RZ 3893 (# ZUR1605-051) 05/17/2016
Computer Science 22 pages

Research Report

Dynamic Information Flow Graphs with Flow Rules

S. Bleikertz*, T. Groß‡, S Mödersheim#

*IBM Research – Zurich
8803 Rüschlikon
Switzerland

‡University of Newcastle upon Tyne

#DTU Compute

Dynamic Information Flow Graphs with Flow Rules

Sören Bleikertz[1], Thomas Groß[2], and Sebastian Mödersheim[3]

[1]IBM Research – Zurich, sbl@zurich.ibm.com
[2]University of Newcastle upon Tyne, thomas.gross@newcastle.ac.uk
[3]DTU Compute, samo@dtu.dk

Abstract

We introduce a static information flow analysis for dynamic systems. Based on user-configurable trust assumptions, our approach computes an information flow graph on top of a system model graph. The edges in this information flow graph are annotated with dependencies on the trust assumptions' conditions, which operate on node attributes and connectivity. A dynamic system model is described as a graph delta of incremental and decremental node and edge changes as well as node attribute changes. Our differential analysis computes the impact of a system model graph delta on the information flow graph based on the information flow edges' dependencies. We apply our approach to the practical and important problem of tenant isolation in dynamic virtualized infrastructures.

1 Introduction

Isolation is a fundamental security requirement in any multi-level security system. The non-interference property [24] is a strict formalization of isolation: Inputs and outputs are classified as either *low* or *high*, and a computation on *low* values must not influence *high* outputs and vice versa. Less strict and practical variants of non-interference have been proposed, which allow for instance mediated communication between different security levels using channel control [35]. Alternatively, access control models for multi-level security systems exist, such as, Biba [5] for integrity, Bell-LaPadula [3] for confidentiality, and Chinese Wall [9] for confidentiality with conflicting parties.

These fundamental security models find their application in practical systems security. For instance in virtualization, the sHype [36] hypervisor mediates inter-VM communication and enforces the aforementioned access control models. Rueda et al. [34] analyzes VM access control policies using information flow graphs to verify inter-VM flows. The TVDc [4] approach enforces access control on the entire virtualized infrastructure level and not just on the hypervisor. The analysis of isolation and mediated inter-VM communication in heterogeneous virtualized infrastructures has also been studied [7]. Similar approaches have been proposed for the Android operating system, such as, domain isolation [11], taint tracking [18], and permission analysis using graph reachability [10].

In general we can classify the isolation approaches as either *static* or *dynamic*. The static approaches operate on a model of the system and compute potential information flows, in order to make a policy decision on illegal flows. The dynamic approaches monitor the running system for actual flows to detect or block illegal ones. A similar classification is done in program analysis where static approaches operate on the source code of the program, and dynamic approaches analyze the executing program. One benefit of the static approach is that we compute all possible flows whereas in the dynamic approach we only see the current actual flows. However, the static approach operates on a model of the system. In dynamic systems we need to ensure that the model is kept in sync with the actual system, otherwise we have delays in the detection of violations or miss transient violations altogether. Previous static approaches, such as [7] and [34], only operate on static system models. In this work we pursue a static analysis approach of dynamic systems based on system change events and a differential analysis. We apply our approach to the case study of isolation in dynamic virtualized infrastructures. However our approach is general enough to be applied also in other domains, such as attacker propagation in digital-physical environments or access control configurations.

Our approach works in two phases: the *initial* phase and the *differential* phase. The initial phase takes a current snapshot of the system modeled as a graph. A directed information flow graph is computed as an overlay graph on top of the system model graph. Using the information flow graph we can compute reachability between any two nodes, in order to verify isolation policies. The information flow edges are constructed based on a set of user-defined flow rules that capture trust assumptions on system elements and their isolation properties. The constructed edges are dependent on the flow rules' conditions on node attributes and connectivity. This lays the foundation for the dynamic analysis because we record the existence requirements for each edge and can verify if these requirements still hold in a changing system. In the differential phase, we obtain a system model change as a graph delta, which describes incremental and decremental node and edge changes as well as node attribute changes. We compute the impact of the system model change on the information flow graph based on evaluating the flow rules for new nodes and edges, remove information flow edges for deleted elements, as well as using the edge's dependencies on node attribute or connectivity changes.

This work is a generalization of previous approaches to dynamic information flow graphs with user-defined flow rules for virtualized infrastructures [6, 8]. We provide detailed specifications of the models, flow rules, and algorithms. We further analyze the correctness and complexity of the algorithms, in particular its termination, the correct ordering of rules using an adapted firewall rules fault model, and the equivalence of a full and differential analyses.

Contributions: In summary we make the following contributions.

- We propose the novel concept of information flow graphs constructed from user-defined flow rules. The flow rules capture trust assumptions on isolation in system components based on their attributes and connectivity. This leads to a generic and user-configurable approach that we apply to the case study of isolation in virtualized infrastructures. We analyze the correctness and complexity of our approach, in particular we adapt a firewall fault model to analyze flow rules sets.
- We establish dynamic information flow graphs that are updated based on system model changes, including incremental, decremental, node property, and resulting connectivity changes. This enables a differential information flow analysis for dynamic systems. We apply our dynamic approach also to the case study of isolation in virtualized infrastructures in combination with a system that provides system model changes.

2 Isolation in Virtualized Infrastructure

Multi-tenant virtualized infrastructures offer self-service access to a shared physical infrastructure with compute, network, and storage resources. While administrators of the provider govern the infrastructure as a whole and the tenant administrators operate in partitioned logical resource pools, both groups change the configuration and topology of the infrastructure. For example, they create new machines, modify or delete existing ones, causing large numbers of virtual machines to appear and disappear, which leads to the phenomenon of server sprawl [22]. Therefore, self-service administration, dynamic provisioning and elastic scaling lead to a great number of configuration and topology changes, which results in a complex and highly dynamic system.

Misconfigurations and insider attacks are the adverse results of such complex and dynamic systems. Indeed, even if committed unintentionally, misconfigurations are among the most prominent causes for security failures in IT infrastructure [31]. Notably, according to studies by ENISA [19] and CSA [15], operational complexity, which leads to misconfiguration and security failures, as well as isolation failures are among the top threats in virtualized infrastructures. Isolation failures put both the provider as well as the consumers at great risk due to potential loss of reputation and the breach of confidential data. Further, malicious insiders and their attacks are considered a top, very high impact security risk. Consider an example of isolation breach from misconfiguration, which we encountered in the security analysis of a financial institution's in-house VMware-based production cloud: An administrator performed a wrong VLAN ID configuration change leading to an unnoticed network isolation breach between the high-security and the test security zone. The goal of our approach is to compute an information flow analysis in such rapidly changing systems.

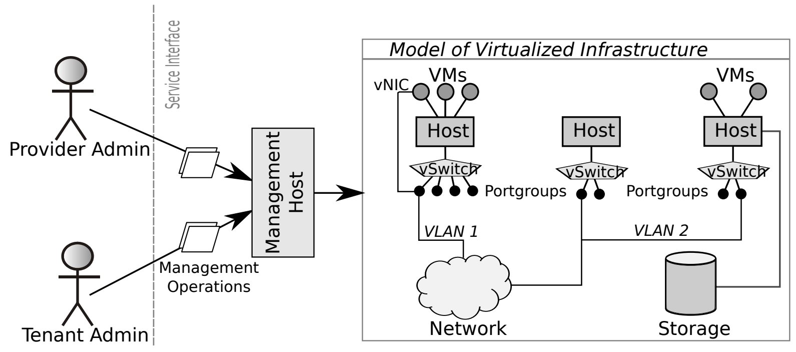


Figure 1: Model of a Virtualized Infrastructure

In Fig. 1 we illustrate our model of a virtualized infrastructure, which consists of (virtualized) computing, networking and storage resources that are configured through a well-defined management interface. In particular, we illustrate the networking part in more detail. Physical hosts and their hypervisors provide networking to VMs by virtual switches that connect the VMs to the network. A virtual switch contains virtual ports, to which the VMs are connected via a virtual network interface card (vNIC). Virtual ports are aggregated into *port groups*, which apply a common configuration to a group of virtual ports. Virtual LANs (VLANs) allow a logical separation of network traffic between VMs by assigning distinct VLAN IDs to the port groups. Our network model is focused on the OSI Layer2.

From an isolation and trust assumptions point of view, two VLANs are logically isolated from each other if they are configured with distinct VLAN ID values (and not configured to 0). However, if two port groups have the same VLAN IDs, but their underlying virtual switches are without a physical network connection, then they are also physically isolated. On the compute level and an arguable assumption is that hypervisors isolate VMs, i.e., no side channels [32] exist. It is crucial to allow user configurable/extensible rules that capture those different and arguable assumptions. The goal of our approach is to capture user-dependent trust assumptions in rules that guide our information flow analysis. The output of the analysis is tightly dependent to the conditions of the rules and these conditions may be invalidated due to system model changes, which leads to a complex analysis.

3 Constructing an Information Flow Graph using Flow Rules

In this section we lay the foundation for the fully dynamic information flow analysis by constructing an overlay information flow graph on a given system model graph using flow rules. We formalize both the system and information flow models, as well as defining the flow rules and their matching. We introduce an algorithm for the first-matching of flow rules and discuss a well-ordering for rules sets. Important for the dynamic analysis is to capture the dependencies of information flow edges on the flow rules' conditions. Additionally we need to capture implicit dependencies due to the first-matching application of rules.

3.1 System and Information Flow Models

The input of the information flow analysis is a system model in the form of a directed, symmetric, vertex-typed and -attributed graph. The analysis produces as an output a directed, edge-labeled graph, which we call an information flow graph, as an overlay on the system model graph. Figure 1 illustrates our model of a virtualized infrastructure including actors such as administrators. We represent the topology of the virtualized infrastructure with the following graph model.

Definition 1 (System Model) *Let $\mathbb{T}$ be a set of vertex types, Σ an alphanumeric alphabet where $\mathbb{A} \subset \Sigma^+$ is a set of vertex attribute names, and $\mathbb{D} \subset \Sigma^*$ is a set of attribute values. The system model graph $G_S = (V_S, E_S, P)$ contains a set of uniquely labeled and typed vertices $V_S \subset \mathbb{V} := (\Sigma^+ \times \mathbb{T})$, a set of edges $E_S \subseteq (V_S \times V_S)$, and a vertex properties set $P \subset \mathbb{P} := (\mathbb{A} \times \mathbb{D})$. A vertex v is a tuple of vertex label and type $(l, t) \in V_S$, and we write $v.t$ to obtain the type of a vertex. The edges are directed and symmetric, i.e., for each edge $e = (v_i, v_j)$ there must exists an edge $e' = (v_j, v_i)$ in E_S. A partial function*

$attr : (\mathbb{V} \times \mathbb{A}) \nrightarrow \mathbb{D}$ is defined as an attribute function which returns for a given vertex and attribute name the attribute value. We also use the notation $v.a$ for $a \in \mathbb{A}$ to obtain the attribute value instead of $attr(v, a)$.

Definition 2 (Hierarchically-Typed and Relational Vertex Model) *The vertex types $\mathbb{T}$ form a tree hierarchy that establishes a partial ordering of the types based on transitive parent-child relations, i.e., child < parent or a directed edge (child, parent), where the root node type is called* Any. *We define a type relation $T \subset (\mathbb{T} \times \mathbb{T})$. A given type pair $(t_i, t_j) \in T$ is considered* adjacent *if there exists a pair $(t'_i, t'_j) \in T$ for which $t_i \leq t'_i$ and $t_j \leq t'_j$. G_S is considered* valid *if $\forall(v_i, v_j) \in E_S : adjacent(v_i.t, v_j.t)$.*

The data model of the system is a simplified form of the *Enhanced Entity-Relationship Model* that establishes sub-typing and relationship modeling.

Definition 3 (Information Flow Model) *The information flow model graph $G_I(G_S) = (V_S, E_I)$ is derived from G_S and contains the set of typed and attributed vertices V_S of the system model graph G_S, as well as a set of directed and labeled edges $E_I \subseteq (V_S \times V_S)$ with an edge label function $f : E \rightarrow \{flow, noflow\}$. An edge $e = (v_i, v_j)$ with label* flow *means that information from v_i can flow to v_j, whereas* noflow *indicates no flow. We write information flow edges in short form as* iedge.

In terms of dynamic behavior of the models, we consider the system model graph to be static. In section 4 we will study a fully dynamic system model graph and its implications on our information flow graph. However the information flow model is dynamic, i.e., during application of rules we are inserting new edges.

3.2 Information Flow Rules

The information flow rules encode trust and isolation assumptions of system model elements by the user. They are a mandatory input for constructing the information flow graph from the system model graph. The application of rules in a first-matching semantic and the construction of the information flow model is discussed in subsection 3.4.

Definition 4 (Information Flow Rule) *Let F be a set of flow types $\{flow, noflow\}$, $\mathbb{T}$ a set of system model vertex types. A rule r is a tuple $r = (ft, t_i, t_j, p_a, p_c)$, where $ft \in F$, $t_i, t_j \in \mathbb{T}$, p_a a predicate on attributes of vertices V_S and p_c a predicate on connectivity of vertices in G_I. The rule describes information flow from a vertex of type t_i to another vertex of type t_j.*

A rule is considered simple *if (t_i, t_j) is* adjacent *(cf. Definition 1) and p_c is always true. A rule is considered* complex *if (t_i, t_j) is non-adjacent and p_c may only be using* connected *statements on simple flow edges. A* default *simple rule only operates on type* Any, *is adjacent, and p_a and p_c are always true. An information flow edge e that is later produced by a simple or complex rule will have the rule type $rt(e)$ of either* simple *or* complex.

We use rules with connectivity conditions for expressing tunneled information flow between two system components that are not directly connected in the system model. For example in our case study, we use connectivity conditions to model VLANs and other form of tunnels (GRE, VPN) between the tunnel endpoints.

Depending on the flow type of the default rule, the analysis may either tend to produce false positives in case of a default flow because we are over-approximating the possible information flow. In case of a default noflow, the analysis may produce false negatives.

3.2.1 Attribute and Connectivity Conditions

Our definition of an information flow rules includes two predicates p_a for attribute conditions and p_c for connectivity conditions. We treat those predicates separately and do not allow mixing attribute with connectivity conditions. The predicates are expressed in Boolean algebra.

The *attribute* predicate p_a takes two vertices v_i, v_j and the property set P of the system model graph. The predicate can use equality expressions on, and only on, the attributes of v_i and v_j. We do not allow nested attribute conditions.

The *connectivity* predicate p_c takes two vertices v_i, v_j and the information flow graph G_I. The connectivity conditions is built upon a connected predicate that we define as the following: connected(a,b) for $a, b \in V_S$ returns true if there exists a path from a to b in the information flow sub-graph $G_{I,flow} = (V_S, E_{I,flow})$ where $E_{I,flow} = \{e \mid e \in E_I \wedge f(e) = \text{flow}\}$. Only *complex* rules are allowed to have connectivity conditions and only on the information flow sub-graph that was produced by *simple* rules, i.e., on the following edge set: $E_{I,flow,simple} = \{e \mid e \in E_{I,flow} \wedge rt(e) = \text{simple}\}$. As the flow edges are directed, connected is not necessarily symmetric. The vertex parameters of connected can either be v_i and v_j, or adjacent vertices of those.

We call a condition predicate *closed* if it has been partially applied with the two vertices v_i, v_j. The resulting closure still takes either the current attribute property set P for attribute conditions or the current G_I for connectivity conditions.

3.2.2 Rule Matching and Evaluation

Given an information flow rule and a pair of vertices, we define when a rule matches and what the evaluation of that rule returns.

Definition 5 (Rule Matching and Evaluation) *Given a rule $r = (ft, t_i, t_j, p_a, p_c)$ and a pair of vertices v_i and v_j. The current system state is given as $G_S = (V_S, E_S, P)$ and $G_I(G_S)$. A rule has a* full *match if the (sub-)types match: $(v_i.t \leq t_i) \wedge (v_j.t \leq t_j)$, and the conjunction of conditions is true: $p_a(v_i, v_j, P) \wedge p_c(v_i, v_j, G_I)$. The rule returns an information flow edge $e = (v_i, v_j)$ with flow label $f(e) = ft$. If the types do not match, then the rule evaluates to* nil. *If the types match, but any of the predicates does not, then we have a* partial *match, and we return an* implicit dependency*, which contains the rule, the closed attribute and connectivity condition predicates, as well as the vertex pair.*

Here we only introduced the matching of a single rule and the possible in return values. In subsection 3.4 we discuss a first-matching algorithm that takes a set of well-ordered rules for evaluation.

3.2.3 Attribute and Connectivity Dependencies

If a rule fully matches and returns an information flow edge, this edge depends on the rule's attribute and connectivity condition. To prepare the grounds for the dynamic system model analysis, we associate these dependencies with the edges.

An *attribute* dependency AttrDep is a set of tuples (v, a), where $v \in V_S$, a is an attribute of vertex v, and a predicate d_a, which is the closed attribute predicate that is true if the attribute dependency is still fulfilled. Each usage of a vertex attribute in the attribute condition will result in a vertex-attribute tuple in the resulting attribute dependency. Similarly, a *connectivity* dependency ConnDep is a set of tuples (v_i, v_j, p) where v_i and v_j are the connectivity endpoints and p an optional connectivity path. Predicate d_c indicates if the connectivity dependency is still fulfilled, which is again the closed connectivity predicate.

3.3 Ordering of Information Flow Rules

The ordering of rules is important since we apply them in a first-matching semantic in our analysis. In this section we discuss how to establish a partial ordering for a given sequence of rules based on the rules' types and conditions. We derive a directed acyclic graph, the *Rule Order Graph*, from the partial ordering, which yields a rule evaluation order for the analysis.

For a sequence of rules R we define a function $\text{cmp} : (R \times R) \rightarrow \{\text{EQ}, \text{LT}, \perp\}$ that establishes a partial ordering for any pair of rules with the return values less-than (LT), equal (EQ), and undefined ($\perp$).

We use a running example to illustrate the rule ordering. We defined a subset of rules in Table 1, which are derived from our case study description from section 2. We establish the ordering using two implementations of the *cmp* function: one for type-based and another for condition ordering. If type-based ordering returns equality for a given rule pair, we need to further order by conditions.

3.3.1 Type Ordering

Given our type hierarchy from Definition 2, two rules may operate on different levels of this hierarchy. In general, given two rules that operate on types that are in a transitive parent-child relationship, then the

Table 1: Subset of Information Flow Rules Relevant for PortGroup (PG) VLAN Isolation.

#	Kind	Flow	Directed Node Pair	Condition(s)	Edge Dependency
1	Simple	noflow	VSwitch → PortGroup	$PG.vlanId \neq 0$	Attribute VLAN ID
2	Simple	noflow	PortGroup → VSwitch	$PG.vlanId \neq 0$	Attribute VLAN ID
3	Simple	flow	Any → Any	—	—
4	Complex	flow	PortGroup → PortGroup	$PG_i.vlanId \neq 0 \wedge PG_i.vlanId = PG_j.vlanId$ $\wedge\ connected(vswitch(PG_i), vswitch(PG_j))$	Attribute VLAN ID Connectivity of VSwitches
5	Complex	noflow	PortGroup → PortGroup	—	—

rule with the child types has to be evaluated first. Otherwise, the more general parent-type rule is always applied. We define the *cmp* function for type-based ordering as the following:

$$\text{cmp}_{\text{type}}(r_1, r_2) = \begin{cases} \text{EQ} & \text{if} \quad r_1.t_i = r_2.t_i \wedge r_1.t_j = r_2.t_j \\ \text{LT} & \text{if} \quad (r_1.t_i < r_2.t_i \wedge r_1.t_j \leq r_2.t_j) \vee (r_1.t_i \leq r_2.t_i \wedge r_1.t_j < r_2.t_j) \\ \text{ERR} & \text{if} \quad (r_1.t_i < r_2.t_i \wedge r_1.t_j > r_2.t_j) \vee (r_1.t_i > r_2.t_i \wedge r_1.t_j < r_2.t_j) \\ \perp & \text{otherwise} \end{cases}$$

The types of the type tree form a partial order and here we establish a product order for tuples of two types. LT if one type is strictly less than the other in the tuple. We have an error case (ERR) if we have conflicting relations, where one type in the tuple is strictly less but the other one is strictly greater than the corresponding type of the other tuple. In any other case the ordering is undefined.

In case of EQ, we require further condition-based ordering. In case of ERR we need to abort the information flow analysis as the rules ordering is inconsistent. If any of the pair types are not in a (transitive) parent-child relation, then the ordering is undefined, i.e., we obtain a partial order of the rules based on their types. If for all rules the node type pairs are distinct/non-relational, then the rules are confluent, i.e., the order of which they are evaluated does not matter.

3.3.2 Condition Ordering

For the rules that are defined for equally typed nodes we require ordering based on their condition predicates. Basically when given two equally typed rules, the rule with the most specific condition has to be first, and the one with the most general condition last. The two rules must not be equal, but can be either in a LT or $\perp$ relation.

Given two rules r_1 and r_2 with their condition predicates $p_1 = r_1.p_a \wedge r_1.p_c$ and $p_2 = r_2.p_a \wedge r_2.p_c$. The predicate atoms are connected statements and attribute equalities on typed constants and variables. We need to define a partial order (EQ, LT, $\perp$) returned by a function cmp_{cond} analog to the type-based ordering. We employ an approach based on the interpretation of predicates and truth assignments. We leverage existing work in this areas, e.g., for the ordering of methods with conditions [20] or predicate interpretation with parametrized truth assignments and value domains [1].

A function truths is defined that returns for two given predicates a list of variable assignments from the value domains as well as connected statement truth assignments. Further, we define a eval function that takes a truth assignment (variable value assignments and connected truth assignments) and a predicate, and returns either true or false for the predicate evaluation under the given variable assignments and connected statement assignments. The function substitutes variables with assigned values and connected statements with truth assignments. The predicate with substitutions is then evaluated.

Connectivity Truth Assignments: We have a set C of connected(a, b) statements each for a unique pair of vertices (a, b). Each unique statement is considered true or false leading to $2^{|C|}$ possible truth assignment combinations. Note that the statement is not symmetric, i.e., when connected(a, b) is assumed true it does not mean that connected(b, a) must be true too.

Attribute Equality Truth Assignments: We write attribute equalities as a predicate AttrEq(a1,a2) for $a1 = a2$. We have a set of AttrEq(a1, a2) where the parameters can either be constants or variables of types integer, string, or Boolean. The *domain* of integers D_i includes the integer constants and for each variable a unique random integer value. Analog are the domains D_s for strings and D_b for Boolean values. We consider the set of attribute variables typed integer A_i, string A_s, or Boolean A_b. The possible combinations of value assignments for the attribute variables are $|D_i|^{|A_i|} \cdot |D_s|^{|A_s|} \cdot |D_b|^{|A_b|}$.

Determining a Partial Ordering of Predicates: The combinations of truth assignments for connected statements and value assignments for attribute variables leads to the overall number of combinations: $2^{|C|} \cdot |D_i|^{|A_i|} \cdot |D_s|^{|A_s|} \cdot |D_b|^{|A_b|}$. It is exponential to the number of connected statements as well as attribute equality parameters.

Given two rules r_1 and r_2 and their predicates $p_1 = r_1.p_a \wedge r_1.p_c$ and $p_2 = r_2.p_a \wedge r_2.p_c$. We compute the truth and variable assignments with truths on those predicates and then iterate over the assignments. For each assignment eval evaluates both predicates. We obtain equality if for all assignments the interpretations of the predicates are simultaneously true. For an LT order we require that for all assignments the first predicate implies the second. Otherwise, the order of the predicates is undefined.

We define the condition-based compare function as the following:

$$\text{cmp}_{\text{cond}}(r_1, r_2) = \text{cmp}'_{\text{cond}}(r_1.p_a \wedge r_1.p_c, r_2.p_a \wedge r_2.p_c)$$

where $\text{cmp}'_{\text{cond}}$ is a helper function that operates on the rules' predicates.

$$\text{cmp}'_{\text{cond}}(p_1, p_2) = \begin{cases} \text{EQ} & \text{if} \quad \forall t \in \mathsf{truths}(p_1, p_2) : \mathsf{eval}(t, p_1) \wedge \mathsf{eval}(t, p_2) \\ \text{LT} & \text{if} \quad \forall t \in \mathsf{truths}(p_1, p_2) : \mathsf{eval}(t, p_1) \rightarrow \mathsf{eval}(t, p_2) \\ \perp & \text{otherwise} \end{cases}$$

The two predicates are equal if they are simultaneously true for all truth assignments. In negated form, they are "mutually exclusive exactly if one implies the negation of the other" [20] or in other words a NAND operation. They are LT if the specific predicate p_1 always implies the more generic predicate p_2, i.e., when p_1 is true then p_2 must be true, which is a logical implication. Ernst [20] uses the term "overrides" to describe that one predicate is a specialization of another, i.e., the specific predicate overrides the general predicate. He defines it as "Method m1 overrides method m2 iff m1's predicate implies that of m2, that is, if (not m1) or m2 is true." [20].

3.3.3 Establish a Rule Order Graph

We establish a *Rule Order Graph* G_R as a directed graph with rule identifiers as vertex labels. A directed edge (r_1, r_2) with edge head r_1 and edge tail r_2 represents that $r_1 < r_2$. It means that r_1 must be evaluated before r_2. The relation between partial orders and DAG (also later topological sorting) is well known and we use it here.

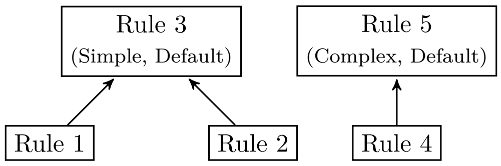


Figure 2: Order Graph of Rules of Table 1 based on Type and Condition Ordering.

Our example rule set produces the following rule order graph illustrated in Figure 2. The simple rules Rule 1 and Rule 2 are dependent on the default simple rule Rule 3. Rule 1 and Rule 2 are however unrelated in terms of the type pair they operate on, because $VS \not\leq PG$ and vice versa. Rule 4 is a complex and non-adjacent rule, therefore independent of all the other simple rules, but dependent on the default complex rule Rule 5.

3.4 Application of Rules and Construction of Information Flow Model

The construction of the information flow model is based on the system model and uses the matching of individual rules (cf. subsubsection 3.2.2) from a well-ordered set of rules (cf. subsection 3.3). Hence, the application of rules requires as inputs the system model graph as well as the rule order graph. The output is an information flow model with the edge dependencies derived from the rule application.

For our case-study we use both the system model sub-graph as illustrated in Figure 3 and the subset of rules shown in Table 1. The final output is shown as an overlay graph in Figure 4.

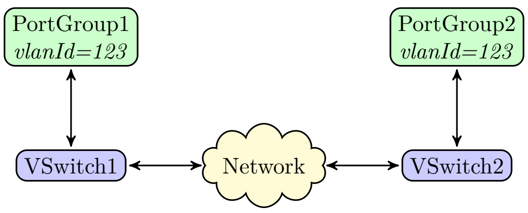


Figure 3: Input Model: Subset of system model graph

The algorithm uses a first-matching semantics of the rules. A topological sort of the rule order graph provides a valid evaluation order that adheres to the ordering of the rules. If the topological sort cannot produce a sorting we will report an error. The topological sort may produce many valid evaluation orders, because two rules with undefined ordering are confluent and can be evaluated in any order.

The application of the rules is defined in Algorithm 1. The key points of the algorithms are:

- *TopoSort* performs a topological sorting and produces a linear ordering R of the rules in the rule order graph G_R. We split the rule sequence R into simple (adjacent) rule set R_{simple} and a complex (non-adjacent) rule set R_{complex}. For the simple rules we iterate over the edge set of the system model graph. For the complex rules we obtain the nodes for the matching (sub-)types using a function *TypedNodes* and evaluate the node pairs.

- We have a function *EvalRule* that tries to match a rule (cf. subsubsection 3.2.2) and returns either an information flow edge (*iedge*), implicit dependency, or *nil* for a given rule, node pair, as well as system and information flow models and optional component graph. A pre-condition of the rule evaluation is that $(u, v) \notin E_I$.

- For a returned information flow edge, we find, obtain, and remove the implicit dependencies of previous rules using the *ImplicitDeps* function (cf. subsubsection 3.4.1). We associate the implicit dependencies with the iedge, i.e., setting *iedge.implicits*, and insert the edge into the information flow model graph.

- If an implicit dependency is returned, we store the dependency together with the rule and for complex rules also together with the evaluated node pair. In the algorithm D_S holds the implicit dependencies of simple rules, and D_C holds the ones of complex rules indexed by the evaluated node pairs.

- In case of *nil* we simply evaluate the next rule. We report an error if we have unclaimed implicit dependencies, which have not been taken by another rule. For example no default rule for adjacent rules, or no catching rule for the non-adjacent rules.

Strongly Connected Components: As an optimization to determine reachability, we use strongly connected components (SCC) and a component graph (or also called reachability tree) [14, Section 22.5]. They allow us to efficiently evaluate connected statements in a rule's connectivity condition. A component graph is a DAG that contains the SCCs as vertices and there exists a directed edge between two SCCs if there exists a directed edge between any two elements that are contained in the respective SCCs. Elements within one SCC are mutually reachable, i.e., reachability can be checked by set membership. To determine reachability between two elements that are not part of the same SCC, we try to find a path in the component graph between their respective SCCs. The reachability is by definition only unidirectional. With the function *ComputeSCC* we compute the SCCs and the component graph on a sub-graph of G_I that only contains flow-labeled edges and is further parametrized by the rule type. During rule application we compute the SCCs [37] after all simple rules have been evaluated, because the subsequent complex rules may have connectivity conditions based on the result of the simple rules. Finally we also compute SCCs after the complex rules have been evaluated for the entire information flow graph, in order for policy checks to verify connectivity.

Algorithm 1: Application of Information Flow Rules.

```
Data: Rule Order Graph $G_R$, System Model Graph $G_S = (V_S, E_S, P)$
Result: Information Flow Graph $G_I = (V_I, E_I)$
$G_I \leftarrow (V_S, \emptyset)$
$R \leftarrow$ TopoSort($G_R$)
$D_S \leftarrow \emptyset$ // Initializing implicit dependencies of simple rules
// Processing simple rules of $R$
foreach $(u, v) \in E_S$ do
    foreach $r \in R_{simple}$ do
        $res \leftarrow$ EvalRule($r, u, v, G_S, G_I$)
        if $res$ is iedge then
            ($D_S$, d) $\leftarrow$ ImplicitDeps(res, r, $D_S$)
            iedge.implicits $\leftarrow$ d
            $E_I \leftarrow E_I \cup \{res\}$
            break
        else if $res$ is implicit dependency then
            $D_S \leftarrow D_S \cup (r, res)$
    if $D_S \neq \emptyset$ then
        ERROR // Unclaimed implicit dependencies
$G_C \leftarrow$ ComputeSCC($G_I$)
$D_C \leftarrow \emptyset$ // Initializing implicit dependencies of complex rules
// Processing complex rules of $R$.
foreach $r \in R_{complex}$ do
    $V_{r,i} \leftarrow$ TypedNodes($r.t_i, V_S$)
    $V_{r,j} \leftarrow$ TypedNodes($r.t_j, V_S$)
    foreach $(u, v) \in V_{r,i} \times V_{r,j}$ do
        $res \leftarrow$ EvalRule($r, u, v, G_S, G_I, G_C$)
        if $res$ is iedge then
            (D, d) $\leftarrow$ ImplicitDeps(res, r, $D_C[(u, v)]$)
            $D_C[(u, v)] \leftarrow D$
            iedge.implicits $\leftarrow$ d
            $E_I \leftarrow E_I \cup \{res\}$
            break
        else if $res$ is implicit dependency then
            $D_C[(u, v)] \leftarrow D_C[(u, v)] \cup (r, res)$
    if $D_C \neq \emptyset$ then
        ERROR // Unclaimed implicit dependencies
$G_C \leftarrow$ ComputeSCC($G_I$)
```

3.4.1 Implicit Dependencies

To further prepare the ground for the dynamic information flow analysis, we need to record when a rule matched because a child rule (in the rule order graph) did not match due to mismatching attribute or connectivity conditions. A resulting information flow edge from a rule evaluation obtains the negative conditions from the rule's child rules. It means the result only exists because one of the child rules did not match. Once the system or information flow model are changing a previous rule may match and the current result needs to be invalidated.

The function *ImplicitDeps* takes an iedge, a rule, and a list of implicit dependencies. It returns a new dependency list with the matching ones removed and a disjunction of the matching dependencies' predicates. The matching dependencies for a rule r are: $D' = \{(r', d_a, d_c) \in D \mid r' < r\}$ with $r' < r$ for a transitive order in the rule order graph (cf. subsubsection 3.3.3). The remaining implicit dependencies are simply $D \setminus D'$. In the case of complex rules, we need to further index the implicit dependencies by the vertex tuple. The implicit dependencies D' are then associated with the iedge. An iedge is valid if it's own dependencies are true, but not any implicits. Given $D' = (i_1, \ldots, i_k)$, then $iedge.implicits(P, G_I) = \bigvee(i_j.d_a(P) \wedge i_j.d_c(G_))$. Both the closed attribute and connectivity predicates

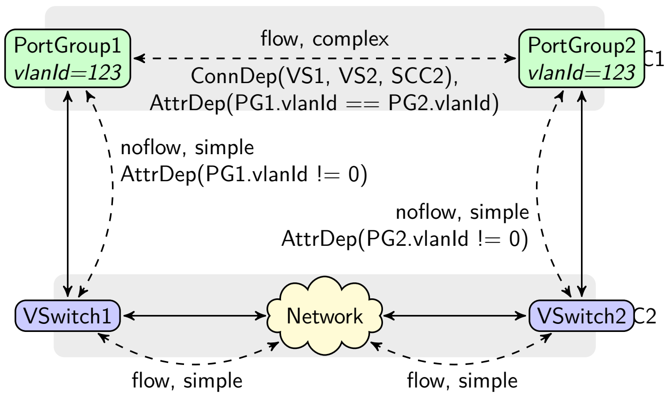


Figure 4: Output Model: Graph model annotated with dashed information flow edges of different kinds (simple, complex, attribute-dependent, connectivity-dependent).

have to be true for a child rule to match. If any of the child rules match then the parent rule must be invalidated, therefore the disjunction of implicit dependency predicates. The validity of an information flow edge under the current property set and information flow graph is given as a predicate $valid(iedge, P, G_I) = iedge.d_a(P) \land iedge.d_c(G_I) \land \neg(iedge.implicits(P, G_I))$ where d_a and d_c are again the closed condition predicates. We derive the attribute and connectivity dependencies not only for the rule's own conditions, but also for its implicits.

3.5 Algorithm Analysis

We analyze our approach with regard to the termination and complexity analysis of the algorithm. We are adapting and extending a firewall fault model and analyze how our analysis prevents such faults. Finally we discuss the correctness of the ordering and application of rules.

3.5.1 Algorithm Termination and Complexity

The termination of Algorithm 1 is given by the following properties:

- *Finite Sets:* We apply a finite set of flow rules. We evaluate the simple rules by iterating over the finite edge set E_S and similarly for the complex rules on subsets of vertex products $V_S \times V_S$. In each iteration of Algorithm 1 the set of *non-evaluated* edges or vertex pairs shrinks, and we do not modify these sets during the execution of the algorithm.
- *Limited Inter-Rule Dependencies:* We do not have any circular or self dependencies among rules, which could otherwise result in a rule application without termination. Rules depend on node types and their attributes, which are not influenced by any other rules. Complex rules may depend on connectivity, which is influenced by other rules. However, we limit connectivity conditions to only iedges produced by simple rules (cf. Definition 4). This provides a one-way dependency from complex to simple rules, but no circular or self dependencies exists among the complex rules.
- *Termination of Helper Functions:* The helper function *EvalRule* performs a rule matching and only uses a terminating BFS on the finite component graph. *ImplicitDeps* is iterating over the finite set of implicit dependencies. *TypedNodes* returns a subset of nodes from the finite nodes set V_S. *TopoSort* and *ComputeSCC* are existing algorithms and are variants of DFS, which is terminating for finite graphs with node coloring.

We analyze the run-time complexity of our analysis by breaking it up into the following steps:

- *Rules Ordering and Topological Sort:* Type-based rule ordering requires $|R|^2$ comparisons, each comparison's complexity linear to the depth of the type hierarchy. Condition-based ordering requires $|R|^2 \cdot 2^{|C|} \cdot |D_i|^{|A_i|} \cdot |D_s|^{|A_s|} \cdot |D_b|^{|A_b|}$ comparisons, which is exponential to the number of connected statements and attribute equalities. Topological sort is $O(|V_R| + |E_R|)$, since it is based on DFS. The ordering and topological sort is done once for a given rule set.

- *Rule Evaluation: EvalRule* is constant with regard to G_S and R, but for a given rule r it depends on the number of condition statements in p_a (the sets of integer/string/boolean attribute atoms: $|A_i| + |A_s| + |A_b|$) and p_c (the set of connected statements $|C|$). Connected statements can be evaluated in constant time using set membership checks when the nodes in the same SCC, or linear to the size of the component graph by using path finding (e.g., BFS). Attribute condition atoms are evaluated in constant time.

- *Implicit Dependency:* For simple rules the implicit dependencies are derived linearly to the size of the rules. For complex rules, we first perform a constant lookup with the node pair (u, v) followed by finding the implicit dependencies in linear time in the rule set.

- *Simple Rules Application:* The evaluation of the simple rules requires $|E_S| \cdot |R_{\text{simple}}|$ applications of *EvalRule* and *ImplicitDeps*.

- *Complex Rules Application:* The evaluation of the complex rules requires $|R_{\text{complex}}| \cdot |V_S'|^2$ applications of *EvalRule* and *ImplicitDeps*. We evaluate the complex rules for the matching pairs of typed nodes, where V_S' is a subset of V_S, which practically makes a difference but not asymptotically.

- *Computation of Strongly Connected Components:* We compute the strongly connected components (SCCs) twice: first after the application of simple rules and a second time after the complex rules application. Tarjan's algorithm [37] to compute SCCs has a complexity of $O(|V_S| + |E_I|)$ as well as the component graph creation is linear [14, Section 22.5].

In summary, we can differentiate between load-time and run-time complexity. Load-time is concerned about rule ordering and run-time about the evaluation of rules. The dominating parts for load-time are the quadratic rule complexity for rule ordering in general and exponential condition ordering in particular. In practice this is not a problem, because we only do it once for a given rule set, the condition ordering only happens for equally node-typed rules, and the number of connectivity and equality statements in the predicates is small. During run-time the dominating factor is the evaluation of complex rules with quadratic vertex set size.

In terms of space complexity, the upper bound is given by a full mesh information flow graph given by $|V_S|^2$ iedges. Each iedge may obtain and store implicit dependencies from $|R| - 1$ rules. The component graph based on the SCCs would as the upper bound contain $|V_S|$ SCCs, where each system model vertex is its own SCC. As a future optimizations, we can introduce information flow *vertices* in addition to the iedges. Such a vertex would allow to move from a full mesh information flow graph to a star topology. For instance in the case of Rule 5 of Table 1, which currently would create a full mesh between the non-matching portgroups, we can create one information flow vertex for the noflow portgroups.

Complexity Comparison with Traversal Analysis: We now compare the complexity of the information flow graph based analysis, denoted as FlowGraph approach, to the graph traversing analysis of [7], which we denote as the Trav approach. The Trav information flow analysis performs a graph traversal based on a set of traversal rules, which are similar to our flow rules, starting from a set of information source vertices. Each source obtains a unique *color* that propagates through the graph based on the rules' decisions. In the worst case, we have $|V_S|$ information sources and perform a DFS-like traversal with $O(|V_S| + |E_S|)$ from each source, leading to a dominating run-time complexity of $|V_S|^2$. In terms of space complexity, each vertex has to store at least $|V_S|$ unique colors, although rules may create an arbitrary number of "sub-colors", also called tags. We compare the two approaches for the following steps:

- *Rules Ordering:* This step is required by both approaches in similar complexity, although in Trav a good ordering is assumed and no automated ordering performed. The rules conditions of Trav do not contain connected statements, which simplifies the condition ordering to just attribute and tags-based conditions.

- *Rule Evaluation:* Considering a single rule evaluation, both approaches operate on the vertices attributes. FlowGraph also allows connectivity conditions that is evaluated linear to the size of the component graph in the case of a complex rule. The Trav approach operates on the current color and sub-colors, which is independent of the graph size.

- *Rule-Dependent Metadata:* Rules can lead to metadata that needs to be stored in addition to the information flow state. In FlowGraph we store explicit and implicit dependencies. Similarly, the Trav rules can create sub-colors or tags. In both cases the space complexity of the metadata is highly dependent on the specific rules set.
- *Rules Application:* In FlowGraph we differentiate between simple and complex rules. We do not have this differentiation in the Trav approach, but only have simple rules equivalents. For complex rules, the dominating run-time complexity is given by $|V_S|^2$, although in practice we evaluate a vertex subset, e.g., only portgroups. In the Trav approach we start for instance from VMs as information sources, and typically $|PG| \ll |VM|$.
- *Connectivity Evaluation:* In FlowGraph we use SCCs for efficient connectivity evaluation, which takes in the worst case linear to the component graph size. In Trav we evaluate connectivity based on colors, which is constant for a color set membership. However the connectivity depends also on the information sources, i.e., we can evaluate connectivity only from specific information source vertices.

In summary, the dominating run-time complexity of $|V_S|^2$ is given in this approach by the complex rules evaluation, although in practice we typically evaluate a subset of V_S. In the Trav approach the same complexity is dominating when starting a graph traversal from each vertex as an information source. The evaluation and application of rules differs slightly between the two approaches, where this one depends on connectivity conditions and the other one on color tags. In terms of space complexity, both approaches have to store in the worst case $|V_S|^2$ information flow states, either in the form of a full mesh information flow graph or as $|V_s|$ colors for each vertex. We outline as part of future optimizations how this space complexity can be reduced by moving from a full mesh to a star topology with *information flow vertices.* Although for the full system mode analysis the both approaches are very similar in terms of run-time and space complexity, in section 4 we compare the two approaches with regard to analyzing a dynamic system model.

3.5.2 Fault Model for First-Matching Rules Application

The goal of the information flow analysis is to extrapolate the isolation decisions between system model components by a user to the entire system. Instead of deciding for each individual edge and node pair in the system model graph if there is an information flow or not, the user captures generalized decisions using the flow rules.

The extrapolation is based on the specific rule set and the application of those rules. The first depends on the decisions by the user and there is no clear right or wrong. We can reduce the correctness of the extrapolation to the correctness of the rules. The second depends on the dependencies between simple and complex rules and the ordering of the rules due to our first-matching semantic. The rule application first evaluates the simple rules and only then the complex rules to satisfy their one-way dependency.

To analyze the rule ordering, we adapt and extend the firewall fault model [12, 13], because in firewall rules we also deal with first-matching semantics and similar faults.

- **Wrong Order:** The ordering of rules is critical in a first-matching application and our approach has to ensure a well-ordered set of rules. We have to consider the following cases how rules can be in a wrong order:
  - Wrong Type Ordering: A rule with more generic types must appear after a rule with more specific types, i.e., sub-typed rules before super-typed rules. We rely on the facts that the type tree establishes a partial order of the types and product order to establish a partial order for the tuple of types.
  - Wrong Conditions Ordering: For two equally typed rules, the rule with the more generic condition must appear after the rule with the more specific condition. We obtain a partial predicate order using truth assignments and interpretation with variable assignments.
  - Inter-Rule Dependencies: Rules may only depend on each other due to connectivity conditions. However this could lead to circular dependencies. We prevent cycles by only allowing one-way dependency from complex to simple rules. We always evaluate simple rules first so that the dependency of the complex rules is satisfied.

  - Conflicting Rules: When rules operate on the same types and conditions but producing different results. We prevent this by requiring a non-equal ordering after type and condition ordering for two given rules.

- **Missing Rule:** Depending on the default rule, a missing rule may lead to false positives (default is flow) or false negatives (noflow). We can reduce this fault to the correctness of the rules and their coverage (number of explicit vs. default rule) [7].
- **Wrong Predicates:** Wrong conditions may result in false positives or negatives when a rule triggers under the wrong circumstances or is not triggered at all. This fault is again reduced to the correctness of the rules.
- **Wrong Decision:** A rule may return a wrong flow decision (Flow/NoFlow). This can be a crucial mistake that can also lead to false positives and negatives. In general we advice to perform NoFlow decisions in the simple rules with a default flow decision, in order to mitigate false negatives. In such a case a wrong decision can be spotted more easily.
- **Wrong Extra Rule:** Old rules may remain in the rule set. They could result in ordering problems, which we would detect. However they could also result in false positives or negatives. We do not see this as a major problem as we are dealing with more static and smaller rule sets compared to network firewall configurations.

In summary, many faults can be reduced to the correctness of the rules themselves. In practice, for the different application domains we envision a rule set that is based on best practices. In addition, the ordering of rules is crucial and our approach ensures a well-ordered rules set and a rule application that satisfies inter-rule dependencies.

3.5.3 Correctness of Rule Ordering and Application

We have to show the correctness of two parts of the analysis for the static system model case: First, the correct ordering of rules; Second, the correct application of the ordered rules.

The ordering of rules relies on the following parts of the analysis, which are build upon existing work:

- *Type Ordering:* The vertex types form a type hierarchy in form of an in-tree, i.e., a rooted tree where all vertices have a unique path to the root. We can derive a partial ordering based on the child-parent edges. A product order establishes a partial ordering for a tuple of partially ordered elements.
- *Condition Ordering:* Using truth assignments and variable assignments from a value domain, we establish a partial ordering of the rule's predicate. In particular two predicates are equal when they are simultaneously true for all assignments, and one predicate is less than another if the first implies the second. Predicate ordering has been used also in other domains [20].
- *Rule Order Graph:* We construct a DAG based on the partial ordering between rules using the type and condition partial ordering. A DAG can represent a partial order, where a directed edge (u, v) represents $u \leq v$. In particular the following properties are fulfilled: *reflexive* since each vertex can reach itself ; *transitive* since we can construct a transitive closure; *asymmetric* because with $u \leq v$ and $v \leq u$ if $u \neq v$ then we would have a cycle, so not a DAG anymore.
- *Topological Sorting:* Given a DAG, the topological sorting produces a linear ordering (out of potentially many valid ones) of the vertices based on their directed edges. This is a well-known algorithm [14, Section 22.4] used in areas such as task scheduling. In our application, we apply topological sorting on our Rule Order Graph, which is a DAG, to obtain a rule evaluation order.

For the second part we show that an error in the reachability of any two vertices in the information flow model can only be caused by an error in the individual information flow rules, but not by an error in the application of the rules by our algorithm.

- *Completeness of rules application:* For a given rule set, the application of these rules is complete. We evaluate all edges with the simple rules in first-matching semantics. We evaluate all vertex pairs that match or are sub-types of a complex rules.

- *Reduction to the correctness of individual rules:* Given any pair of vertices (a, b) where *connected*(a, b) is true. There must exists a path $p = [e_1, \dots, e_k]$ of ordered k iedges with flow type flow, by definition of the connected predicate. An edge $e_i = (u, v)$ in the path has either been created by a simple rule if there exists an edge $(u, v) \in E_S$ or by a complex rule in the non-adjacent case. Either rule has made a flow decision. If the expected outcome was that *connected*(a, b) is false, then a simple or complex rule returned the wrong flow decision: instead of flow it should have return noflow.

  Similarly, if *connected*(a, b) is false, given all possible paths P between a and b, then all paths must contain a noflow iedge: $\forall(p \in P)\exists(e \in p) : f(e) = \mathsf{noflow}$. If the expectation was that *connected*(a, b) is true, then there must exists one path for which all iedges are flow. At least one rule made a wrong flow decision by returning noflow.

3.6 Summary

We lay the foundation for the dynamic information flow analysis by introducing a rule-based construction of an information flow graph for a static system model. We defined both the system and information flow models as graphs, introduced the information flow rules and their ordering, and shown an algorithm for the application of such rules. Overall the key concepts of our approach are the following:

- *System and Information Flow Models as Graphs:* The system is modeled as an directed, symmetric, vertex typed and attributed graph. The vertex types form a tree-like type hierarchy and relationships between vertices are modeled. The information flow model is an overlay on the system model (i.e., using the same vertex set) but edge-labeled with flow and noflow as well as directed.
- *Flow Rules and Matching:* The rules capture isolation and trust assumptions of the user into system model components. Based on a vertex pair types as well as attribute and connectivity conditions, the rule returns either a flow or noflow decision. A rule matches a given pair of vertices of the system model when the types are equal or subtypes of the rule, and when the conditions are true. Important for the dynamic analysis is that we record for each iedge the attribute and connectivity dependencies, as well as implicit dependencies due to the first-matching rule application.
- *Rule Ordering and Application:* The rules are applied in a first-matching way. Therefore the rules ordering is crucial. We establish a partial ordering based on rules' types and – if equally typed– also on conditions. On the resulting *Rule Order Graph* we perform a topological sort which yields an evaluation order. We always evaluate first the simple rules then the complex ones due to possible connectivity dependency.

4 Fully Dynamic Information Flow Analysis

We lay the foundation for the dynamic information flow analysis in the previous section, in particular by recording for the created information flow edges the condition dependencies as well as implicit dependencies from preceding rules that did not match. If connectivity or attributes change, the affected information flow edges with their dependencies have to be re-validated and if necessary partially re-computed.

In this section we discuss the handling of a fully dynamic system model and the implications on the information flow model. Instead of performing an information flow analysis always from scratch when the system model graph changes, we perform an analysis that updates the information flow graph. First we define a change to the system model as a graph delta.

Definition 6 (System Model Change) *Given a system model graph $G'_S = (V'_S, E'_S)$. We define a system model change as a graph delta $\Delta = (V^+, V^-, E^+, E^-, M)$, where $V^+ \subset \mathbb{V}$, $V^- \subseteq V'_S$, $E^+ \subset \mathbb{E}$, $E^- \subseteq E'_S$, $M \subset (V'_S \times \mathbb{A} \times \mathbb{D})$. The delta contains creator as well as eraser nodes and edges, and a set of node attribute modifiers (vertex, attribute, value).*

We rely on an existing system [8] that provides us such system model changes for our case study.

Given a delta Δ and two versions of the system model graph: before the change $G'_S = (V'_S, E'_S)$ and after the change $G_S = (V_S, E_S)$. G_S is constructed from the given Δ and G'_S in the following way: $V_S = (V'_S \setminus V^-) \cup V^+$, $E_S = (E'_S \setminus E^-) \cup E^+$, and applying the node modifiers on V'_S. A *differential* information flow analysis computes an information flow graph G_I based on an information flow graph

G'_I of the previous version of the system model graph G'_S and Δ. Thereby it operates on the *difference* between the system models given as Δ to compute the updated information flow model graph G_I.

The challenge we solve is to maintain an information flow graph, which is build from simple as well as attribute and/or connectivity-dependent information flow edges, even when connectivity or attributes are changed. Our differential analysis works in two phases: First, given a graph delta, we process the changes to the system model graph and the impact on the information flow graph. In particular applying flow rules for new vertices and edges, removing affected iedges while removing vertices and edges, and determining attribute dependency violations due to attribute changes. Second, based on the changes to information flow model in the first phase, we compute and process connectivity changes, in particular determining connectivity dependency violations.

4.1 Translating System Model Changes to Information Flow Changes

In the first phase we process the system model graph delta, and for each element of the graph delta $\Delta = (V^+, V^-, E^+, E^-, M)$ compute information flow graph changes.

- **Node Attribute Changes:** Find all affected attribute-dependent edges and remove them if they are invalid, i.e., their attribute condition does not hold anymore or one of the implicit dependencies is true. Re-evaluate the vertex pairs of the removed iedges and insert potentially new iedges due to re-evaluation.
- **Eraser Edges:** For each system model edge we remove the corresponding information flow edge. Further, if one of the edge's vertices is part of a connectivity-dependent iedge with another vertex as a connectivity endpoint, then the iedge is invalid if the removed edge provides the relation between the vertex and endpoint.
- **Eraser Nodes:** Remove the nodes as well as all their incoming and outgoing edges from the information flow graph. Find all connectivity-dependent edges that require the erased nodes as connectivity endpoints, and remove those edges too.
- **Creator Nodes:** For each node evaluate the complex rules (given the new node, and all the existing matching typed nodes, as well as vice versa), which may create new information flow edges, and insert the created edges into the information flow graph.
- **Creator Edges**: Evaluate simple rules for each new edge and insert the resulting information flow edges in the graph. Analog to edge removal, we need to find the (negative) connectivity-dependent iedges, where the new edge establishes the relation between the vertex and its connectivity endpoint.

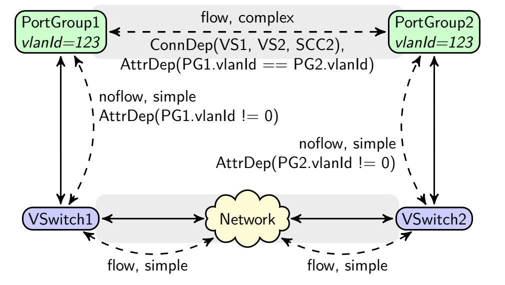
Regarding the ordering of the graph delta processing, deletion and modification of vertices can only be performed on the existing vertices of the system model graph as defined in Definition 6. We first perform the node attribute changes, then the deletion of edges and vertices. The final step is the creation of nodes and edges.

Examples of system model changes for our case study are the following and illustrated in Figure 5. In case of an attribute change where PortGroup1's VLAN ID changes to zero (denoted as $\Delta_{\mathrm{V}LAN}$), the edges between the vswitch as well as the other port group are removed, but a new flow edge is introduced between the vswitch. If we have a node removal, i.e., VSwitch is removed from Figure 4 (denoted as $\Delta_{\mathrm{V}Switch}$), the edges to PortGroup1 and Network are removed. Additionally, the edge between the port groups is removed, because it is dependent on the connectivity of the vswitches.

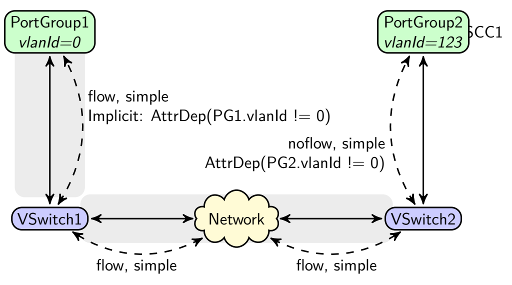
In summary, removal of graph elements directly impacts the associated iedges, but also the connectivity-dependent iedges that rely on a removed element as part of their connectivity endpoints. Attribute changes affect the attribute-dependent iedges and require a rule re-evaluation when an iedge's attribute dependencies are violated. For new graph elements we simply evaluate them with our rule set.

4.2 Processing Connectivity Changes

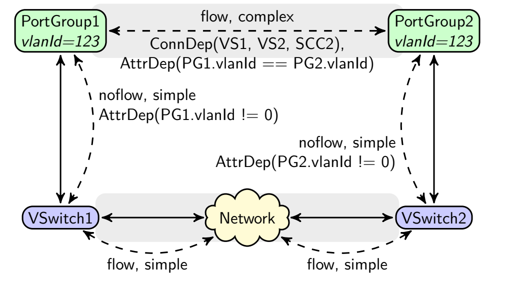
In the previous phase of our dynamic information flow analysis we processed the system model changes and modified the information flow graph accordingly. The addition and removal of information flow edges may cause changes in the overall connectivity which we have to process and handle as well. In particular



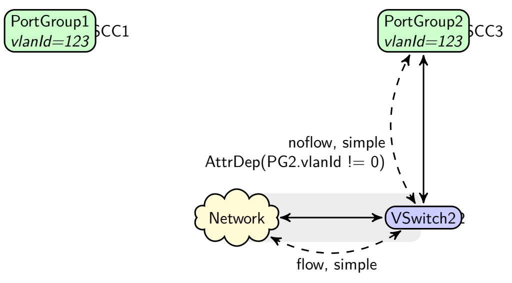
(a) System and Information Flow Models before Δ_{VLAN}.



(b) Models after Changing PortGroup1's VLAN ID.



(c) System and Information Flow Models before $\Delta_{VSwitch}$.



(d) Models after Removing VSwitch1.

Figure 5: Examples of Graph Deltas on the System and Information Flow Models.

we need to handle connectivity-dependent iedges. Only the iedges that have been created by complex rules can be affected, as only them have connectivity dependencies.

We need an interface that notifies us about connectivity changes, in particular if there exists increased or reduced connectivity, and if any existing connectivity paths in dependencies are affected. Since we are using strongly connected components (SCCs) for efficient connectivity checks, we also use SCC re-computations to tell us after inserting a set of edges, which SCCs have been added/removed, and which edges have been added/removed in the component graph.

With Tarjan's algorithm [37], we can do set operations between an old and a new component graph's vertex and edge sets. Ideally, we would use a dynamic reachability approach [33] that can also provides us notifications on reachability changes. The SCC component graph $G'_C = (V'_C, E'_C)$ is derived from G'_I, and a new $G_C = (V_C, E_C)$ from G_I. We compute the removed SCCs $V'_C \setminus V_C$ and removed inter-SCC edges $E'_C \setminus E_C$. As well as new SCCs $V_C \setminus V'_C$ and new inter-SCC edges $E_C \setminus E'_C$. We process the connectivity changes as reflected in the component graph changes in the following ways.

- **Removed SCCs or inter-SCC edges:** In the case of *reduced* connectivity, we find all iedges that have a connectivity dependency *with* a connectivity path. These iedges depend on a found path and the reduced connectivity may have invalidated this path. In particular we have to check the iedges with a path that contains a removed SCC or inter-SCC edge if their connectivity dependencies are still valid.

- **New SCCs or inter-SCC edges:** In the case of *increased* connectivity, we find all iedges that have a connectivity dependency *without* a connectivity path. In contrast to the previous case, these iedges exists because no path was found for their connected statements. The increase in connectivity may have established such a missing path. We have to check the connectivity dependency of all iedges without a connectivity path.

In our case study, if the Network node is removed from the example of Figure 4, SCC2 splits into two new SCCs, i.e., SCC2 is removed and two new SCCs are added. In the example, the connectivity-dependent edge between the port groups is affected and removed, because SCC2 appears in its connectivity path. Since another connectivity path could exists for the connectivity endpoints, we re-evaluate the complex rules for the removed edges' node pairs. However, in the example no other connectivity path exists.

4.3 Algorithm Analysis

In this section we discuss the termination and complexity of the dynamic analysis algorithm. Further we show the equivalence between the fully and differential information flow analyses.

4.3.1 Algorithm Termination and Complexity

The termination of the dynamic information flow analysis is given by the same set of properties as for the static analysis (cf. subsubsection 3.5.1):

- *Finite Sets:* The differential analysis operates on the finite sets of the graph delta with created/erased vertices and edges, as well as attribute modifiers. The iteration over these sets processes each element only once and does not modify the sets. In particular our definition of graph deltas only allow deletion and modifications of existing nodes and not of newly created ones in the same delta.
- *Limited Inter-Rule Dependencies:* Attribute and connectivity changes may result in the invalidation and re-evaluation of iedges. The termination of the algorithm is given because of the one-way connectivity dependency of complex rules towards simple rules, therefore we do not have circular connectivity dependencies. Attributes are only changed through system model changes and not through the application of rules, therefore no attribute dependencies among rules.

We analyze the complexity of our differential algorithm with regard to the processing of system model changes and the handling of connectivity changes. The complexity of processing the system model changes is the following:

- *Attribute Changes:* Processing of attribute changes is linear to the number of modifiers $|M|$. For each modifier (v, a, d) we perform a constant lookup of the affected attribute dependencies for (v, a).
- *Eraser Edges:* The removal of iedges is linear to the number of removed system model edges $|E^-|$.
- *Eraser Nodes:* For each removed node in $|V^-|$ we remove the incoming and outgoing iedges, which is linear to the number of iedges with a constant lookup of connectivity-dependent iedges indexed by endpoints.
- *Creator Nodes:* Given the node set V_S' of the previous system model, we need to evaluate the complex rules for the pairs between the existing nodes and the new nodes: $V^+ \times V_S' + V_S' \times V^+$, as well as between the new nodes themselves $V^+ \times V^+$. For each we perform the rule application (*EvalRule* and *ImplicitDeps*) and an edge insertion.
- *Creator Edges:* We evaluate the simple rules for all new edges: $|E^+| \cdot |R_{simple}|$, for each edge we evaluate the rule and find the implicit dependencies. We further find the connectivity-dependent iedges where the edge establishes the relation between a node and endpoint.

The complexity of the connectivity change processing is the following:

- *Reduced Connectivity:* We are iterating over the connectivity-dependent iedges that have a connectivity path (subset or equal of E_I). An iedge is affected if an element of its connectivity path is removed. For an affected edge we try to find an alternative path (linear to the size of the component graph, BFS for shortest path).
- *Increased Connectivity:* We are iterating over the connectivity-dependent iedges that have no connectivity path (subset or equal of E_I). For all iedges we need to check if a path has been established between the connectivity endpoints (linear to the size of the component graph, BFS for shortest path).

Complexity Comparison with Traversal Analysis: The analysis approach of [7] is not designed to handle a dynamic system model. In particular rules dependent on the current color or color tag lead to an information flow state that highly depends on the current system model. Changes to the system model requires to re-run the entire analysis. Therefore the comparison boils down to the differential complexity as previously discussed and the full analysis complexity of [7] as discussed in subsubsection 3.5.1.

4.3.2 Full and Differential Analyses Equivalence

The objective is that there is no difference in the information flow graphs produced by the differential analysis compared to the full one.

Given the current system model graph G_S, a system model change Δ, and the information flow graph G'_I of the previous system model graph G'_S. The full information flow analysis (cf. section 3) of G_S produces $G_{I,\text{full}}$. The differential information flow analysis using G'_I and Δ produces an information flow graph $G_{I,\text{diff}}$. Both $G_{I,\text{full}}$ and $G_{I,\text{diff}}$ are equal, i.e., the edge sets are equal and all edges have the same flow type. We show the equivalence between the full analysis on the changed system model and the differential analysis based on the graph delta in the following cases:

- **Node Attribute Changes:** The full analysis would never have seen the original vertex attributes, only the changed attribute value. The regular rules application of attribute-conditioned rules may either match in a first matching semantic, or no match.

  For the differential analysis we have to consider two cases: would the same rule that was applied still hold, i.e., is the attribute condition fulfilled, and would a previous rule match instead. To achieve the same results as the full analysis, the differential analysis needs to handle the two cases: the rule does not match anymore (attribute condition not fulfilled), that means the iedge has to be removed. Second, a previous rule now matches (first matching semantic), therefore the iedge produced by the current rule has to be removed. We achieve this through attribute dependencies and implicit dependencies.

- **Eraser Nodes and Edges:** The full analysis would never create iedges to or from any erased node (complex rules) nor based on any removed edges (simple rules). The differential analysis achieves the same result by removing the iedges that connect to any removed node and the iedges corresponding to the removed edges.

  In addition, the removal of edges can also break the relation between nodes and their connectivity endpoints. The differential analysis removes the complex iedges where the relation is broken due to edge removal.

- **Creator Nodes and Edges:** The changed system model is given as $V_S = (V'_S \setminus V^-) \cup V^+$ and $E_S = (E'_S \setminus E^-) \cup E^+$. We already showed the equivalence for the eraser nodes and edges, therefore we now consider $V_S = V''_S \cup V^+$ and $E_S = E''_S \cup E^+$ with the erasers already applied in V''_S and E''_S.

  Given the new edge set $E_S = E''_S \cup E^+$, for the full analysis we can split the rule application (cf. Algorithm 1) into iterating over E''_S and iterating over E^+. The differential analysis already iterated over E''_S for the previous model and now only iterates over E^+.

  Similarly for the new vertex set $V_S = V''_S \cup V^+$. The full analysis will evaluate the complex rules on typed node pairs of V_S. We can split the evaluation into the set of typed node pairs of $V''_S \times V''_S$, as well as evaluating the sets $V^+ \times V''_S$, $V''_S \times V^+$, and $V^+ \times V^+$. The differential analysis already evaluated the node pairs of $V''_S \times V''_S$ for the previous model, and now only considers the pairs between the new vertices and the existing ones.

  Additionally, the creation of new edges can establish the relation between vertices and their connectivity endpoints in case of (negative) connectivity-dependent iedges. We handle this case in the creator edge processing and re-evaluate the affected iedges.

4.4 Summary

Building upon the concepts of the static information flow analysis of section 3 we defined a fully dynamic analysis. The key concepts are:

- *System Model Changes as Graph Deltas:* Changes to the system model, which is a graph model, are defined as graph deltas consisting of creator nodes and edges, node modifiers, and eraser nodes and edges.

- *Processing of System Model Changes:* Given a system model and a system model change as a graph delta, the differential analysis computes the information flow model changes based on the graph delta. We show that the full and differential analyses result in the same information flow model.

- *Processing Information Flow Changes:* Changes to the system model result in changes in the information flow model. The differential analysis processes how the connectivity changes and determines the connectivity-dependent iedges that are affected.

Overall the differential analysis builds upon the full analysis and partially applies the rule evaluation. The attribute and connectivity dependencies of iedges are essential in producing an equal information flow result for the differential analysis compared to the full one.

5 Related Work

A variety of algorithms, including connectivity, on dynamic graphs have been long studied [16]. Of our particular interest are fully dynamic graph reachability algorithms, such as proposed by Roditty and Zwick [33], which would replace our current SCC re-computations using Tarjan's algorithm. Our approach builds up on dynamic graph reachability, where our approach computes the graph delta that is consumed to update the graph reachability. Recently, differential computation frameworks, such as *differential dataflow* [29], have been applied to differential graph computations [28], in particular to graph connectivity. As future work we will evaluate and try to implement our algorithm as a differential dataflow computation. CellIQ [27] analyzes cellular network topologies using differential graph computations. They compute connected components over a sliding window based on the GraphX [25] framework.

A similar model of information flow graphs has been used to model and analyze virtual machine system policies [34]. Two static rules translate an access control statement into a graph edge based on the statement's permission and a classification of permissions as read-like or write-like [26]. The system does not handle dynamic access control policies and neither dynamic information flow graphs. Similarly, taint tracking is guided by a set of static rules for the particular language semantic, such as TaintDroid's propagation logic [18]. Attacker propagation has been studied in socio-technical systems. Groove [23] is a graph transformation environment that has been used to model attacker actions in social-technical environments. The underlying Portunes [17] model establishes a set of semantics for the possible attacker actions. Given the variety of socio-technical environments, user-defined trust assumptions and attacker propagation rules are required. This allows to model the effectiveness of different physical security measures, e.g., door strengths, and digital security properties, such as, software vulnerabilities.

In previous work we studied the information flow analysis of *static* virtualized infrastructures with user-defined trust assumptions [7]. The analysis uses a graph traversal approach with "color" propagation starting from a set of information source vertices. However, the approach's stateful graph traversal renders this approach difficult to adapt to graph changes. Instead, in this work we pursue the use of information flow graphs rather than graph traversals. This work is a generalization of first dynamic analyses using such information flow graphs [6, 8] and provides detailed algorithm specifications and analyses.

In this work we pursue a *static* information flow analyses of a dynamic virtualized infrastructure topology. This means we analyze for potential information flows rather than actual flow. This is analog to static program analysis where the program's source code is studied rather than the executed program. On the other hand, *dynamic* approaches for information flow control (IFC) have been proposed. For instance, SilverLine [30] offers data and network isolation based on data labeling, however requires changes to both Xen and the guest VM kernel. CloudFlow [2] is based on VM introspection to monitor and extract the tasks with their SELinux security labels that running inside a VM. They implement a Chinese wall policy to prevent, for instance, that a VM with unlabeled tasks is running on the same physical server as a VM that runs a top secret task. IFC also finds application in infrastructure management and administration. For instance *H-one* [21] is a system that uses information flow tracking to establish an audit log of VM configuration tampering by administrators.

6 Conclusions and Future Work

In this work we propose a approach for the static information flow analysis for dynamic systems. We introduce the concept of dynamic information flow graphs with user-defined flow rules. The flow rules capture the user's trust assumptions in system components and their isolation. The dynamic information flow graphs are dependent on the flow rules' conditions and changes to the system model may require re-computation of parts of the information flow graph. However compared to other approaches our analysis operates in a differential way, i.e., the analysis is updated based on the changes rather than performing an entire analysis. We apply our approach to the case study of isolation in virtualized infrastructures, where we model the infrastructure's configuration and topology as a system model graph and capture assumptions in the network isolation as flow rules. An existing system provides us with system model changes that lead to updates in our dynamic information flow graph. Security systems can build upon our information flow graph to verify isolation between system components using graph reachability in dynamic systems.

As part of future work and further optimization of our approach, we propose the following directions. We aim for a graph reduction by replacing potential full-mesh graph structures, e.g., as created by default complex rules, with star topologies. For this we need to introduce the concept *information flow nodes* that can be created by flow rules. In our case study, the default complex rule for portgroups would create a noflow information flow node to connect the portgroups to. The information flow nodes are also dependent on flow rules conditions and need to be adapted based on system model changes. Furthermore, we can optimize our approach by using a dynamic reachability or SCC algorithm rather than re-computing the SCCs using Tarjan's algorithm. Finally, we aim to apply our approach to new case studies, such as for attacker propagation in digital-physical environments.

Acknowledgments

This work is partially supported by the EU H2020 projects SUPERCLOUD (grant No. 643964) and PrismaCloud (grant No. 644962), and Swiss Secretariat for Education, Research and Innovation (contract No. 15.0025).

References

[1] Aho, A. V., and Ullman, J. D. *Foundations of Computer Science.* Computer Science Press, W. H. Freeman and Company, 1995.

[2] Baig, M. B., Fitzsimons, C., Balasubramanian, S., Sion, R., and Porter, D. E. Cloudflow: Cloud-wide policy enforcement using fast vm introspection. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering* (Washington, DC, USA, 2014), IC2E '14, IEEE Computer Society, pp. 159–164.

[3] Bell, E. D., and La Padula, J. L. Secure computer system: Unified exposition and multics interpretation, 1976.

[4] Berger, S., Cáceres, R., Pendarakis, D., Sailer, R., Valdez, E., Perez, R., Schildhauer, W., and Srinivasan, D. Tvdc: managing security in the trusted virtual datacenter. *SIGOPS Oper. Syst. Rev. 42* (January 2008), 40–47.

[5] Biba, K. J. Integrity considerations for secure computer systems. Tech. rep., MITRE Corp., 04 1977.

[6] Bleikertz, S., Gross, T., Mödersheim, S., and Vogel, C. Proactive Security Analysis of Changes in Virtualized Infrastructures. In *Annual Computer Security Applications Conference (ACSAC 2015)* (Dec 2015), ACM.

[7] Bleikertz, S., Gross, T., Schunter, M., and Eriksson, K. Automated Information Flow Analysis of Virtualized Infrastructures. In *16th European Symposium on Research in Computer Security (ESORICS'11)* (Sep 2011), Springer.

[8] Bleikertz, S., Gross, T., and Vogel, C. Cloud Radar: Near Real-Time Detection of Security Failures in Dynamic Virtualized Infrastructures. In *Annual Computer Security Applications Conference (ACSAC 2014)* (Dec 2014), ACM.

[9] Brewer, D., and Nash, M. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on* (May 1989), pp. 206–214.

[10] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network & Distributed System Security Symposium (NDSS'12)* (Feb 2012).

[11] Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., and Shastry, B. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2011), SPSM '11, ACM, pp. 51–62.

[12] Chen, F., Liu, A. X., Hwang, J., and Xie, T. First Step Towards Automatic Correction of Firewall Policy Faults. In *Proceedings of the 24th International Conference on Large Installation System Administration* (Berkeley, CA, USA, 2010), LISA'10, USENIX Association, pp. 1–8.

[13] Chen, F., Liu, A. X., Hwang, J., and Xie, T. First Step Towards Automatic Correction of Firewall Policy Faults. *ACM Trans. Auton. Adapt. Syst. 7*, 2 (July 2012), 27:1–27:24.

[14] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[15] CSA. Top threats to cloud computing v1.0. Tech. rep., Cloud Security Alliance (CSA), mar 2010.

[16] Demetrescu, C., Finocchi, I., and Italiano, G. *Handbook on Data Structures and Applications.* Dinesh Mehta and Sartaj Sahni (eds.), CRC Press Series, in Computer and Information Science, 2005, ch. 36: Dynamic Graphs.

[17] Dimkov, T., Pieters, W., and Hartel, P. Portunes: Representing Attack Scenarios Spanning through the Physical, Digital and Social Domain. In *Proceedings of the 2010 joint conference on Automated reasoning for security protocol analysis and issues in the theory of security* (Berlin, Heidelberg, 2010), ARSPA-WITS'10, Springer-Verlag, pp. 112–129.

[18] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–6.

[19] ENISA. Cloud computing: Benefits, risks and recommendations for information security. Tech. rep., European Network and Information Security Agency (ENISA), nov 2009.

[20] Ernst, M., Kaplan, C., and Chambers, C. Predicate Dispatching: A Unified Theory of Dispatch. In *Proceedings of the 12th European Conference on Object-Oriented Programming* (London, UK, UK, 1998), ECCOP '98, Springer-Verlag, pp. 186–211.

[21] Ganjali, A., and Lie, D. Auditing cloud management using information flow tracking. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing* (New York, NY, USA, 2012), STC '12, ACM, pp. 79–84.

[22] 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.

[23] Ghamarian, A. H., de Mol, M., Rensink, A., Zambon, E., and Zimakova, M. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)* (March 2011).

[24] Goguen, J. A., and Meseguer, J. Security policies and security models. In *IEEE Symposium on Security and Privacy* (1982), IEEE, pp. 11–20.

[25] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 599–613.

[26] Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. Verifying Information Flow Goals in Security-enhanced Linux. *J. Comput. Secur. 13*, 1 (Jan. 2005), 115–134.

[27] Iyer, A., Li, L. E., and Stoica, I. CellIQ : Real-Time Cellular Network Analytics at Scale. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, May 2015), USENIX Association, pp. 309–322.

[28] McSherry, F. Differential graph computation. `http://www.frankmcsherry.org/differential/dataflow/2015/05/12/bfs.html`, May 2015.

[29] McSherry, F., Murray, D. G., Isaacs, R., and Isard, M. Differential dataflow. In *Proceedings of CIDR 2013* (January 2013).

[30] Mundada, Y., Ramachandran, A., and Feamster, N. Silverline: Data and network isolation for cloud services. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2011), HotCloud'11, USENIX Association, pp. 13–13.

[31] Oppenheimer, D., Ganapathi, A., and Patterson, D. A. 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* (Berkeley, CA, USA, 2003), USITS'03, USENIX Association.

[32] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. 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* (New York, NY, USA, 2009), ACM, pp. 199–212.

[33] Roditty, L., and Zwick, U. A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing* (2004), STOC '04, ACM, pp. 184–191.

[34] Rueda, S., Vijayakumar, H., and Jaeger, T. Analysis of Virtual Machine System Policies. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies* (New York, NY, USA, 2009), SACMAT '09, ACM, pp. 227–236.

[35] Rushby, J. Noninterference, transitivity, and channel-control security policies. Tech. rep., SRI International, Dec 1992.

[36] Sailer, R., Jaeger, T., Valdez, E., Caceres, R., Perez, R., Berger, S., Griffin, J. L., and Doorn, L. v. Building a mac-based security architecture for the xen open-source hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 276–285.

[37] Tarjan, R. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing* (1972).