), IEEE Computer Society, pp. 156–.

[17] Rocha, F., and Correia, M. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Proceedings of the 1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV, with DSN'11)* (June 2011).

[18] Turuani, M. The CL-Atse Protocol Analyser. In *Term Rewriting and Applications*, F. Pfenning, Ed., vol. 4098 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 277–286.

[19] Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., and Wischnewski, P. SPASS Version 3.5. In *Automated Deduction CADE-22* (2009), R. Schmidt, Ed., vol. 5663 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 140–145.

[20] Xie, G. G., Zhan, J., Maltz, D. A., Zhang, H., Greenberg, A., Hjalmtysson, G., and Rexford, J. On Static Reachability Analysis of IP Networks, 2004.