Formalizing Threat Models for Virtualized Systems

Daniele Sgandurra(✉), Erisa Karafili, and Emil Lupu

Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK
{d.sgandurra,e.karafili,e.c.lupu}@imperial.ac.uk

Abstract. We propose a framework, called *FATHoM* (FormAlizing THreat Models), to define threat models for virtualized systems. For each component of a virtualized system, we specify a set of security properties that defines its control responsibility, its vulnerability and protection states. Relations are used to represent how assumptions made about a component's security state restrict the assumptions that can be made on the other components. FATHoM includes a set of rules to compute the derived security states from the assumptions and the components' relations. A further set of relations and rules is used to define how to protect the derived vulnerable components. The resulting system is then analysed, among others, for consistency of the threat model. We have developed a tool that implements FATHoM, and have validated it with use-cases adapted from the literature.

1 Introduction

Addressing security concerns in computing systems requires careful consideration of the threats, usually described through *threat models*. But for virtualized systems, attacks, solutions and threat models have evolved considerably over the years [10]. Before presenting a security solution, research papers usually describe their *assumptions* on the environment where the solution is meant to be deployed. However, threat models are given in a *descriptive* rather than a formal syntax, which is also not standardized. As a consequence, many publications rely on implicit, and different, assumptions or lack clarity in their assumptions for which there is no commonly understood semantics. For example, different terms are used to refer to a component assumed to be insecure such as "malicious", "untrusted", "in control of the attacker", when the underlying assumption is whether the component is inside or outside the trusted computing base (TCB).

We believe that using a precise model for threat modelling in virtualized systems would help understanding: (i) the meaning of each assumption, by harmonizing the terminology; (ii) whether the threat model has included all the required assumptions at all architectural levels; and (iii) whether these assumptions are consistent. Currently, the *relations* among components, and how assumptions on one component impact (e.g., restrict) assumptions that can be made on other components, are usually not considered. We propose a model,

© IFIP International Federation for Information Processing 2016
Published by Springer International Publishing Switzerland 2016. All Rights Reserved
S. Ranise and V. Swarup (Eds.): DBSec 2016, LNCS 9766, pp. 251–267, 2016.
DOI: 10.1007/978-3-319-41483-6_18

FATHoM, that allows developers and system designers to precisely state the conditions under which a component can be assumed trusted (or untrusted) given that another component is assumed trusted (or untrusted). FATHoM allows designers to define and analyse a threat model to determine whether it is consistent and complete. The framework, which can be applied with fine granularity, can also be used to describe how components are protected. Furthermore, the framework is compositional.

The main contributions of this work are:

– a precise notation to define threat models (FATHoM), which considers the components' security states, the relations among them, and the rules to compute the derived security states based on the assumptions and the relations;
– a set of relations and rules used to define how to protect vulnerable components, derived by checking the threat model definition for consistency;
– a prototype tool that implements FATHoM that can be used to define and analyse threat models. FATHoM allows system designers to check the threat model for consistency, completeness, and equality among threat models given a subset of required states.

The paper is structured as follows. In Sect. 2 we list some existing threat models and discuss their limitations. In Sect. 3 we describe FATHoM, including the assumptions, relations and the composition rules. Section 4 describes how threat models are defined in the FATHoM prototype tool, and how they can be analysed. In Sect. 5 we show some instantiations of threat models using our approach. Section 6 discusses related works, while we conclude in Sect. 7.

2 Current Threat Models in Virtualized Systems

Virtualization is a technique used to emulate in software the physical properties of a computer, which is encapsulated in a *virtual machine* (VM) managed independently from other VMs. This allows physical resources such as processor, memory, storage and I/O channels, to be shared between concurrent VMs, while preserving isolation. This enables a more efficient use of the resources, e.g. on Cloud computing datacenters, as they can be allocated on-demand. A *virtual machine monitor* (VMM) is the software component that creates, manages and monitors VMs. In virtualized systems, the assumptions described in a *threat model* form the basis from which security control is enforced. For example, if in one threat model physical access is not considered possible, attacks trying to subvert the VMM from the lower levels may not be taken into account by the solution, but may still exist. Similarly, if the threat model assumes the VMM to be with no (exploitable) bugs, whereas the OS is vulnerable, then a proposed protection solution can be deployed directly inside the VMM while considering attacks against the kernel as possible. Threat models have considerably evolved over the years, in an attempt to cope with novel attacks. However, we lack a standard approach to define and analyse the threat models, which is used across the research community [10]. Some approaches, such as STRIDE [11], allow designers to draw trust boundaries across components, and check for a set of known (classes of) vulnerabilities. However, they are not used to check if the definition of threat model is consistent and, furthermore, it is not possible to perform custom analysis, e.g. to compare two threat models for equality. Note that inconsistencies in the description of a threat model can arise as the exact meaning of each assumption is not precise, and how assumptions on a component are related to assumptions on other components is not defined. It is therefore difficult to compare two threat models because, even if they appear similar, they could mean different things to their designers. Furthermore, if some component is not included in the model, the consequences to the trustworthiness of the model, when this component exists in the real world, are not determined. Concerning the nomenclature, the most common set of assumptions encountered in the literature is expressed using a terminology that includes terms such as *trusted*, *vulnerable* (or *exploitable*), *untrusted* or *malicious*. However, the words are not used consistently, or with a well understood semantics. Our goal is to provide a framework that enables the definition of the threat models more precisely and clearly. The properties and features of the model we want to provide are: (i) it should be easy to define a threat model based on this framework; (ii) it should be clear what the assumptions are; (iii) the threat model should be consistent and complete; (iv) it should be possible to compose several components together.

3 Language for the Threat Models

The system we represent is composed of different components that have different *security properties* (also called *security states*), and different *relations* between them. We denote by $\mathcal{A}$ the nonempty set of components of the system:

$$\mathcal{A} ::= \{A, B, C, \cdots\}.$$

Components can be added or removed from the system. We denote by *Insert* the function for adding a new component to the system, and by *Remove* the one that removes a component from the system:

$$Insert(\mathcal{A}, E) := \mathcal{A} \cup \{E\} \qquad\qquad Remove(\mathcal{A}, D) := \mathcal{A} \backslash \{D\}$$

where E is a new component inserted to $\mathcal{A}$, while D is removed from $\mathcal{A}$.

We denote by Φ the set of all formulas of our system. Given the set of components, $\mathcal{A}$, the formulas of our system $\phi \in \Phi$, are defined by the grammar:

$$\phi ::= true \mid false \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid (\phi) \mid \pi \mid \neg\pi \mid \rho$$
$$\pi ::= \upsilon(A),\ \text{where}\ \upsilon \in \mathcal{V} \qquad\qquad \rho ::= \varrho(A, A),\ \text{where}\ \varrho \in \mathcal{R}$$

where A is a component of our system ($A \in \mathcal{A}$), $\mathcal{V}$ is the set of all security properties of the components, and $\mathcal{R}$ the set of all relations between components. The connectors $\wedge$, $\vee$, $\rightarrow$, (,), and $\neg$ are the standard ones. Through our language, we can define a set of *rules* which governs how the relations are used to compute the different properties of the components starting from other properties.

3.1 Security Properties of Components

The components of our system can have different *security properties*, for simplicity just *properties*, which are associated with their trustworthiness, reliability and restorability values. We divide the properties of our system in *basic*, *high-level*, *derived* and *accessory* properties, respectively divided in the following sets $\mathcal{V}_\mathcal{B}$, $\mathcal{V}_\mathcal{H}$, $\mathcal{V}_\mathcal{D}$, $\mathcal{V}_\mathcal{A}$, as represented in Table 1. These sets compose the set of properties of our system: $\mathcal{V} = \mathcal{V}_\mathcal{B} \cup \mathcal{V}_\mathcal{H} \cup \mathcal{V}_\mathcal{D} \cup \mathcal{V}_\mathcal{A}$.

Table 1. The Security Properties Sets for FATHoM

$\mathcal{V}_\mathcal{B} = \{assContr, assSafe, assProt\}$
$\mathcal{V}_\mathcal{H} = \{assTrust, assVuln, assUntrust, assMalic\}$
$\mathcal{V}_\mathcal{D} = \{derContr, derSafe, derProt, derTrust, derVuln, derUntrust, derMalic\}$
$\mathcal{V}_\mathcal{A} = \{derCanBeCompr, derCanBeExpl, derIsProt\}$

The designer has to specify only the assumptions of the basic properties, which are the most important ones. The rest of the properties are mostly syntactic sugar and can be derived from the basic properties and the relations. The high-level properties, $\mathcal{V}_\mathcal{H}$, are constructed from the basic ones. As the high-level properties are commonly used, the designer sometimes specifies these properties instead of the basic ones. We distinguish between the derived and accessory properties ($\mathcal{V}_\mathcal{D}$ and $\mathcal{V}_\mathcal{A}$) of the system components and the assumed ones that are given by a designer. For sake of simplicity we will consider the high-level, derived, and accessory properties as properties of their own.

Basic Properties. The basic properties represent the main assumptions we can make about the properties of a component. These define whether the components are assumed to be under control of the defenders or the attackers, by using the *controlled* property (*trustworthiness*), *assContr* and ¬*assContr* respectively. When a component is controlled, we consider whether it is assumed to be *exploitable* or not by attackers (*reliability*), and thus can be compromised. We use the properties *assSafe*, and $\neg assSafe$, respectively for a reliable component and an exploitable (vulnerable) one. If the component is compromised (¬*assContr*), we consider whether it can be protected or not (*restorability*), and introduce *protectable* or *unprotectable*, respectively *assProt* and ¬*assProt*.

Derived Properties. The system also includes the derived version of the assumed properties: *derContr*, $derSafe$, and *derProt*, taken from $\mathcal{V}_\mathcal{D}$. Note that the relations between these properties are the same as their assumed version. Thus, we omit the assumed/derived prefix for them in Fig. 1, which shows the security properties associated with system components. When a component is *Controlled* the relevant properties are *Safe* and *Vulnerable*, which specify whether a component A cannot be compromised ($Safe(A)$) or is vulnerable ($\neg Safe(A)$), because it has known vulnerabilities or is believed it can be maliciously exploited by an attacker. This property identifies components that need to be protected. Alternatively, if a component is *Compromised* (and thus not controlled), the relevant properties are *Protectable* and *Unprotectable*, which specify whether a compromised component A can be protected ($Prot(A)$) or not ($\neg Prot(A)$). The difference with the previous case is that a *Vulnerable* component can always be protected, since it is *Controlled*, whereas an *Unprotectable* component refers to a *Malicious* component, such as an external attacker, a malicious provider, or more generally any external component already compromised and that impacts the security properties of other components, and cannot always be reverted to the *Controlled* state.

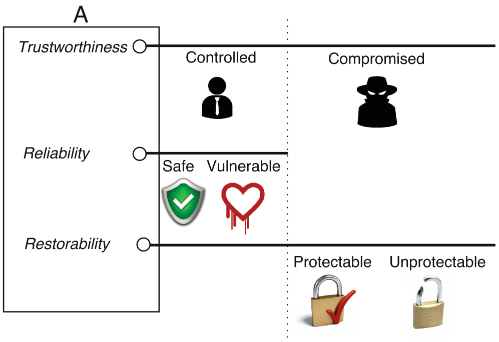


Fig. 1. Security properties

High-Level Properties. Let us introduce FATHoM's high-level properties, which can be defined from the basic ones. We define a component A *assumed trusted*, if it is assumed controllable and safe:

$$assTrust(A) := assContr(A) \wedge assSafe(A).$$

A component is *assumed vulnerable*, if it is assumed controllable and not safe:

$$assVuln(A) := assContr(A) \wedge \neg assSafe(A).$$

A component is *assumed untrusted*, if it is assumed not controllable (compromised) but is assumed protectable:

$$assUntrust(A) := \neg assContr(A) \wedge assProt(A).$$

Finally, a component is *assumed malicious*, if it is assumed not controllable and not protectable:

$$assMalic(A) := \neg assContr(A) \wedge \neg assProt(A).$$

We represent the rules between these properties and the basic ones in Table 2.

Table 2. Derivation rules between basic and high-level properties in FATHoM

$assTrust(A) \rightarrow assContr(A)$	$assTrust(A) \rightarrow assSafe(A)$
$assVuln(A) \rightarrow assContr(A)$	$assVuln(A) \rightarrow \neg assSafe(A)$
$assUntrust(A) \rightarrow \neg assContr(A)$	$assUntrust(A) \rightarrow assProt(A)$
$assMalic(A) \rightarrow \neg assContr(A)$	$assMalic(A) \rightarrow \neg assProt(A)$

In Table 3 we show the semantics of each combination, and give examples of their occurrence in virtualized systems. Recalling the terms discussed in Sect. 2, which are used frequently in related works, if we compare our nomenclature with these terms, we can see that the combinations of security properties in Table 3 correspond to (from top to bottom): Trusted ($assTrust$), a component that is assumed to be trustworthy; (ii) Vulnerable ($assVuln$), a component that can have bugs and/or can be attacked but is not in the hand of the attacker; (iii) Untrusted ($assUntrust$), assuming that an attacker has full control of this component; (iv) Malicious ($assMalic$), i.e., a component that is compromised by the attacker and cannot be recovered or protected.

3.2 Relations and Derivation Rules

In FATHoM components can have the following relations between them:

$$\mathcal{R} := \{Contr, Threat, Protect, Ign, Contain, Group, Merge\}.$$

Contr defines a binary relation, where $Contr(A, B)$ means that component A controls component B, e.g., when A is at lower virtualisation layer than B, or A is more privileged than B. Therefore, A can (potentially) change B's security properties by attacking or protecting it. This relation is already given to the system, or it can be implied by other relations. *Contr* is transitive.

Threat defines a binary relation, where $Threat(A, B)$ means that component A can threaten (attack) component B. Thus, A may be able to exploit B's vulnerabilities, e.g., when A is a remote attacker and B a reachable server from the Internet, or A is a component at a higher level (e.g., application) and B one at a lower level (e.g., OS). A successful attack must be carried out by a compromised component A against a vulnerable one B. Component A threatens an other component B because of the design of the system, or if A controls B and component A is assumed/derived compromised.

$$Contr(A, B) \wedge (\neg assContr(A) \vee \neg derContr(A)) \rightarrow Threat(A, B)$$

Some examples of these relations, in the context of virtualized systems, are shown in Fig. 2.

As we focus on threat models, we consider the *worst-case* scenario. This means that, for example, if A controls B, and A is assumed not controlled, then we cannot assume B to be controlled. Thus, we consider as not controlled all components that could be compromised within our knowledge. We can now introduce one of the accessory properties from $\mathcal{V}_{\mathcal{A}}$, which is the *can be compromised*

Table 3. Combinations of security properties for components

$assContr$	$assSafe$	$assProt$	Meaning	Examples	High-level property
✓	✓		A component that is assumed not to have been compromised and that cannot be exploited	A micro-kernel OS formally verified, or an app that is protected from integrity attacks from lower levels, or where the attacker has no access	**Trusted** ($assTrust$)
✓	✗		A component that is assumed not to have been compromised but that can be compromised in the future due to a vulnerability	An OS with bugs during the boot, where the attacker has not direct access to it but can, for example, access it remotely	**Vulnerable** ($assVuln$)
✗		✓	A component that is assumed to be compromised and that can be protected	The OS kernel in a VM, assumed to be compromised, and that is protected at run-time through the hypervisor for control-flow integrity attacks	**Untrusted** ($assUntrust$)
✗		✗	A component that is assumed to be compromised and that cannot protected	An external component (the attacker has full control of it) on which the security solution cannot do anything to change its state	**Malicious** ($assMalic$)

property, $derCanBeCompr$, that can be derived from the above relation. We say that B can be compromised, if it is controlled by a compromised component, or if B is threatened by a compromised component and B is vulnerable:

$$\begin{aligned}&((Contr(A,B) \wedge (assContr(B) \vee derContr(B)))\\ &\vee(Threat(A,B) \wedge (\neg assSafe(B) \vee \neg derSafe(B))))\\ &\wedge(\neg assContr(A) \vee \neg derContr(A) \vee derCanBeCompr(A)) \rightarrow derCanBeCompr(B).\end{aligned}$$

The derived versions of the high-level properties: $derTrust$, $derVuln$, $derUntrust$, and $derMalic$, which are part of $\mathcal{V}_{\mathcal{D}}$, are defined as follows:

$$\begin{aligned}derTrust(A) &:= \neg derCanBeCompr(A) \wedge assTrust(A)\\ derVuln(A) &:= \neg derCanBeCompr(A) \wedge assVuln(A)\\ derUntrust(A) &:= derCanBeCompr(A) \vee assUntrust(A)\\ derMalic(A) &:= assMalic(A).\end{aligned}$$

We represent in Table 4 the derivation rules between derived properties. Finally, the Ign (*Ignore*) relation, used to ignore components, forces the model to assume a given component as trusted: $Ign(A) \rightarrow assTrust(A)$.

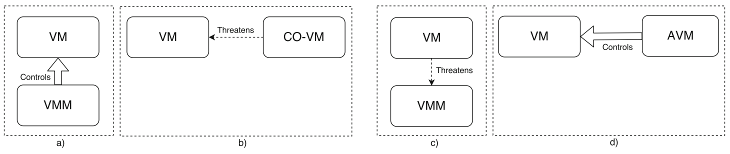


Fig. 2. Examples of *Controls* and *Threatens* relations

Table 4. Derivations rules between derived properties in FATHoM

$derTrust(A) \rightarrow derContr(A)$	$derTrust(A) \rightarrow derSafe(A)$
$derVuln(A) \rightarrow derContr(A)$	$derVuln(A) \rightarrow \neg derSafe(A)$
$derUntrust(A) \rightarrow \neg derContr(A)$	$derUntrust(A) \rightarrow derProt(A)$
$derMalic(A) \rightarrow \neg derContr(A)$	$derMalic(A) \rightarrow \neg derProt(A)$

3.3 Composability

We further introduce the possibility of composing components together, by exploiting three compositional relations: *Contains*, *Groups* and *Merges*. They are respectively used to: (i) enforce consistency among the state of a composite component and the state of its set components; (ii) simplify the description of the threat model, by grouping components together; (iii) abstract from inner components by introducing a new component that summarizes their security states and relations.

Contain is a binary relation, where *Contain*(A, B) means that A contains B, and all of A's internal components must have the same security property as A. This is useful when an external component always includes an internal one, such as a VM that includes the OS, and the security states need to be coherent among them. This relation is transitive given the assumptions in the internal components, represented in Table 5 (1)–(5). *Group* is a binary relation, where *Group*(A, B) means that A is a new (virtual) component, which groups B (together with, possibly, other components). In this case A is a new component, and *Group* is consistent in terms of *Contr* and *Threat* relations, as shown in Table 5 (6)–(12). The last composability relation is $Merge$, where $Merge(A, B)$ means that a new component A is used to consolidate all the internal components (B, and possibly other components) into a new one and merges their states and relations, as represented in Table 5 (13)–(18). The internal components are no longer considered, and assumptions and relations can be now expressed about A. This is useful when we want to consider several, similar, components in a single one with the same properties. The difference between *Groups* and *Merges* is that the *Groups* relation is only used to facilitate the definition of several rules and assumptions on several components together, but the external (virtual) component is not used by the rules, only the internal (real) ones. Instead, *Merges* is used to remove from the model similar components and replaces them by a new (real) one. In this case, the assumptions and the relations are firstly given on the internal component B, and applied to the external one. Note that a

Table 5. The composability rules in FATHoM

$Contain(A,B) \wedge Contain(B,C) \rightarrow Contain(A,C)$	(1)
$Contain(A,B) \wedge assTrust(A) \rightarrow assTrust(B)$	(2)
$Contain(A,B) \wedge assVuln(A) \rightarrow assVuln(B)$	(3)
$Contain(A,B) \wedge assUntrust(A) \rightarrow assUntrust(B)$	(4)
$Contain(A,B) \wedge assMalic(A) \rightarrow assMalic(B)$	(5)
$Group(A,B) \wedge Group(B,C) \rightarrow Group(A,C)$	(6)
$Group(B,C) \wedge Contr(A,B) \rightarrow Contr(A,C)$	(7)
$Group(A,B) \wedge Contr(A,C) \rightarrow Contr(B,C)$	(8)
$Group(B,C) \wedge Threat(A,B) \rightarrow Threat(A,C)$	(9)
$Group(A,B) \wedge Threat(A,C) \rightarrow Threat(B,C)$	(10)
$Group(A,B) \wedge Group(C,D) \wedge Contr(A,C) \rightarrow Contr(B,D)$	(11)
$Group(A,B) \wedge Group(C,D) \wedge Threat(A,C) \rightarrow Threat(B,D)$	(12)
$Merge(C,A) \wedge Contr(A,B) \rightarrow Contr(C,B)$	(13)
$Merge(C,A) \wedge Threat(A,B) \rightarrow Threat(C,B)$	(14)
$Merge(C,A) \wedge assTrust(A) \rightarrow assTrust(C)$	(15)
$Merge(C,A) \wedge assVuln(A) \rightarrow assVuln(C)$	(16)
$Merge(C,A) \wedge assUntrust(A) \rightarrow assUntrust(C)$	(17)
$Merge(C,A) \wedge assMalic(A) \rightarrow assMalic(C)$	(18)

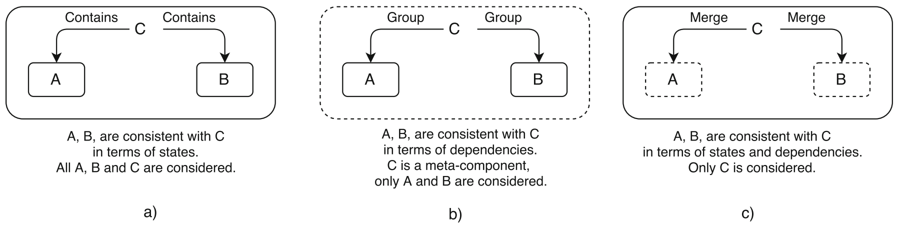

Contains
C
Contains
A
B
A, B, are consistent with C in terms of states.
All A, B and C are considered.
a)
Group
C
Group
A
B
A, B, are consistent with C in terms of dependencies.
C is a meta-component,
only A and B are considered.
b)
Merge
C
Merge
A
B
A, B, are consistent with C in terms of states and dependencies.
Only C is considered.
c)

Fig. 3. *Contains* (a), *Groups* (b) and *Merges* (c) Relations

precondition to merge components together is that they are similar, i.e., there are no conflicts. An example of these three relations is shown in Fig. 3.

3.4 Protecting Components

We introduce in FATHoM two versions of the *Protect* relation. This relation is given by the system/designer, and through it we can derive a very important security property *derIsProt*, which defines the property of a component being protected. The first version of *Protect* is a binary relation, where $Protect(A, B)$ means that A can protect component B from the threats considered in the model.

The second version is a ternary relation, where $Protect(A, C, B)$ means that component A can protect component B, with the help of a new component C. This relation is reflexive, where $Protect(A, C, A)$ means that component A can protect itself, with the help of a new component C (that enhances A). For the ternary *Protect* relation, the new component C is inserted in the set of components of A. We show these relations in Fig. 4 (for the sake of readability, we split the ternary relation in two cases, (a) and (c)).

The *Protects* relations are used to define the *derIsProt* property, which means that a component is in the state of being *protected*. There are four cases when we can derive that a component is protected, as shown in Table 6. We say that A is protected when it is derived trusted, as shown in (19). We say that A is protected when there exists a controlled component B that protects A (and B can control A), in case A is not safe or protectable, as shown in (20). We say that A is protected when there exists a controlled component B that protects A, with the help of another component C, where B controls A and A is not safe or protectable, as shown in (21). In this case, C is a new component, inserted in the set of components, that is contained in B, and is assumed controlled or is controlled by B. Finally, we say that A is protected when it can protect itself

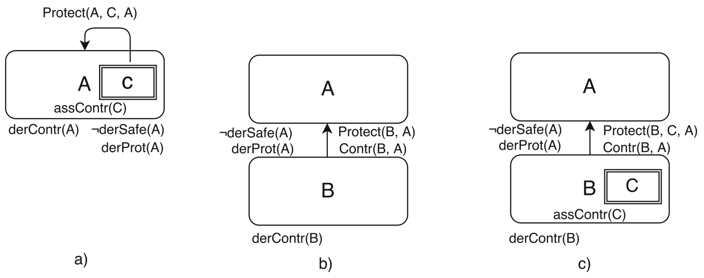


Fig. 4. Rules *Protects* relation: (a) Using a patch, (b) Using an external component, (c) Enhancing an external component with a patch

Table 6. The four cases when a component is derived protected

$$derTrust(A) \rightarrow derIsProt(A) \tag{19}$$

$$\begin{aligned} &Protect(B, A) \wedge Contr(B, A) \wedge derContr(B) \wedge \\ &\quad (\neg derSafe(A) \vee derProt(A)) \rightarrow derIsProt(A) \end{aligned} \tag{20}$$

$$\begin{aligned} &Protect(B, C, A) \wedge Contr(B, A) \wedge derContr(B) \wedge \\ &(\neg derSafe(A) \vee derProt(A)) \wedge Insert(\mathcal{A}, C) \wedge \\ &Contain(B, C) \wedge (assContr(C) \vee Contr(B, C)) \rightarrow derIsProt(A) \end{aligned} \tag{21}$$

$$\begin{aligned} &Protect(A, C, A) \wedge derContr(A) \wedge \\ &(\neg derSafe(A) \vee derProt(A)) \wedge Insert(\mathcal{A}, C) \wedge \\ &Contain(A, C) \wedge (assContr(C) \vee Contr(A, C)) \rightarrow derIsProt(A) \end{aligned} \tag{22}$$

with the help of C, where A is controllable, exploitable or protectable, as shown in (22). In this case, C is a new component, contained in A, assumed or derived controlled by A. Note that with the current rules, a component is derived *Vulnerable* only if it is assumed so. Furthermore, a component is derived *Malicious* iff it is assumed so, using the worst-case rule. However, in some situations, we need to take into account threat models where a component A is assumed to be vulnerable and either (i) it is the target of the protection solution itself or (ii) it is assumed protected by another component (e.g., a vulnerable VMM protected by a trusted module). Hence, it is assumed, and required for the threat model definition, that the component is vulnerable. To limit the cascading effect of this vulnerable component being exploited, we need to check whether it is already protected. This is captured by one of the properties in $\mathcal{V}_\mathcal{A}$:

$$derCanBeExpl(A) := \; derCanBeCompr(A) \wedge assVuln(A)$$

where a component is derived exploitable if it can be compromised and it is assumed vulnerable.

4 Using FATHoM to Define and Analyse a Threat Model

We now discuss the implementation of FATHoM, and how it can be used to define and analyse threat models.

4.1 FATHoM Prototype Tool

We have designed and developed a tool implementing FATHoM (available at http://rissgroup.org/fathom/) that includes in its knowledge base all the general rules discussed so far. The tool is used also to load a template with the components and relations. A graphical interface facilitates the description of the threat model, which is converted into FATHoM language. (An example of a template is described in Sect. 5.) After loading the template, the user can customize some components, and then specify the security states of the components. Finally, the FATHoM tool compiles the threat model (derived from the template and the user choices) and the model's rules into an executable program for XSB[1]. After converting the threat model and the rules for XSB, a designer can query the FATHoM tool to analyse the system: to display those components whose assumed state is different from the derived ones (i.e., a consistency check); to add protection components and exploit the *Protects* relations. At this point, FATHoM re-compiles the updated threat model (i.e., the derived initial model plus the new protection components and relations), and shows the derived and final threat model.

[1] http://xsb.sourceforge.net/.

4.2 Definition of the Threat Model

The following steps are used to define threat models within our model: (i) FATHoM loads the *rules* already defined for all domains, i.e., *Controls* and *Threatens*; (ii) the user defines (or imports an existing) *template* that includes the components' *ontology* for the domain of interest (e.g., virtualized system), which defines: the *components* specific to the domain and the *Controls* and *Threatens* relations on these components; (iii) the user selects those components that need to be *Merged* (if any) and those to *Ignore* (if any); (iv) the user can add new components (optional) or new relations in the model to both existing or new components (optional); finally, (v) the user sets, for each component, the assumed value for *Trusted* or *Vulnerable* or *Untrusted* or *Malicious*. In contrast to the current, verbose, definitions of threat models, which often span several pages in research papers, in FATHoM users only need to define the assumed security states for all the components, e.g., through a table, such as Table 7, or through a figure, such as Fig. 5b. Not only is the representation very succinct, but the underlying semantics is also common across all the threat models.

4.3 Derivation and Analysis

Once the model has been defined (and possibly customized), and the values of the assumptions on the security states has been chosen, FATHoM is used to derive the security states. FATHoM allows the developer to analyse the consistency and completeness of the model, and query possible combinations of assumptions to derive a desired target model. In particular, FATHoM allows the designers to check if there are some security states whose assumptions are different from the derived ones. When an inconsistency is found, the developer is asked to only update the inconsistent assumptions. FATHoM also forces the designers to define the security states for all the components. When a security state for a component is not defined, unless the user has specifically chosen to *Ignore* it, a warning is returned. Designers can analyse possible solutions to protect the vulnerable and protectable components in the derived threat model. This can be done by refining the model as follows: (i) the user introduces new protection components in the system, such as a hardware co-processor; (ii) the users specifies which components are used to *Protect* which other components, e.g. the *Vulnerable* ones. Then, FATHoM enables the user to perform the same analyses on the improved model.

5 Use-Cases Analysis: Virtualized Systems

We show an instantiation of FATHoM for virtualized systems including the template we have defined for a use-case scenario.

5.1 Ontology of Components and Relations

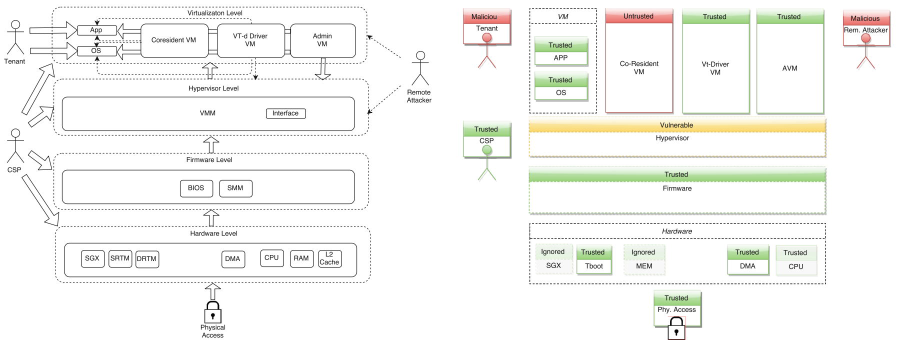
The *components* we model in this template are defined at four layers: (i) *virtualization* level: OS, application (App), administrative VM (AVM), co-resident VM (CO-VM), VT-Driver-domain (i.e., a VM that interfaces all the requests to the devices using shared drivers); (ii) *hypervisor* level: VMM, VMM interface; (iii) *firmware* level: BIOS (or UEFI), SMM; (iv) *hardware* level: DMA, Trusted Boot, which can be specialized in static-root-of-trust (SRTM), dynamic-root-of-trust (DRTM), Intel SGX [2], Memory (MEM), which can be refined into RAM and L2-Cache, CPU[2]. We augment these elements with additional ones to consider further threats. In detail, when the virtualized environment is run by an external provider, the trust placed in that provider also needs to be considered [5,7,9,14]. Hence, other components that we introduce are: (i) *physical-access*: a component that defines whether physical access is possible for the attacker; (ii) *actors*: the Cloud service provider (CSP), the tenant and a generic external attacker. Note that for some components, such as actors and physical-access, the only relevant security state is the *Controlled/Compromised* one, which is semantically equivalent to the component being trusted or not, whereas the *Protected/Vulnerable* may not need to be considered. Hence, we will restrict their security states to *Trusted* and *Malicious* only. Furthermore, the actors in the system can be further refined if we consider entities such as the manufacturer of the hardware used by the provider, the developers of the software run by the provider, and, in general, any third-party involved with the Cloud provider [3].

The template we have defined includes the following *relations*: (i) we *Group* the components at the hardware-, firmware-, hypervisor- and virtualization-level; (ii) any element at lower-levels can *Control* those at higher ones (and physical access *Controls* all the levels); (iii) the Tenant *Controls* the user VM; (iv) the remote attacker *Threatens* all the Virtualization level and VMM; (v) the CSP *Controls* all the VMs, the VMM and lower levels; (vi) the VT-d *Threatens* the VM, VMM; (vii) the AVM *Controls* the VM, VMM; (viii) the Co-resident VM *Threatens* the VM, VMM. These relations used on the previous ontology give rise to the template shown in Fig. 5a.

5.2 Analysis of the Threat Model

Defining the Threat Model and Checking for Inconsistencies. This analysis is used to identify inconsistencies in the threat model's assumptions, by showing the derived security states that are different from the assumed ones. In this use-case, we firstly customize the template depicted in Fig. 5a to adapt it to the description of this use-case. Here, we consider a threat model that assumes the VMM to be *Vulnerable*, the tenant and remote attacker to be *Malicious*, and co-resident VM to be *Untrusted*. The goal, from the point of view of the provider, is to protect the OS from attacks by tenants and remote attackers by

[2] Further components that could be considered here are virtualization extensions, chipset, hard-disk firmware.



(a) FATHoM Template for Virtualized System: Components and Relations (b) Use-Case of Assumptions on a FATHoM Threat Model

Fig. 5. FATHoM template and initial assumptions

enhancing the VMM to check the integrity of the OS. This use case is adapted from [13,15]. In particular, we consider that a trusted boot-based solution is used to protect the loading of the hypervisor. Then, the provided solution will describe how a new component inside the VMM is used to protect the OS. In detail, we use the FATHoM compositional rules on the template as follows: (i) we *Merge* RAM and L2Cache into MEM; (ii) we *Merge* SRTM and DRTM into a single Tboot component; (iii) we *Merge* BIOS and SMM into a Firmware component; (iv) MEM and CPU are *Ignored*; (v) SGX is *Ignored*. We then set the assumptions on the security states as shown in Fig. 5b (the relations are not shown, since after loading the template we only need to define the assumptions). This graphical description is then translated in the FATHoM syntax, and then compiled for XSB. By using these assumptions, and by querying the FATHoM tool for inconsistencies among assumptions and derived states, FATHoM shows that the components App, OS, AVM, Co-VM and hypervisor are inconsistent[3]. These components are those whose assumed states is different than the derived one. In this specific case, FATHoM shows that, by letting the attacker being able to threaten the hypervisor, he/she can compromise the AVM and Co-VM too, as well as the OS and App, and this is not consistent with the initial assumptions. Furthermore, the tenant can compromise the OS. Note that, since in this use-case we have assumed the VMM to be vulnerable, but protected by a Tboot solution[4], we need to update the model with the relations to protect the VMM from Tboot, and check the model again. By querying FATHoM we see that the hypervisor is consistent with the assumptions, and, hence, we only need to change the App and OS to "untrusted" to be consistent. We have now an initial

[3] For the sake of conciseness, we do not show the complete rules, ontology, assumptions, and analysis in XSB here. All the examples are available at http://rissgroup.org/fathom/, along with a technical report.

[4] We only focus on static integrity protection.

Table 7. Components and assumptions of the use-cases

Component	Trusted	Vulnerable	Untrusted	Malicious
APP			✓	
OS			✓	
Co-VM			✓	
Vt-Driver	✓			
AVM	✓			
Hypervisor		✓		
Firmware	✓			
Tboot	✓			
DMA	✓			
Physical Access	✓			
CSP	✓			
Tenant				✓
Remote Attacker				✓

consistent threat model, whose instantiation is shown in Table 7. As we can see, the formulation of the threat model is very succinct.

Protecting the Components and Derivation Analysis. On this stable threat model, we then introduce a patch in the VMM to protect the OS[5]. Then, by recompiling the threat model with the new protection component, and its relation(s), we query the FATHoM tool again to check the derived *Protected* components. In this case, FATHoM derives VMM and OS as *Protected* as well.

6 Related Work

Mulval [8] is a logic-based analyser that enables the modelling and reasoning about components' interaction. The main difference with our model is that in FATHoM we define *assumptions* on the security of components, not vulnerabilities or attack paths. Furthermore, our goal is to formalize these assumed security states using a common terminology and ontology, and check its soundness. The *CORAS* method [4] defines a language, and a set of UML-based diagrams, for threat and risk modelling. FATHoM differs from CORAS as it focuses only on threat modelling. Furthermore, FATHoM facilitates the description of the system's assumptions, and follows a worst-case approach, instead of a probabilistic one, and can analyse its soundness. The goals are also different: using CORAS it is possible to implement a risk-evaluation plan, while in our model we depict a static scenario. Similarly, *CySeMoL* [12] is a modelling language

[5] The description of the patch is outside the scope of the use-case.

targeted at enterprise architectures, to enable administrators to perform a probabilistic inference analysis. The model includes a set of meta-components to define probabilistic dependencies, attack-paths, preconditions, etc. In FATHoM we are focused on describing the initial security preconditions of the system in an easy and sound way, and to take into account generic attacks. *Nemesis* [6] is a risk-assessment framework to test Cloud systems, by collecting measurements on known vulnerabilities of the system components, modelling the threats and assessing the risk. FATHoM is a framework to define in a consistent way the threat model and used to reason on generic security solutions valid for a set of use-cases. Finally, [1] exploits model checking to verify system-level security properties of interacting VMs, and is focused in particular on distributed access control policies. Even if the context is similar, i.e., virtualized environment, the goal of FATHoM is the formalization and analysis of threat models, rather than the verification of access control policies. In summary, none of the the existing works allows designers to (i) define in a compact way threat models, and (ii) perform custom analysis, e.g. to check their consistency. Most existing tools allow users to perform predefined queries over such a threat model (i.e., to check for vulnerabilities), which is supposed to be consistent. Hence, these approaches and FATHoM are complementary.

7 Conclusion and Future Work

Current threat models lack a common terminology to describe the trusted components, and the relations between them. This gives rise to possible inconsistencies, and incompleteness, of the considered definition. This issue may also impact the assumptions on which a security solution is built upon. In this paper we have proposed FATHoM to describe and analyse threat models and have demonstrated that in real-world scenarios the usability of the FATHoM model is very simple, since it only requires to load an existing template, optionally customize it, and set the assumed values for the security states. We are considering other attacker's goals, such as confidentiality, to be defined in the relations and rules, and other scenarios (e.g., mobile). Finally, the worst-case assumptions underlying our model could be mitigated using probabilistic assumptions on the relations and rules.

Acknowledgement. Supported by FP7 EU-funded project Coco Cloud under grant no. 610853, and EPSRC Project CIPART grant no. EP/L022729/1.

References

1. Alexander, P., Pike, L., Loscocco, P., Coker, G.: Model checking distributed mandatory access control policies. ACM Trans. Inf. Syst. Secur. **18**(2), 6:1–6:25 (2015)
2. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: 2nd Workshop on Hardware and Architectural Support for Security and Privacy, HASP 2013 (2013)

3. Bleikertz, S., Mastelic, T., et al.: Defining the cloud battlefield - supporting security assessments by cloud customers. In: 2013 IEEE Cloud Engineering (IC2E), pp. 78–87, March 2013
4. Brændeland, G., Dahl, H.E.I., Engan, I., Stølen, K.: Using dependent CORAS diagrams to analyse mutual dependency. In: Lopez, J., Hämmerli, B.M. (eds.) CRITIS 2007. LNCS, vol. 5141, pp. 135–148. Springer, Heidelberg (2008)
5. Butt, S., Lagar-Cavilla, H.A., et al.: Self-service cloud computing. In: ACM Conference on Computer and Communications Security, pp. 253–264. ACM (2012)
6. Kamongi, P., Gomathisankaran, M., Kavi, K.: Nemesis: automated architecture for threat modeling and risk assessment for cloud computing. In: Academy of Science and Engineering, USA (2015)
7. Li, M., Zang, W., Bai, K., Yu, M., Liu, P.: Mycloud: supporting user-configured privacy protection in cloud computing. In: Annual Computer Security Applications Conference, ACSAC 2013, pp. 59–68. ACM (2013)
8. Ou, X., Govindavajhala, S., Appel, A.W.: MulVAL: a logic-based network security analyzer. In: 14th USENIX Security Symposium, SSYM 2005, vol. 14, p. 8 (2005)
9. Santos, N., Rodrigues, R., Gummadi, K.P., Saroiu, S.: Policy-sealed data: a new abstraction for building trusted cloud services. In: 21st USENIX Conference on Security Symposium, Security 2012, p. 10 (2012)
10. Sgandurra, D., Lupu, E.: Evolution of attacks, threat models, and solutions for virtualized systems. ACM Comput. Surv. **48**(3), 46:1–46:38 (2016)
11. Shostack, A.: Threat Modeling: Designing for Security. Wiley (2014)
12. Sommestad, T., Ekstedt, M., Holm, H.: The cyber security modeling language: a tool for assessing the vulnerability of enterprise system architectures. IEEE Syst. J. **7**(3), 363–373 (2013)
13. Srivastava, A., Raj, H., Giffin, J., England, P.: Trusted VM snapshots in untrusted cloud infrastructures. In: Balzarotti, D., Stolfo, S.J., Cova, M. (eds.) RAID 2012. LNCS, vol. 7462, pp. 1–21. Springer, Heidelberg (2012)
14. Szefer, J., Keller, E., Lee, R.B., Rexford, J.: Eliminating the hypervisor attack surface for a more secure cloud. In: 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 401–412. ACM (2011)
15. Xiong, X., Tian, D., Liu, P.: Practical protection of kernel integrity for commodity OS from untrusted extensions. In: NDSS (2011)