Bootstrapping Trust in a "Trusted" Virtualized Platform

Hagen Lauer
Monash University / CSIRO's Data61
hagen.lauer@monash.edu

Amin Sakzad
Monash University
amin.sakzad@monash.edu

Carsten Rudolph
Monash University
carsten.rudolph@monash.edu

Surya Nepal
CSIRO's Data61
surya.nepal@monash.edu

ABSTRACT

The Trusted Platform Module (TPM) can be used to establish trust in the software configuration of a computer. Virtualizing the TPM is a logical next step towards building trusted cloud environments and providing a virtual TPM to a virtual machine promises a continuation of trusted computing concepts. The association between a virtual TPM and a virtual machine is a critical concern. We show that a "trusted" virtualized platform may fall victim to a *Goldeneye* attack. In this work, we put forward a formal model for virtualization systems and trusted virtualized platforms. We pair this with a model for establishing trust in a virtualized platform following conventional reasoning over trusted computing systems. We show that if a *Goldeneye* attack is successful, it would allow a verifier to establish trust in an untrustworthy platform. We discuss attack vectors and possible solutions which would mitigate *Goldeneye*.

CCS CONCEPTS

• **Security and privacy** → **Trusted computing**; *Trust frameworks*; Virtualization and security.

KEYWORDS

Trusted computing, virtual TPM, trust model, vulnerabilities.

1 INTRODUCTION

Before entrusting a computer with a secret, a client needs some assurance that the computer can be trusted [18]. Without such trust, performing tasks involving secrets or sensitive data securely is currently not possible. Practical trust establishment technologies are crucial for the secure operation, proliferation and adoption of current and future services [12]. In fact, the need to bootstrap trust is most evident when using hosted and remote computing systems [16, 21].

For example, a cloud provider P offers to host an instance of a customer's OS of choice in a virtual machine which the customer may freely use. This allows customer C to run certain applications forming a service. Conveniently, C does not have to maintain any hardware. Customers need to trust P's virtualization system (Fig. 1) to provide important properties: isolation between their system and other customers as well as access to interfaces, keys, storage objects, and resources. If a customer wants to establish trust in her services, i.e. the guest OS and applications in Fig. 1, then an associated vTPM should contain information about the software state of that part of the system. Unless C trusts P implicitly, C will also have to establish trust in the virtualization system which hosts her workload, i.e. the machine and virtual machine monitor in Fig. 1. Lastly, cloud

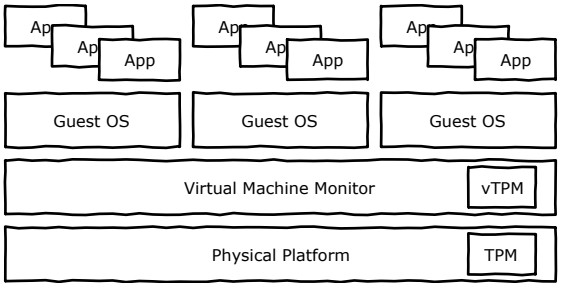

Figure 1: A sketch of a virtualization system equipped with a TPM which hosts virtualized systems with virtual TPMs. A guest OS running in a virtual machine and composed with a virtual TPM is referred to as a trusted virtualized platform.

customers C hosting their service on P's virtualization system may need to prove compliance to other parties as well. Being able to establish trust hosted services and the underlying virtualization system becomes crucial if the hosted service needs to be trusted. A provider should be able to demonstrate continuous security compliance and prove the integrity of her provided service to customers. Alternatively, customers may want an option to monitor the system hosting their services.

One way to establish trust in a computer is to use the secure hardware such as the Trusted Platform Module (TPM) to record and report its software state. Using a process called remote attestation this information can be securely conveyed to a remote party. Given the software state, the client (or any other agent) can decide whether the platform should be trusted—assuming that a trustworthy software state and configuration is known to the client. Similar forms of this approach are the basis of many practical security features of modern operating systems[1]. In short, with appropriate software support, the TPM can be used to measure and record each piece of software loaded [22] and securely convey this information to a remote party [4, 16]. Consequently, the TPM can be used to effectively establish trust in the software configuration of a machine. This can in principle be applied to a virtual machine monitor using a physical TPM. It can also be applied to its guests with the support of a virtual TPM [7, 23] as shown in Fig. 1. The virtual

[1]Microsoft operating systems use the TPM for BitLocker and protecting and limiting the use of cryptographic keys https://docs.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview. Similarly, Linux systems offer secure boot features and integrity subsystems which also use the TPM as hardware anchor.

machine associated with a virtual TPM form what is called the trusted virtualized platform [8].

The question remains: How do we bootstrap trust in the association between a virtual machine and a vTPM? Currently, the specifications for "virtualized roots of trust" and "trusted virtualized platforms" as well as the academic literature offer no answer [3, 7, 8, 11, 23]. Instead, it is assumed that a client has prior knowledge of the relevant virtualization system and its TPM. Supposedly, a client can then "verify" her way up and determine, through a few recursive schemes that a particular vTPM is associated to her VM. While this assumption conveniently suspends the issue and allows higher level discussions, it never actually solves it. Practically a client knows nothing about the (1) virtualization system, (2) the physical TPM, or (3) the vTPM associated to her virtual machine. Unfortunately, without a way to assert that the VM is in fact associated to the vTPM and vice versa, any verifier can be tricked into establishing trust in an untrusted system.

We exploit the lack of a verifiable association and introduce *Goldeneye* attack. The attack leads a verifier to establish trust in an untrustworthy VM and establish trust in untrustworthy virtualization system. We use our *Goldeneye* attack as a vehicle to emphasize the need to bootstrap trust in a trusted virtualized platform by asserting and allowing the remote verification of the association between VMs and their vTPMs.

In this paper, we make the following contributions:

- We highlight the importance of VM-vTPM association by introducing and demonstrating *Goldeneye* which uses vTPMs against verifiers.
- We formally define a virtualization system supporting vTPMs (using bigraphs) and provide a trust model which captures our attack on the VM-vTPM association (using predicate logic).
- We show how related work [3, 7, 23] falls victim to *Goldeneye* and we discuss solutions to address the issue of associating VMs and vTPMs.

No straightforward solution appears to preclude *Goldeneye* entirely and we suggest improvements for virtualization systems which aspire to provide trusted virtualized platforms for its clients.

2 BACKGROUND AND PRIOR WORK

This section will briefly summarize relevant background terminology including virtualization systems and the Trusted Platform Module (TPM), both physical and virtual.

We make our definitions clear and easy to translate by adopting relevant terminology from protection profiles [2] and Trusted Computing Group (TCG) specifications [3]. We annotate definitions to lay out how elements are composed to form systems as a primer for our formal concept notation later on.

2.1 Virtualization System

The virtualization system is a software system which enables multiple independent computing systems to execute on the same hardware . The virtualization system in Fig. 1 consists of the physical platform (hardware), the virtual machine monitor / hypervisor, a management system and other helper components. The definition of a virtualization system is relative, and in the case of this paper, the virtualization system is the system supporting virtual machines running guest operating systems. However, if we would like to describe OS-level virtualization, the virtualization system would include all components required to reach the desired level of virtualization. In short, the virtualization system is the all encompassing abstraction enabling the execution of multiple operating systems on the same hardware. A virtualization system supports the virtual platform. The virtual platform, in case of Fig. 1 is the abstraction around the guest OS. Similar to a trusted platform, i.e. hardware + TPM, we define the trusted virtual platform as virtual machine + virtual TPM [8]. We give an inductive definition of a virtualization system and highlight the composition of the elements in the process.

2.1.1 Process (App). In our model, processes and apps are instances of programs. Each process is said to have a set of instructions (code), a finite address space (memory), a state structure, and resource descriptors. We assume that process are isolated from other process in the sense that one process can not directly modify resources owned by another. The process structure in general is defined by an operating system. In our model, there is a one-to-one correspondence between processes and apps and we use them interchangeably. Running apps as processes are supervised by an operating system. The security properties of processes and apps are relative to and best explained through an operating system.

2.1.2 Operating System. An operating system provides the *process* abstraction and runs programs. An operating system itself is a program with the purpose to control, abstract, and multiplex hardware for the benefit of running programs as processes on the system independently. The minimal operating system is referred to as the *kernel* which provides the process abstraction, virtual memory, scheduling, and inter-process communication. The operating system may also provide a number of other services which are not necessarily part of the kernel. We make no assumptions about the kind of operating system. Instead, we allow modelling *any* operating system by either defining additional functionality as part of the kernel or we model it using additional processes implementing operating system functionality on top of a kernel. As an element of our model, the OS abstracts a hardware interface and provides a process abstraction.

2.1.3 Virtual Machine. A virtual machine is taken to be an isolated duplicate of a real machine. Similar to a process, each virtual machine is said to own some memory partition, structures for the VM state as well as other resources. Operating systems running on a virtual machine are referred to as guest operating systems (Fig. 2.1). The concept of a virtual machine is relative to and best explained through the idea of a *virtual machine manager*.

2.1.4 Virtual Machine Manager (VMM). The virtual machine manager provides a virtual machine for other programs, e.g. guest operating systems, which is essentially identical to a physical machine. Despite allowing for a guest OS to run on the same machine, the virtual machine manager is in complete control of the machine and its resources. The minimal set of VMM functionality facilitating

[2] https://www.niap-ccevs.org/Profile/Info.cfm?PPID=408&id=408
[3] https://trustedcomputinggroup.org

virtual machines is referred to as the *hypervisor*. The hypervisor provides the virtual machine abstraction and acts as a dispatcher, resource manager, and if necessary interpreter for certain instructions. Virtual machine managers provide additional components such as drivers, emulators, virtual devices and management systems. Further functionality provided by a virtual machine monitor can be modelled by including it as part of the hypervisor or using virtual machines or processes—depending on the *type* of hypervisor we wish to model. We consider two *types* of hypervisors in our model. Type-I or *bare-metal* hypervisors expect a hardware interface and provide virtual machines. Type-II or *hosted* hypervisors are part of an operating system which abstracts the hardware while hypervisors provide the virtual machine abstraction. Both types provide a (virtual) machine interface in our system.

2.2 Trusted Platform Module

A Trusted Platform Module (TPM) [10] is a hardware, firmware, or virtual device which acts as a secure co-processor and supports securing machines in a number of ways: it can securely generate, store, and apply symmetric and asymmetric keys internally. TPM's can certify internal keys based on its root Endorsement Key (EK) [9] which is typically signed by the device or platform manufacturer. For simplicity, we say that TPMs are uniquely identified by a key pair $\{K_{TPM}, K_{TPM}^{-1}\}$. Root keys never leave the TPM and are therefore considered *secure* against all software attackers including compromised operating systems and highly privileged admins.

A particularly useful TPM feature is a special type of register called Platform Configuration Register (PCR). TPMs have a number of them and they essentially provide an append-only update operation which allows us to record transitions in the systems state in a concise way. The process of recording, or appending, system states, if continued properly, creates a "chain of trust measurements" which allows PCRs to reflect the boot process, launched applications [22], and internal states of a system [14]. Signing the recorded system state using an internal key produces a *quote*. The PCR values along with the quote can be securely conveyed to a (remote) verifier. Based on the PCR values, a verifier may check the system state and make judgements (e.g., are whether a machine is trusted or not). This process is referred to as remote attestation [4, 10]. Overall, we say that if we trust the TPM, we are able to establish trust in the (software) state of a system.

2.2.1 virtual TPM. In our model, all machines have TPM devices as separate chip or part of their firmware. TPMs, unfortunately, have no support for virtualization and are generally not shared among operating systems (hosts and guests). Instead, virtual TPMs are employed to serve as roots of trust for virtual machine. All virtual TPMs are essentially programs which need some execution environment. Their security directly depends on the environment and must be architecturally isolated from a guest operating system so as to support guest facing TPM use-cases (e.g., key storage outside guest OS and append only PCR updates).

Beyond architectural soundness[4], the vTPM must be implemented in a way that associates it to a VM the same way a real TPM is associated to a machine. On a real machine, this association can be safely assumed as the TPM is typically part of the chip-set. However, a vTPM is an element of the system and not implicitly associated to a VM. This weakness is exploited in our *Goldeneye* attack.

3 GOLDENEYE

The goal of a verifier is to establish trust in a VM and whatever runs in it using trusted computing technology. To this end, we need some fundamental trust assumptions outside of the typical trusted computing framework. The first assumption is that the verifier, or agent acting as a verifier, has a trusted device with which she can connect to a VM. Examples for this would be a local, trusted device which is used to connect to a service interface or create and manage VMs. An alternative to a trusted verifier setup would be using a trusted third party which can help a curious client with a trust decision. Further, we need agents to be able to trust someone (at least themselves) to vouch for the integrity of some system element. Without such assumptions, we can show that it may be hard to satisfy a general, secure cloud computing use-case. In summary, we assume that

(1) verifiers and their setups are trusted and
(2) trusted agents can vouch for other agents and system elements.

Practically, these two assumptions allow us to describe the cloud provider P as *semi-trusted*, i.e. its machines are trusted but the software may get compromised or misconfigured. Using (1) and then (2) we can state that verifiers trust cloud providers and that cloud providers vouch for the security of their machines, respectively. Consequently, we may at any point trust a physical platform and it's TPM. This and the overall complexity of our system and the fidelity of our model is what distinguishes *Goldeneye* from similar attacks like cuckoo [18]. Our attack succeeds on a physically secure machine.

3.1 Informal Description

In a *Goldeneye* [5] attack, an adversary effectively controls the vTPM associated with a virtual machine. In this attack, an adversary with some control over the virtualization infrastructure may swap out the vTPMs used to establish trust in a VM.

3.1.1 Setup. Figure 2 shows this attack in frame a) with a VMM_E which is the VMM expected by the client, the red coloring indicates that this VMM may not be acceptable because it does not provide necessary security properties or may be used to violate them. The vTPM is the VM's interface to the underlying trusted computing infrastructure. A verifier with access only to the VM relies on a vTPM to indicate the virtualization system correctly. To the right of VMM_E is another virtualization system which is part of the cloud infrastructure. The system to the right is virtually identical except that it runs a secure virtualization system. We simply name it VMM_O to which the vTPM_O points (O for *other*). An adversary can now, through a variety of channels (Section 4), funnel VM-vTPM communication to vTPM_O. The possibility of a *Goldeneye* attack has an effect on a number of security mechanisms. We focus on establishing trust in the software state of the virtualization

[4]We have an example of an *unsound* construction later on but we omit the formal definition here.

[5]Common goldeneyes are known to lay her eggs in the nest of other goldeneyes. By doing so they escape their parental investment which allows them to do other things.

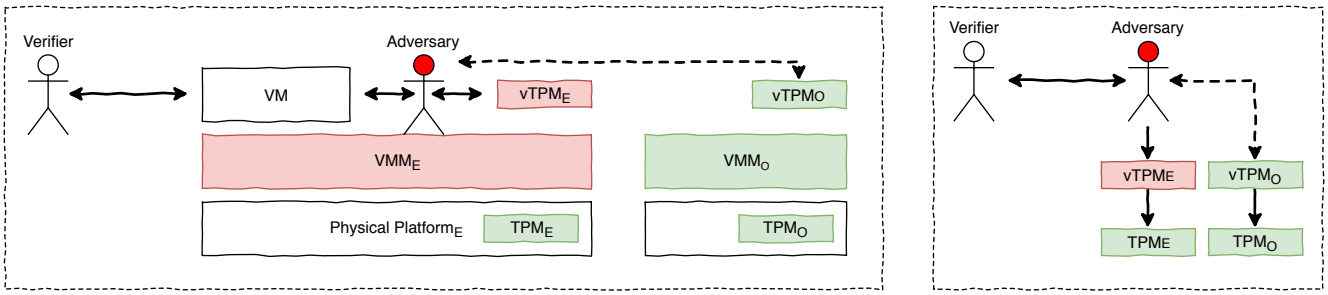

Figure 2: Sketch of our *Goldeneye* variant 2. The adversary has control over the VM-vTPM association. Adversaries use the vTPM (vTPM$_O$) to trick a verifier into establishing trust in another virtualization system (VMM$_O$).

system and the VM in this work. We identify two main variants of *Goldeneye* and discuss them below.

3.1.2 Variant 1 - changing virtual Root of Trust. Exploiting the VM-vTPM association on the same virtualization system enables the first variant of *Goldeneye*. The goal of the adversary in this variant is to allow trust establishment in a potentially untrustworthy VM. The variant is executed by changing the (virtual) root of trust associated with a virtual machine. An adversary with the ability to clone or fabricate vTPMs, can freely drop or inject certain TPM commands such as PCR updates. Furthermore, it would allow adversaries to install malware on in the VM and hide it by allowing only known-good PCR values in the vTPM. Secrets bound to a known-good state of the vTPM become accessible and remote attestation would allow a verifier to establish trust and release secrets to a VM which is not trustworthy.

3.1.3 Variant 2 - changing physical Root of Trust. The variant shown in Fig. 2 likens *Goldeneye* to a cuckoo attack [18]. The verifier in our setup has no prior knowledge about a vTPM, a virtualization system, and a TPM but trusts the physical components. An adversary access to the association between VMs and vTPMs can exploit the fact that commonly the virtual TPM is used as a pointer to the physical TPM. By associating a VM on some virtualization system with a vTPM on another virtualization system, an adversary can convince the verifier to establish that her VM is running on another virtualization system which may be trustworthy. This would allow an adversary to run virtual machines on bad virtualization systems while using the virtual TPM to hide the fact.

3.1.4 Executing the Attack. The attack exploits an architectural *gap* in the design of a trusted virtual platform. A TPM is used to establish the software state of a virtualization system. The "trusted" virtualization system hosts VMs and vTPMs. The vTPM presents TPM PCRs or a pointer to a TPM encoded in a vTPM Endorsement Key Certificate [3]. From a verifiers perspective, the vTPM tells the software state of the VM as well as the underlying virtualization system. Although a "strong" VM-vTPM association is a topic and requirement in prior work [3, 7, 8, 23], the software defined association is not verifiable (nor enforceable by trusted elements) and thus a concerned client can not establish trust in it. This in turn allows an adversary to carry out a *Goldeneye* attack without detection or spoiling the known-good state of any virtualization system. We found that *Goldeneye* can be executed by using standard tools to manage VMs and vTPMs. For example, Xen allows attaching and detaching vTPMs to VMs (Xen-*domains*) as part of the vTPM management interface[6]. Usage of such commands does not spoil the known-good state of the virtualization system in the TPM. Given the complexity of a virtualization system and its interfaces, we expect more opportunities for an adversary.

Without improvements to the VM-vTPM association, the possibility of a *Goldeneye* attack contradicts the declared objective of [7, 23] to reduce the amount of trust needed in a *semi-trusted*[7] provider.

3.2 Graphical Model

A key feature of the model visualized in our introductory figure (Fig. 1) is its inherent layering. The entire system is visually and functionally separated into different layers such as hardware, VMM, virtual machines / guest OSes and applications. The form of its structure and transactions can be captured quite naturally using layered graphs. A bigraph [17] is a form of layered graph that superimposes a spatial *place graph* of physical or virtual locations and a *link graph* which denotes interaction on a single set of nodes.

The system in our introductory example (Fig. 1) can be translated using virtualization system to form the graph of Fig. 3. We formalize it in a bigraphical fashion by defining *linking* and *placing* independently. Thereupon, we describe *interfaces* for *composition* (i.e., how to create larger system out of elements) which we constrain with *controls* for different kinds of nodes. Finally, we offer a natural translation of our graphical structure and compositions to formulae of first-order logic with which we further formalize *Goldeneye*.

3.2.1 Interaction. To express interactions between nodes (elements) in our model we introduce a *link graph*. In a graph with nodes V and edges E, edges join pairs of nodes. A *hypergraph* is a generalization which allows edges to join any number of nodes. Hyperedges can be seen in a straightforward manner using only edges (u, v), $u, v \in V$ with node u being part of the original graph and placing v as a

[6] https://wiki.xenproject.org/wiki/Virtual_Trusted_Platform_Module_(vTPM)
[7] Semi-trusted agents and elements are trusted but they may become compromised in an attack.

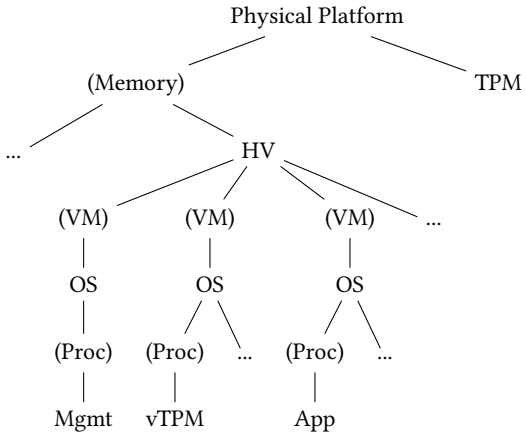

Figure 3: Graphical model of a virtualization system showing the *placing* of its elements. VMs and Processes are included as abstractions for guest OS and apps, respectively.

inserted "hyperedge" node for each hyperedge $e \in E$ to join any number of nodes at v.

Definition 3.1 (Hyperedge). A Hyperedge is an edge $e \in E$ between any number of nodes V. E is a non-empty set of non-empty subsets of V ($E \subset \mathcal{P}(V) \setminus \{\emptyset\}$).

Each node $v \in V$ has an *arity* and has *ports*. The arity, e.g. a natural number, both names and limits the number of ports for each node available in the hypergraph ($P_v := \{(v, i) \mid i \in ar(v)\}$). Thus the ports available in the hypergraph is the union of disjoint sets P_v, $v \in V$ ($P_V := \uplus_{v \in V} P_v$). The hypergraph is then defined by the quadruple

$$(V, E, ar, link)$$

in which the ar is a mapping of nodes to ordinal numbers ($ar : V \to ordinal$), and $link$ assigns each port to an edge in E ($link : P_V \to E$) (Fig. 4) .

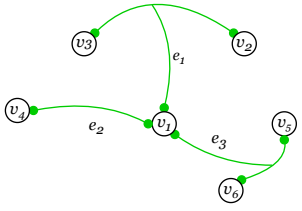

Figure 4: A hypergraph with nodes $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ and edges $E = \{e_1, e_2, e_3\}$. Nodes are circles, ports are blobs, and an edge links ports.

We describe VM-vTPM association using a link graph later on. Another typical interaction in a virtualization system is *controlling* or *configuring* in the sense that one element configures another one, e.g. launching or deleting virtual machines.

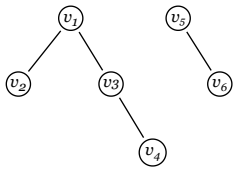

Figure 5: A forest over nodes $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ defined using a parent mapping $prnt_V(v \in V)$.

3.2.2 Placing. We use a place graph to express Figure 3 more formally and using the same nodes as before. This gives us a bigraph with nodes V and edges E has a hypergraph (V, E) and a superimposed forest with nodes V [17]. Our representation of a virtualized system naturally resembles a parent-child relationship where some elements supervise others. Including a mapping over nodes $v \in V$ in our hypergraph which allow us to express parent-child relationships.

Definition 3.2 (Place Graph). The graph of figure 3 is a hierarchical tree of vertices and a parent mapping. V is the set of nodes and $prnt : V \to V$ is the parent map and defines the *nested* place structure. The parent map is acyclic (i.e., $\forall k > 0, v \in V, prnt^k(v) \neq v$).

With the superimposed forest and parent mapping from $prnt : V \to V$ we make our hypergraph into a bigraph. Consequently, the 5-tuple

$$(V, E, ar, link, prnt)$$

describes our bigraphs alone and without *interfaces*.

Continuing the example in Fig. 4, we superimpose a place graph in Fig 5. The resulting bigraph allows nodes to be placed and linked independently and freely. However, we need a way make more precise definitions for the kinds of system we wish to model. Indeed, components of a virtualization system can not be nested arbitrarily and not all nodes may be linked or be part of interactions.

3.2.3 Controls. The nodes of a bigraph are of different kinds and they contribute differently to its dynamics. To each node in a bigraph is assigned a *control* to express properties of its kind. Each model may define different controls, specified—together with arities—in a *signature* such as

$$\mathcal{K} = \{\mathrm{X} : n\}$$

which states that nodes of a kind X have n links and so on. Controls serve a purpose similar to a type system by defining the arity for kinds of nodes. The set of controls is referred to as a bigraphs *signature*. Consequently, in any bigraph with a control $\mathcal{K}$, the arity of a node is the arity of its control. Bigraphs with controls are written as

$$(V, E, ctrl, link, prnt)$$

where ar has been replaced by the now appropriate $ctrl$ for each node. To model our virtualization system, we might simply say that nodes of the vTPM kind have exactly one link (to a VM). We might say the same about a VM which also has only one link (to a vTPM). Stating only those controls ($\mathcal{K}_{\mathrm{VS}} = \{\mathrm{vTPM} : 1, \mathrm{VM} : 1\}$) lets us constrain our model to only consider interactions like VM-vTPM association.

3.2.4 Interfaces. A bigraph has interfaces, which define its use as a building block to make larger bigraphs from smaller ones. Interfaces are defined independently for the *link-* and *place graph*. Interfaces are separated into *inner* and *outer* faces allowing composition of bigraphs in an intuitive way: if the outer faces of a bigraph match the inner names of another, the graphs may be connected at these vertices.
All interfaces take the form $I = \langle n, X \rangle$. If $X = \emptyset$ we say $I = n$, if $n = 0$ we say $I = X$, or $I = x$ if $X = \{x\}$. If $n = 1$ the interface is said to be *prime*. The empty interface $\epsilon = \langle 0, \emptyset \rangle$ is called the *origin* of a bigraph [17].
Bigraphs can be joined either by nesting or by linking. Linking simply means to connect the inner names of some host or contextual bigraph with the outer names of the bigraph which we would like to compose. E.g., if a host graph H has inner names $\{x, y\}$ and graph F has $\{x, y\}$ as its inner names, then we may join the link graphs of H and F at $\{x, y\}$.

The interfaces of a place graph are referred to as *sites* (or holes) and *regions* (or roots) for it's *inner* and *outer face*, respectively. Sites and roots are devices to define composition by *nesting* in a place graph. The abstraction is quite intuitive in that sites act as *holes* for nodes. Given that each node without another node as parent must be a *root*, we still need a way to express *where* this root node can be placed. To do so, we explicitly give each such node a virtual *root* node m, (m, v). The set of roots and the set of nodes are disjoint. The counterpart to the virtual root is the *site*. Nodes may or may not have a site, i.e. a point at which we can add another graph. Simply put, graphs may be joined at the roots and sites by *plugging* the roots of one graph into the sites of another. The sites are the inner face of a place graph. As an example, we could describe the forest in Fig. 5 as a place graph which has exactly two roots, the root of v_1 and the root of v_2, and offers no sites at which we could add more nodes (or a graph). Furthermore, the links according to Fig. 4 are all closed, so there are no inner names at which we could join other nodes (or a graph). We write this as $\epsilon \rightarrow \langle 2, \emptyset \rangle$.

3.2.5 Composition. When treated categorically, bigraphs are simply *arrows* and interfaces are corresponding *objects* in a symmetric (partial) monoidal category (spm-category) [17]. As a result, the composition of bigraphs is definable in terms of the composition of arrows in the category. Elements of the virtualization system are the arrows and their interfaces are objects.

We distinguish between composition by nesting and composition by juxtaposition. If we want to express nesting, we use a . (dot). E.g. to express Fig 5 we can write $v_5.v_6$ to express that v_6 is nested in v_5.

If we want to express that two nodes are juxtaposed we write | between them. We express the first tree in the same figure as $v_1.(v_2 \mid v_3.v_4)$. By using || to indicate different roots, we can express the entire forest as $v_1.(\ldots) \mid\mid v_5.v_6$.

In short, the composition of two bigraphs is defined by matching the inner interface of the first graph with the outer interface of the second. For nesting, this means (a) filling the sites of the first graph (holes) with regions of the second graph and (b) merging the inner names of the first graph with the outer names of the second graph.

The virtualization system can be described in terms of elements. We can describe their interfaces independently for each element and then composing them as individual bigraphs with one node each. We will simply treat the virtualization system as a contextual graph since we are not concerned with different kinds of virtualization systems. For simplicity, we say that each virtualization system has exactly one root and offers *sites* for a number of vTPMs and VMs. Virtualization systems can be juxtaposed. VMs and vTPMs have exactly one link x which allows them to be joined. VMs offer sites for an OS whereas vTPMs offer no sites for nesting.

3.2.6 Semantics. Since we aim to develop a model with which we can reason about simple properties of a system, we need to attach some meaning to the graphical model we have introduced. Most importantly, we will be likening the place graph and nesting with an implicit dependence, i.e. a child node depends on parent, and the link graph with an optional dependence, e.g. one element might control another. The integration of bigraphs with logical frameworks is studied in depth and in relation to spatial and separation logic in [5]. Our reasoning follows the naive notion that a child depends on a parent—any assumptions we make about a node directly depend on the assumptions we make about the parent of that node. This dynamic can be given as a set of inference rules for introduction

$$\frac{P(y) \qquad P(x) \qquad (x.y)}{P(x.y)} \text{ -Dot} - \text{I}$$

and elimination, respectively:

$$\frac{P(x.y)}{P(x)} \text{ -Dot} - \text{E}_1 \qquad \frac{P(x.y)}{P(y)} \text{ -Dot} - \text{E}_2 \qquad \frac{P(x.y)}{x.y} \text{ -Dot} - \text{E}_3$$

The transitivity of (.) *dot* composition is studied and detailed in [17]. Our reasoning over properties follows this exactly and we can intuitively explain the transitivity of our Dot rules above.

In addition to nesting (Dot-Rule), we have a *controls/configures* relation between two nodes which is derived from the link graph and therefore orthogonal to nesting. In short, the controls/configures relationship extends the nodes which we have to consider in our reasoning. Whenever we naturally have to consider parent nodes of a node of interest, we can expand this set of nodes by declaring nodes which *controls* the node of interest. The reasoning is no different in the sense that any property we want to show about a node, we also have to show in the node that configures it. The configures relationship is an edge in the hypergraph and requires that if x configures y ($x, y \in V$) then, in order to prove a property of y, we need to also prove the property in x. In other words, if x configures y, we can conclude $P(y)$ only if we can prove P for *x and y*. The rules for this kind of composition allow similar inferences to the Dot − E/I rules above. However, x and y are not necessarily in a *parent-child* relationship. In short, if we wish to prove a property about $P(y)$ with the knowledge that y is configured by x, $P(x)$ and $P(y)$ need to agree as the proof of $P(y)$ *depends* on the result on a proof of $P(x)$.

3.3 Trust Model

In this section, we construct a trust model which will bring our formal discussion as close as possible to actual reasoning about trusted systems. To analyze *Goldeneye*, we model different situations using predicate logic with our graphical model as context. The graphical model describes the system of interest and predicates and axioms

Table 1: Trusted Computing Predicates

Predicate	Meaning
$\text{Trusted}_{\text{A}}$(a)	Agent a is trusted
SaysSecure(a, e)	a says element e is secure
$\text{Trusted}_{\text{VS}}(vs)$	vs is trusted
On(tpm, m)	TPM tpm is on m
$\text{Trusted}_{\text{TPM}}(t)$	TPM t is trusted
$\text{Trusted}_{\text{VTPM}}$(vtpm)	vTPM $vTPM$ is trusted
Bound$(vtpm, vm)$	$vTPM$ is bound to vm
SaysBound$(vm, vtpm)$	vm says $vtpm$ is bound
$\text{Trusted}_{\text{VM}}$(vm)	vm is trusted

Table 2: Axioms

$\forall a, e$ $\text{Trusted}_{\text{A}}(a) \wedge \text{SaysSecure}(a, e) \supset \text{Secure}(e)$
$\forall t, m$ $\text{On}(t, m) \wedge \text{Secure}(m) \supset \text{Trusted}_{\text{TPM}}(m.t)$
$\forall vm, vtpm$ $\text{Secure}(vm) \wedge \text{SaysBound}(vm, vtpm)$ $\supset \text{Bound}(vm, vtpm)$
$\forall vtpm, vm$ $\text{Bound}(vtpm, vm) \wedge \text{Trusted}_{\text{VTPM}}(m.vs.vtpm)$ $\supset \text{Trusted}_{\text{VM}}(m.vs.vm)$
$\forall t, vs$ $\text{Trusted}_{\text{TPM}}(m.t) \supset \text{Trusted}_{\text{VS}}(m.vs)$
$\forall t, vs, vtpm.$ $\text{Trusted}_{\text{TPM}}(m.t) \wedge (m.vs.vtpm)$ $\supset Trusted_{VTPM}(m.vs.vtpm)$

describe conventional reasoning strategies for trusted computing systems. Table 1 details the trust model with predicates and general deductions are given in this Table. Table 2 shows axioms of our system. The axioms allow inferences which follow a natural and mostly conventional reasoning over trusted computing systems.

We begin our formal modelling process by defining a vulnerable system as a bigraph. Then we state our assumptions and exercise a reasoning process about the system by applying accepted axioms to assumptions. This allows us to conclude that a VM is trusted the VM should not be trusted—we use the contradiction to demonstrate the possibility of a *Goldeneye* attack.

3.3.1 Variant 1: chaining virtual Root of Trust. In the first variant of *Goldeneye* the adversary changes the virtual root of trust of a VM by switching between multiple instances of a vTPM. The following bigraph describes the system setup:

$$/x\ (vm : VM)_x \| (vtpm1 : VTPM)_x \| (vtpm2 : VTPM)_x$$

We have a closed link x denoted by $/x$ among VM and vTPMs after composing them. We can then compose this graph with a virtualization system as context:

$$m.tpm | vmm.(\square : VM | \square : VTPM)$$

The squares indicate sites and $: VM$ to indicate which kind of element it is for. This effectively encodes the setup of the first variant of a *Goldeneye* attack. Next, we encode assumptions about our semi-trusted provider P:

(1) $\text{Trusted}_{\text{A}}(P)$—P is a trusted agent,
(2) $\forall m$, SaysSecure(P, m)—P asserts that machines are secure,
(3) (1), (2) $\supset \forall m.\text{Secure}(m)$,
(4) $\forall m, t.\text{On}(m, t) \supset \text{Trusted}_{\text{TPM}}(m.t)$

The semi-trusted provider allows us to regard all TPMs involved as trusted. Then we encode the assumptions about our client as follows:

(1) $\text{Trusted}_{\text{A}}(C)$—C is a trusted agent,
(2) SaysSecure(C, vm)—C assumes that vm is secure,
(3) SaysBound$(vm, vtpm_1)$—Client's VM says that $vtpm_1$ is bound
(4) $(m.vmm.vtpm1_x)$—vTPM 1 is on $m.vmm$
(5) $(m.vmm.vtpm2_x)$—vTPM 2 is on $m.vmm$
(6) $(vtpm1_x), (vtpm2_x) \supset \neg\text{Bound}(vm, vtpm1)$—VM could be associated to either $vTPM1$ or $vTPM2$.

In summary, we assume that the client trusts herself. The client provided what she believes to be a secure VM. The VM says its bound to a vTPM $m.vmm.vtpm_1$. By applying our axioms, we can first conclude establish trust in the virtualization system ($m.vmm$) and $m.vmm.vtpm_1$. From our trusted vTPM, we now conclude that our bound virtual machine is trusted as well. The conclusion is $\text{Trusted}_{\text{VM}}(m.vmm.vm)$.
At the same time, we also know that $vTPM2$ exists with association x. This implies that neither $vtpm1$ nor $vtpm2$ are bound to vm. Consequently, $m.vmm.vm$ is not trusted.

Effectively, we would have to prevent the existence or prove the absence of more than one vTPM with port x, where x is the VM's port to which vTPMs can be attached. We will discuss possible mitigation below.

3.3.2 Scenario 2: changing physical Root of Trust. In the second variant, the adversary does not use the possibility of more than one vTPM per VM. Instead, she exploits that we assume the root of trust of the vTPM to be the root of trust of the VM (Fig. 2). We conclude that a particular VM is trusted based on a chain of trust measurements of a bound virtual TPM on a different machine. The following bigraph represents the target of the attack:

$$/x\ vm_x \ || vTPM_x$$

With two different virtualization systems as a context graph:

$$m1.vmm1.(\square : VM | \square : VTPM) || m2.vmm2.(\square : VM | \square : VTPM)$$

We adopt the assumptions of our semi-trusted provider P and our client C as the verifier but explicitly state that the machines of vm_x and $vtpm_x$ are different. When we apply axioms, again, we conclude that the vTPM $m2.vmm2.vtpm_x$ is bound to the vm. TPM $m2.tpm$ lets us establish trust in the virtualization system and consequently, we trust the vTPM. A trusted and bound vTPM lets us establish trust in the VM.
However, vm is not on $m2$ and consequently, we can not say anything about it. In fact, we have to assume that $m1.vmm1$ is not trusted and that consequently the VM is not trusted.

4 EVALUATION AND DISCUSSION

In this section we will review existing work, show how and why a *Goldeneye* attack could succeed, and discuss possible improvements towards the end. To our knowledge, Berger et al. presented and evaluated the first vTPM implementation [3]. A secure vTPM migration process and security improvements are introduced in [7]. Finally, Schear et al. introduce *keylime* as a cloud key-management system which involves integrity reporting using virtual TPMs and so called *deep quotes* of the infrastructure. We demonstrate our attack

by giving an overview and abstract architecture and by reasoning using previously introduced strategies.

4.1 Goldeneye in Related Work

4.1.1 Virtualizing the TPM [3]. The implementation of the TPM in software presented by Berger et al. presented in 2006 has evolved and is now used across architectures and platforms. The vTPM implementation is composed of a vTPM manager and a number of vTPM instances. Intuitively, each VM is assigned a virtual TPM instance. The vTPM manager performs functions such as creating vTPM instances and "multiplexing" requests from VMs to their associated vTPM instances. The architecture of the solution follows the Xen model by running vTPM extensions in a VM (Fig. 6).

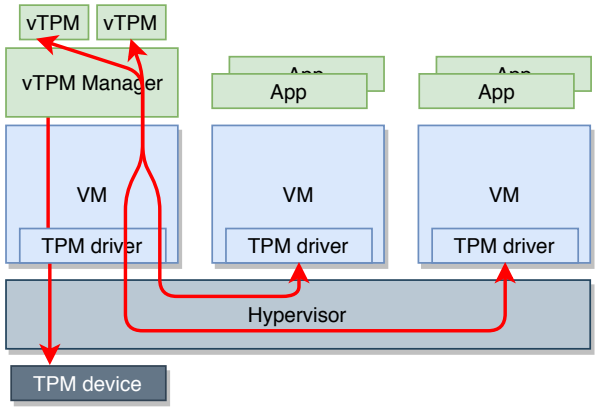

Figure 6: Architecture of the Xen-based vTPM extension.

Association between vTPM instances and VMs is one of the concerns raised in their paper. To address this concern, instantiating, migrating, destroying etc. have all been implemented to require some form of authorization. In short, unless a command to create or migrate a new vTPM instance is authorized, it would be blocked. Whoever administrates the system is also allowed to handle vTPM management. We model the system using three bigraphs: the virtualization system as the contextual graph for the vTPMs and VMs (Fig. 7).

The intention of [3] is that once a VM with vTPM support is launched, a vTPM instance is created. The VM has x as its interface and can host arbitrary apps / processes. The vTPM on the other hand requires a vTPM manager in its root (we do not explicitly encode this) and has x as its outer face. After composing the graphs and joining VM and $vTPM$ at x the link is closed and a vTPM associated. This process is the same for every other vTPM and VM. Without further assumptions, we can reduce this scenario to the attack scenario one. Following the same reasoning, we arrive at the conclusion that we can establish trust in our VM. However we can not exclude that there can be no other vTPM instance associated to our VM ($x = \{vm_x, vtpm1_x, vtpm2_x\}$).

4.1.2 Secure vTPM-VM Migration [7]. Migrating VMs and the ability to do so relatively freely and at any time is one of the most distinguishing features of virtual machines compared to operating systems running on hardware. Performing migration securely is a challenge and performing migration of the vTPM in a secure and trustworthy manner a priority. Typically, the cloud provider can be modelled as an agent having more than two machines at its disposal [7], all equipped with physical TPMs and virtual TPMs for VMs. Each virtual machine interfaces with the physical TPM through a software vTPM. [7] assumes that vTPMs do not contain hardware and hypervisor configuration information—unlike [3]. Instead, this information is held by the physical TPM and obtained by querying it directly. Vice versa, the hardware TPM does not include any VM specific information. A perfect setup for *Goldeneye*. The setup evaluated by Danev et al. is shown in Fig. 8.

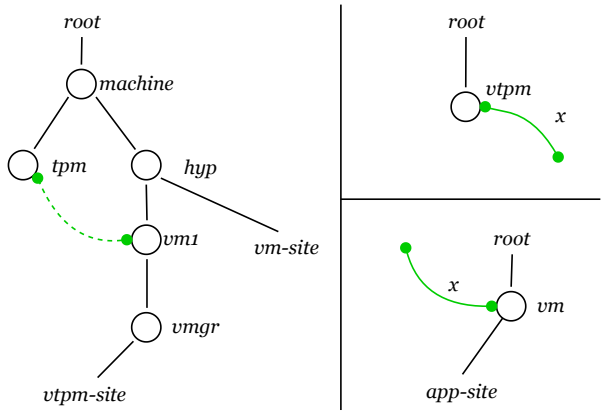

Figure 7: Bigraphical abstraction of the Xen-based vTPM extension. We model three separate graphs: the virtualization system and the vTPM extension (left), virtual TPM instances (top right), and VMs with vTPM support (bottom right).

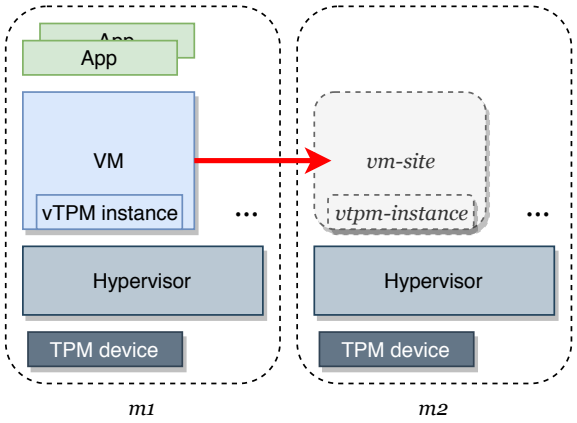

Figure 8: Secure vTPM-VM migration setup. The vTPM instance, according to [7] is running in a VM itself. When migrated, the VM is shutdown and transferred under encryption from $m1$ to $m2$, deleted on $m1$ and resumed on $m2$. We denote this without discussing the protocol by placing a bigraphical $vm - site$ on $m2$ which may become inhabited by the target VM.

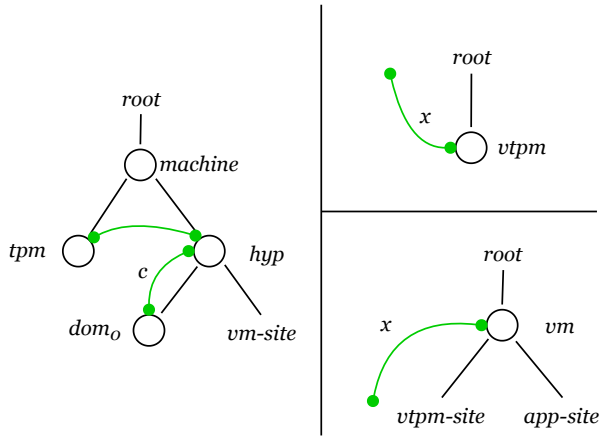

Figure 9: The proposed system model of Danev et al. which uses Xen with virtual machines hosting their own vTPMs. The solution makes no claims about the Xen specific privileged domain dom_0 managing the virtualization system (left). We denote the privilege using the edge $c = \{dom_0, hyp\}$ to state that dom_0 configures hyp. To make the vTPM definition compatible with the [7], we denote x as being an inner face of $vTPM$ and an outer face of a VM. The composition $vm.vtpm$ with the association $\{vm, vtpm\}$ is discussed later on.

The migration protocol dictates that only upon successful attestation, implying that both machines are *honest* [7], the VM and vTPM are migrated—the memory image of the selected VM which includes the vTPM is migrated. Ignoring the dynamics of this scenario, we express the system proposed as the following bigraph (Fig. 9):

The construction in 8 and 9 allows a straightforward migration process [7].

Intuitively, running the vTPM $vtpm$ in the VM vm while serving as the root of trust for said VM does not fit into the way we naturally reason about trusted computing systems. Figure 10 allows us to conclude that $vtpm$ can only be isolated from applications but not the VM.

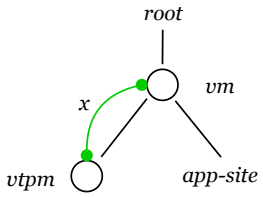

Figure 10: Assembled vTPM-VM in the system model of [7]. The composition of $vm.vtpm$ with the association $x = \{vm, vtpm\}$ produces a circular argument when establishing trust.

Using our axioms, we would like to establish trust in a VM, in this case vm. As per usual, we trust it when the VM tells us that it is bound to some vTPM. We also need to find and establish trust $vtpm$ which takes us back to proving Trusted($m.hyp.vm$). Danev et al. explicitly do not include VM information in a TPM and this prevents us from moving beyond our initial assumption Secure(VM).

We wanted to show the vulnerability to *Goldeneye* by proving the usual contradiction Trusted(vm) $\wedge \neg$Trusted(vm) but we simply failed to establish trust in the vTPM. In our model, the presented construction is simply not sound.

4.1.3 Bootstrapping and Maintaining Trust in the Cloud [23]. Schear et al. describe a system which aims to provide a *hardware rooted* root of trust for virtualized environments in an infrastructure as a service (IaaS) environment. In the IaaS environment, a cloud provider offers to run a client's workloads, as virtual machines for example, on a provided system. Since the TPM is proven to be useful in physical infrastructure, the authors aim to port the functionality to virtualized systems using vTPMs. To this end, *keylime* is introduced as an end-to-end solution for both bootstrapping hardware rooted cryptographic identities for IaaS nodes and for system integrity monitoring of those nodes via remote attestation. This is supported both on bare metal and in virtualized environments using virtual TPMs. Such an architecture can in theory simply provide vTPMs in support of higher level security mechanisms such as disk encryption and integrity monitoring.

Like previous papers, *keylime* also assumes a semi-trusted cloud provider and considers rouge administrators as part of their threat model. Furthermore, it is assumed that no tampering may occur on physical components, including the TPM, and that the hypervisor is not compromised. The authors wish to detect or defend against an attacker trying to gain persistent access to a resource. They assume that in order to do so, the attacker would have to perform the kind of modification which is detected by integrity measurements.

A node in *keylime* is implemented based on Xen and with extensions of [3]. Consequently, vTPMs are isolated from other guest VMs. The vTPM interface is TPM compatible, meaning that no modifications in trusted computing enabled guest operating systems is required. *keylime* adds a special command called *DeepQuote* which will get a quote form the TPM device. A quote of the TPM which can be *linked* to the vTPM quote is therefore considered to be a *deep* quote. The process of deep quoting in [23] is defined as performing a vTPM quote first (*shallow* quote). The hash of a nonce with the output of a shallow quote is then provided as the nonce for the TPM quote. Discussing the soundness of this approach is beyond this paper. However, we use this deliberate two step protocol with which the vTPM quote seems to be authenticated in our trust model. We express the node model of *keylime* as the following bigraph (Fig. 11):

One of *keylime*'s contributions is to form a pool of trusted IaaS nodes via TPM endorsement key enrolment and continuous attestation. The VM can be attested using the vTPM and if necessary, the infrastructure below the VM and vTPM can be attested using a *deep*-quote. Following our process we would conclude that a vTPM is bound to the VM. The vTPM is also bound to the *other* hypervisor—in fact, the binding of the vTPM to a machine and its TPM is not useful in this case. By way of a deep quote—as defined in [23]—we learn from ($m.tpm$), the trusted TPM device of a platform m that the VS and vTPM ($m.hyp.dom0.vtpm - mgr.vtpm$) can be trusted.

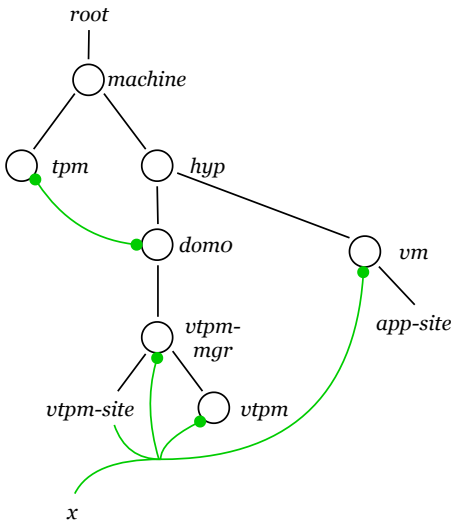

Figure 11: IaaS node model of *keylime* using vTPMs associated to VMs. All vTPMs are isolated from guest VMs vm and hosted on a privileged domain with access to the real TPM. The hyperedge x denotes the vTPM-VM association and that the vTPM-manager is responsible for the association $\{vtpm, vm\}$.

Subsequently, we are able to conclude that then also ($m.hyp.vm$) is trusted which was what we set out to prove. However, there is no guarantee that $m.hyp$ is the same for the VM and the associated vTPM. Without extra assumptions, keylime is vulnerable to a *Goldeneye* attack.

4.2 Solutions

The *Goldeneye* attack is possible whenever an attacker can control the association between VM and a virtual TPM. Indeed, our review shows that this association is at least mentioned and sometimes addressed in solutions involving a vTPM. However, we only see assertions about a *strong* association. No mechanism is proposed which would with a degree of certainty assure a strong association (i) or, perhaps more importantly, enable a verifier to detect potentially weak or broken associations (ii).

A similar problem partially exists in a *bare-metal* scenario. It is elaborated in [18] that without *knowing* the TPM on a particular machine, an adversary on that machine can convince a verifier to establish trust in another machine equipped with another TPM. The association between a TPM and an execution environment is generally not a problem on a real machine and needs special consideration on a virtual platform. Consequently, there is no analog concept which we can simply adopt.

We discuss state-of-the-art trusted computing techniques and how they can be applied to remedy the problem of a remotely verifiable association.

4.2.1 Trust by default. The most obvious solution to the association problem is to make more assumptions or trust the providers assertion that VMs and vTPMs are bound. The problem and possibility of an exploit is known. A standard way to address it on the providers behalf is to acknowledge the problem and make assertions through some level of validation: it is impossible to change initial associations and a vTPM would never indicate a root of trust other than the one underlying the VM as well.
Pros: Requires no changes to the sate of the art and requires only awareness of the problem on client and providers' side. Client can simply trust that *Goldeneye* configurations are not present.
Cons: Virtualization systems in their entirety quickly grow complex and expose attack surfaces and enable bad configurations. Xen-based solutions today offer specific tools for *listing*, *attaching*, and *detaching* vTPM instances from Xen domains. For a verifier, this means that large parts of the infrastructure simply have to be trusted from the start in order to establish trust in a virtual machine and virtualization system.

4.2.2 Removing malware. An alternative approach to thwart *Goldeneye*'s attacks is to remove malware on a cloud system. With a sure way to run only malware-free cloud nodes, it wouldn't matter which virtualization system in particular a VM is running on. In our formal model, this would result in adding the assumption that all components of the virtualization system, including the vTPM, are already trusted.
Pros: All issues related to security and trust are now deferred to a client's VM. As long as this VM is trusted, naturally, the composition with a virtualization system is trusted also.
Cons: This approach tends to be circular—the whole point of getting quotes of a VM or deep quotes of the virtualization system is to check its make, compliance, and ultimately absence of any malware and to make a trust decision of the running system.

4.2.3 Removing interfaces. We showed that *Goldeneye* attacks are thought to be possible since there is no way for a verifier to check the setup of her virtual platform as soon as the provider runs it. The easiest way to produce a *Goldeneye* attack is to change vTPM instances *locally*. As outlined earlier, those actions are considered to be *malicious configurations* and do not require malware or any other detectable software additions.
Pros: Prevents the reconfiguration of VM-vTPM associations while the VM or the vTPM is running by malicious admins.
Cons: Actions such as attaching and detaching vTPM instances are still needed. If implemented elsewhere—potentially as part of the hypervisor—will increase the trusted computing base and negate disaggregation efforts.

4.2.4 Integrity measurements. Various integrity measurement architectures have been proposed to extend a chain of trust (measurements) from the TPM and a trusted BIOS and boot loader all the way into operating systems and application space [14, 20, 22, 24]. Similar concepts have been proposed for hypervisors and VMMs [1, 15]. Using such measurement architectures, we could record running VMs and their images as well as vTPM software in the physical TPM.
Pros: Integrity measurements of vTPM and VMs in the TPM could help linking a VM to a hypervisor or VMM. Combined with integrity measurements of the vTPM software, this would allow us to conclude that an instance of a VM is executed on a virtualization

system running trusted vTPMs.
Cons: Asserting (using measurements) that a virtualization system runs a certain *kind* of vTPM, shown via hash of its binary, is helpful but provides little information as to whether a particular *instance* is associated to a VM on the platform. Furthermore, measuring vTPM instances as well as VMs will cause frequent updates to the TPMs PCRs. Aside from performance issues, frequent changes to PCRs make it hard to support TPM-based encryption and binding of secrets to a platform state.

4.2.5 Recording vTPM-VM Association. A promising solution, which would confine adversaries, is to record interactions with vTPM and vTPM management interfaces outside of the normal operation. An example would be to allow changes to the vTPM association but to record them before applying those changes. This would prevent an adversary from switching between local instances, or routing vTPM commands to another vTPM instance on another machine using known channels. Interface access and issued commands can be extended into TPM and vTPM PCRs to record and signal the interference.
Pros: Implementing this can be straightforward by extending the vTPM implementation and manager at the interfaces. Supporting protocols and TPM policies could react to such changes by making keys for storage and attestation keys unavailable.
Cons: This approach on assumptions about the vTPM implementation on a virtualization system. Asserting that an implementation will record adversary interference and concluding that—unless reported—there can be no interference requires that we fully trust the implementation in the first place (comparable to "trust by default" in 4.2.1).

So far, we have viewed vTPMs as processes running alongside other vTPMs in an environment controlled by the vTPM manager (Fig. 7). We will now discuss architectural solutions which change how vTPMs are implemented on a virtualization system.

4.2.6 Unikernel vTPMs. Disaggregation is a core concept in the development of Xen[2] and comparable in philosophy to *microkernelification* of systems [13]. The same principles can be applied to implementing vTPMs. Currently, vTPMs are implemented as applications / processes which run on top of an OS. Unikernels [19] provide the necessary interfaces between applications and real hardware without the need of a complete OS. Instead, unikernels allow us to run applications with minimal overhead directly on a physical or virtual machine. Each vTPM in Xen is running on a small guest operating system as a virtual machine or domain. All vTPM domains are controlled by a vTPM manager in a privileged domain which supervises VM-vTPM communication. By excluding the vTPM management, its code base, and interfaces, we could achieve a design which practically involves *only* vTPMs and associated VMs.
Pros: Using unikernels would allow the implementation of vTPMs as standalone components. With the association deferred to the hypervisor, this would significantly reduce the interfaces and amount of code that is part of the trusted computing base in our vTPM design.
Cons: We would have to implement the *association* as part of the hypervisor or involve a privileged domain to handle association for us. Further, migration in such a design is not as straightforward as in [7] and would require migrating vTPM and VM separately.

4.2.7 vTPM as part of VM abstraction. When we discussed possible vTPM architectures, we have so far only considered having vTPMs as *hosted* processes. Alternatively, we could also include vTPM functionality in a hypervisor or at least provide a way to save and restore vTPM states per VM as part of the virtual machine control structure.
Pros: We effectively reduce the trusted computing base of the vTPM and place the responsibility of associating and maintaining the association between VM's and vTPMs in the hypervisor. This limits the interfaces to the vTPM and the possibilities for attackers to make changes.
Cons: Putting vTPM code and functionality in the hypervisor would obviously contradict the design philosophy of disaggregation. Furthermore, vTPMs are essentially VM controlled, we would provide clients with a *window* to the hypervisor through vTPM commands. Tools limiting resources of VMs would not be able to also protect hypervisors if they executed vTPM commands on clients behalf.

4.2.8 vTPMs in secure execution environment. The default solution today is to run vTPMs as processes. For a KVM/Qemu based solutions, those would be processes implementing the back-end of a Linux vTPM device and on Xen it would be vTPM processes as in [3, 23]. Running a vTPM in a secure execution environment is the next step to reducing the vTPMs trusted computing base (TCB). Separating the vTPMs TCB from the *untrusted* virtualization system would allow clients to use vTPMs even if the virtualization system is not trusted. Examples of *secure* execution environments, i.e. isolated from the VS, include Intel's Software Guard EXtensions and MIT's Sanctum [6] as well as trusted execution environments (TEEs). Environments which are isolated from the VS also include code which runs in system management mode (SMM), or generally, closer to what would be considered firmware of a machine.
Pros: The the trusted computing base of the vTPM would be independent from the VS and comparatively minimal. Furthermore, the attestation and root of trust of such a solution could be independent from a TPM and would therefore scale better than deep quotes involving TPMs.
Cons: While vTPMs are architecturally isolated from the virtualization system, we still rely on the VS to make the connections between VMs and the vTPMs. Furthermore, since vTPMs and VMs no longer have a shared trusted computing base, vTPMs would have to explicitly include the VMs TCB as well. Generally, we found that changing the trusted computing base of vTPM and VM alone does not sufficiently address VM-vTPM association as well.

4.2.9 Hardware assisted vTPMs. A native solution would to integrate vTPM functionality as part of the machine architecture and machine virtualization support. Such support would effectively extend a physical TPM, often implemented as part of firmware, to support have native support for virtual machines. Such a solution would effectively provide a TPM with a shared context for the entire machine and an individual context for each VM to capture state information. In such a case, the virtualization system would only be entrusted with the setup of virtual machines and would delegate

vTPM allocation and association to the hardware.

Pros: This would allow us to set up VMs in the hypervisor and with added hardware support we could use a vTPM with a trust model similar to the real TPM. The physical platform alone would be responsible for pointing vTPMs to the relevant TPM PCRs when establishing trust in a virtualization system.

Cons: Requires by far the most significant changes in hardware and implementation. The TPM itself is not designed to context switch. With TPM 2.0 some of this could be emulated but the context switching alone would cause severe performance issues with currently available TPM devices.

5 CONCLUSION AND FUTURE WORK

Creating a "trusted" virtualized platform is necessary for a variety of secure cloud computing scenarios. Ideally, we should be able to use secure hardware such as the TPM to bootstrap trust in a virtualization system and utilize a virtual TPM as a root of trust for a virtual machine. Our formal model reveals that the process of extending trust from the TPM all the way up into a VM is vulnerable to a *Goldeneye* attack. The *Goldeneye* attack emphasizes the importance of a trusted association between VM and vTPM. The association between a system and its root of trust is typically not considered on a real machine but we emphasize that it needs special considerations in case of a VM and and a vTPM. We demonstrated attack vectors in recent papers and proposed solutions which could be implemented today. Currently, we rely on assuming an invariant about the association between VMs and vTPMs. We suggest and discuss several solutions including constraining interfaces, changes to measurement architectures, and vTPM architectures. A clean solution might be to provide architectural support for roots of trust for virtual machines. In future work, we will discuss cryptographic approaches and explore some these options in depth. The combination of bigraphs with a trust model was useful in modelling trusted virtual platforms structurally. We expect interesting results when we add dynamics and transitions to our trust model in the future.

REFERENCES

[1] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang. 2009. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *2009 Annual Computer Security Applications Conference*. 461–470. https://doi.org/10.1109/ACSAC.2009.50

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 164–177. https://doi.org/10.1145/945445.945462

[3] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. 2006. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15 (USENIX-SS'06)*. USENIX Association, Berkeley, CA, USA, Article 21. http://dl.acm.org/citation.cfm?id=1267336.1267357

[4] Ernie Brickell, Jan Camenisch, and Liqun Chen. 2004. Direct Anonymous Attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. ACM, New York, NY, USA, 132–145. https://doi.org/10.1145/1030083.1030103

[5] Giovanni Conforti, Damiano Macedonio, and Vladimiro Sassone. 2005. Spatial Logics for Bigraphs. In *Automata, Languages and Programming*, Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 766–778.

[6] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[7] Boris Danev, Ramya Jayaram Masti, Ghassan O. Karame, and Srdjan Capkun. 2011. Enabling Secure VM-vTPM Migration in Private Clouds. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/2076732.2076759

[8] Trusted Computing Group. 2011. *Virtualized Trusted Platform Architecture Specification*. Trusted Computing Group. Rev. 1.26.

[9] Trusted Computing Group. 2014. *TCG EK Credential Profile*. Accessed: 2019-05-01.

[10] ISO. 2015. *Trusted Platform Module Library*. ISO ISO/IEC 11889-1:2015. International Organization for Standardization, Geneva, Switzerland.

[11] ISO. 2018. ISO/IEC NP 27070 Information Technology – Security Techniques – Security requirements for establishing virtualized roots of trust. https://www.iso.org/standard/56571.html. Accessed: 2018-10-31.

[12] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv e-prints*, Article arXiv:1902.03383 (Feb 2019), arXiv:1902.03383 pages. arXiv:cs.OS/1902.03383

[13] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[14] H. Lauer. 2019. A Logic of Secure Stratified Systems and its Application to Containerized Systems. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications*. 1–8. https://doi.org/tba

[15] H. Lauer and N. Kuntze. 2016. Hypervisor-Based Attestation of Virtual Environments. In *Advanced and Trusted Computing (ATC), 2016 Intl IEEE Conferences*. IEEE, 333–340.

[16] Andrew Martin. 2008. The ten-page introduction to Trusted Computing.

[17] Robin Milner. 2009. *The Space and Motion of Communicating Agents* (1st ed.). Cambridge University Press, New York, NY, USA.

[18] Bryan Parno. 2008. Bootstrapping Trust in a "Trusted" Platform. In *Proceedings of the 3rd Conference on Hot Topics in Security (HOTSEC'08)*. USENIX Association, Berkeley, CA, USA, Article 9, 6 pages. http://dl.acm.org/citation.cfm?id=1496671.1496680

[19] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '19)*. ACM, New York, NY, USA, 7–13. https://doi.org/10.1145/3317550.3321445

[20] Andre Rein. 2017. DRIVE: Dynamic Runtime Integrity Verification and Evaluation. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. ACM, New York, NY, USA, 728–742. https://doi.org/10.1145/3052973.3052975

[21] Mark D. Ryan. 2013. Cloud Computing Security. *J. Syst. Softw.* 86, 9 (Sept. 2013), 2263–2268. https://doi.org/10.1016/j.jss.2012.12.025

[22] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=1251375.1251391

[23] Nabil Schear, Patrick T. Cable, II, Thomas M. Moyer, Bryan Richard, and Robert Rudd. 2016. Bootstrapping and Maintaining Trust in the Cloud. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications (ACSAC '16)*. ACM, New York, NY, USA, 65–77. https://doi.org/10.1145/2991079.2991104

[24] Juhyung Son, Sungmin Koo, Jongmoo Choi, Seong-je Choi, Seungjae Baek, Gwangil Jeon, Jun-Hyeok Park, and Hyoungchun Kim. 2017. Quantitative Analysis of Measurement Overhead for Integrity Verification. In *Proceedings of the Symposium on Applied Computing (SAC '17)*. ACM, New York, NY, USA, 1528–1533. https://doi.org/10.1145/3019612.3019738