Some Ideas on Virtualized System Security, and Monitors

Hedi Benzina[1]* and Jean Goubault-Larrecq[1]

[1] LSV, ENS Cachan, CNRS, INRIA
61 avenue du Président Wilson, 94230 CACHAN, France
{benzina, goubault}@lsv.ens-cachan.fr

Abstract. Virtualized systems such as Xen, VirtualBox, VMWare or QEmu have been proposed to increase the level of security achievable on personal computers. On the other hand, such virtualized systems are now targets for attacks. We propose an intrusion detection architecture for virtualized systems, and discuss some of the security issues that arise. We argue that a weak spot of such systems is domain zero administration, which is left entirely under the administrator's responsibility, and is in particular vulnerable to trojans. To avert some of the risks, we propose to install a role-based access control model with possible role delegation, and to describe all undesired activity flows through simple temporal formulas. We show how the latter are compiled into Orchids rules, via a fragment of linear temporal logic, through a generalization of the so-called history variable mechanism.

1 Introduction

Intrusion detection and prevention systems (IDS, IPS) have been around for some time, but require some administration. Misuse intrusion detection systems require frequent updates of their attack base, while anomaly-based intrusion detection systems, and especially those based on statistical means, tend to have a high false positive rate. On the other hand, personal computers are meant to be used by a single individual, or by several individuals, none being specially competent about security, or even knowledgeable in computer science.

We propose to let the user run her usual environment inside the virtual machine of some virtualized architecture, such as Xen [28], VirtualBox [24], VMWare [25], or QEmu [15]. The idea has certainly been in the air for some time, but does not seem to have been implemented in the form we propose until now. We shall argue that this architecture has some of the advantages of decentralized supervision, where the IDS/IPS is located on a remote machine, connected to the monitored host through network links.

* This work was supported by grant DIGITEO N^{o}2009-41D from Région Ile-de-France, and by project PFC ("plateforme de confiance"), pôle de compétitivité System@tic Paris-région Ile-de-France.

Outline. We survey related work in Section 2. Section 3 describes the rationale for our virtualized supervision architecture, and the architecture itself. We identify what we think is the weakest link in this architecture in Section 4, i.e., the need for domain zero supervision. There does not seem to be any silver bullet here. We attempt to explore a possible answer to the problem in Section 5, based on Sekar *et al.*'s *model-carrying code* paradigm. We conclude in Section 6.

2 Related work

Much work has been done on enhancing the security of computer systems. Most implemented, host-based IDS run a program for security on the same operating system (OS) as protected programs and potential malware. This may be simply necessary, as with Janus [6], Systrace [13], Sekar *et al.*'s finite-state automaton learning system [19], or Piga-IDS [1], where the IDS must intercept and check each system call before execution. Call this an *interception* architecture: each system call is first checked for conformance against a security policy by the IDS; if the call is validated, then it is executed, otherwise the IDS forces the call to return immediately with an error code, without executing the call.

On the other hand, some other IDS are meant to work in a *decentralized* setting. In this case, the IDS does not run on the same host as the supervised host, S. While in an interception architecture, the IDS would run as a process on S itself, in a decentralized setting only a small so-called *sensor* running on S collects relevant events on S and sends them through some network link to the IDS, which runs on another, dedicated host M.

Both approaches have pros and cons. Interception architectures allow the IDS to counter any attack just before they are completed. This way, and assuming the security policy that the IDS enforces is sufficiently complete, no attack ever succeeds on S that would make reveal or alter sensitive data, make it unstable, or leave a backdoor open (by which we also include trojans and bots).

On the other hand, decentralized architectures (see Figure 1) make the IDS more resistant to attacks on S (which may be any of S_1, ..., S_4 in the figure): to kill the IDS, one would have to attack the supervision machine M, but M is meant to only execute the IDS and no other application, and has only limited network connectivity. In addition to the link from S to M used to report events to the IDS, we also usually have a (secure) link from M to S, allowing the IDS to issue commands to retaliate to attacks on S. While this may take time (e.g., some tens or hundreds of milliseconds on a LAN), this sometimes has the advantage to let the IDS *learn* about intruder behavior once they have completed an attack. This is important for forensics.

Decentralized architectures are also not limited to supervising just one host S. It is particularly interesting to let the supervision machine M collect events from several different hosts at once, from network equipment (routers, hubs, etc., typically through logs or MIB SNMP calls), and correlate between them, turning the IDS into a mix between host-based and network-based IDS.

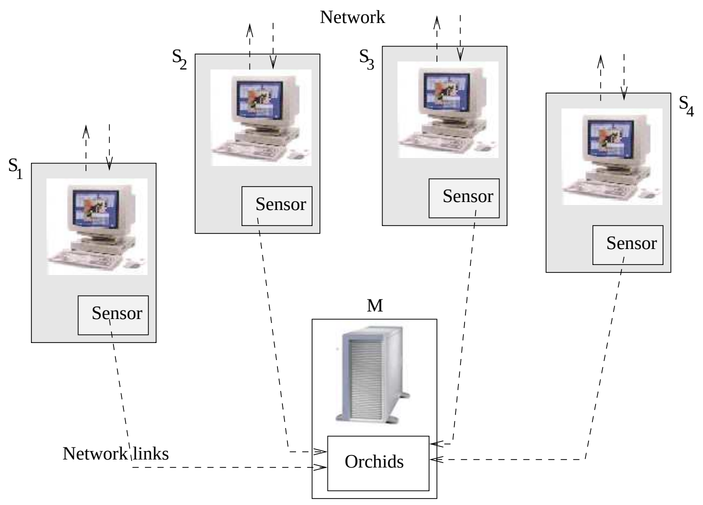

Fig. 1. Decentralized Supervision

We shall argue in Section 3 that one can simulate such a decentralized architecture, at minimal cost, on a single machine, using modern virtualization technology. We shall also see that this has some additional advantages.

A *virtualized* system such as Xen [28], VirtualBox [24], VMWare [25], or QEmu [15] allows one to emulate one or several so-called *guest* operating systems (OS) in one or several *virtual machines* (VM). The different VMs execute as though they were physically distinct machines, and can communicate through ordinary network connections (possibly emulated in software). The various VMs run under the control of a so-called *virtual machine monitor* (VMM) or *hypervisor*, which one can think of as being a thin, highly-privileged layer between the hardware and the VMs. See Figure 2, which is perhaps more typical of Xen than of the other cited hypervisors. The solid arrows are meant to represent the flow of control during system calls. When a guest OS makes a system call, its hardware layer is emulated through calls to the hypervisor. The hypervisor then calls the actual hardware drivers (or emulations thereof) implemented in a specific, high privilege VM called *domain zero*. Domain zero is the only VM to have access to the actual hardware, but is also responsible for administering the other VMs, in particular killing VMs, creating new VMs, or changing the emulated hardware interface presented to the VMs.

In recent years, virtualization has been seen by several as an opportunity for enforcing better security. The fact that two distinct VMs indeed run in separate sandboxes was indeed brought forward as an argument in this direction. However, one should realize that there is no conceptual distinction, from the point of view of protection, between having a high privilege VMM and lower-privileged VMs, and using a standard Unix operating system with a high privilege kernel and lower-privileged processes. Local-to-root exploits on Unix are bound to be

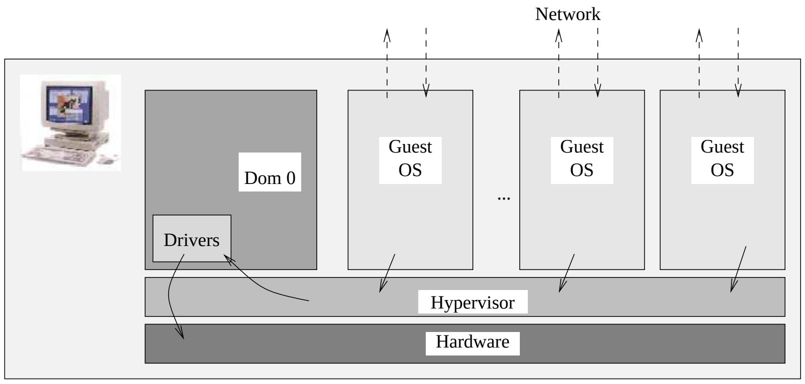

Fig. 2. A Virtualized System

imitated in the form of attacks that would allow one process running in one VM to gain control of the full VMM, in particular of the full hardware abstraction layer presented to the VMs. Indeed, this is exactly what has started to appear, with Wojtczuk's attack notably [26].

Some of the recent security solutions using virtualization are sHype [18] and NetTop [10]. They provide infrastructure for controlling information flows and resource sharing between VMs. While the granularity level in these systems is a VM, our system controls execution at the granularity of a process.

Livewire [5] is an intrusion detection system that controls the behavior of a VM from the outside of the VM. Livewire uses knowledge of the guest OS to understand the behavior in a monitored VM. Livewire's VMM intercepts only write accesses to a non-writable memory area and accesses to network devices. On the other hand, our architecture can intercept and control all system calls invoked in target VMs.

In [12], Onoue *et al.* propose a security system that controls the execution of processes from the outside of VMs. It consists of a modified VMM and a program running in a trusted VM. The system intercepts system calls invoked in a monitored VM and controls the execution according to a security policy. Thus, this is a an interception system. To fill the semantic gap between low-level events and high-level behavior, the system uses knowledge of the structure of a given operating system kernel. The user creates this knowledge with a tool when recompiling the OS. In contrast, we do not need to rebuild the OS, and only need to rely on standard event-reporting daemons such as `auditd`, which comes with SELinux [22], but is an otherwise independent component.

We end this section by discussing run-time supervision and enforcement of security policies. Systems such as SELinux (op. cit.) are based on a security policy, but fail to recognize illegal *sequences* of legal actions. To give a simple example, it may be perfectly legal for user A to copy some private data D to some public directory such as `/tmp`, and for user B to read any data from `/tmp`, although our security policy forbids any (direct) flow of sensitive data from A to

B. Such sequences of actions are called *transitive flows* of data in the literature. To our knowledge, Zimmerman *et al.* [29–31] were the first to propose an IDS that is able to check for illegal transitive flows. Briffaut [1] shows that even more general policies can be efficiently enforced, including non-reachability properties and Chinese Wall policies, among others; in general, Briffaut uses a simple and general policy language. We propose another, perhaps more principled, language in Section 5, based on linear temporal logic (LTL). Using the latter is naturally related to a more ancient proposal by Roger and the second author [17]. However, LTL as defined in (the first part of) the latter paper only uses future operators, and is arguably ill-suited to intrusion detection (as discussed in op. cit. already). Here, instead we use a fragment of LTL *with past*, which, although equivalent to ordinary LTL with only future operators as far as satisfiability is concerned (for some fixed initial state only, and up to an exponential-size blowup), will turn out to be much more convenient to specify policies, and easy to compile to rules that can be fed to the Orchids IPS [11, 7].

3 Simulating Decentralized Supervision, Locally

We run a fast, modern IPS such as Orchids [11, 7] in another VM to monitor, and react against, security breaches that may happen on the users' environment in each of the guest OSes present in a virtualized system: see Figure 3.

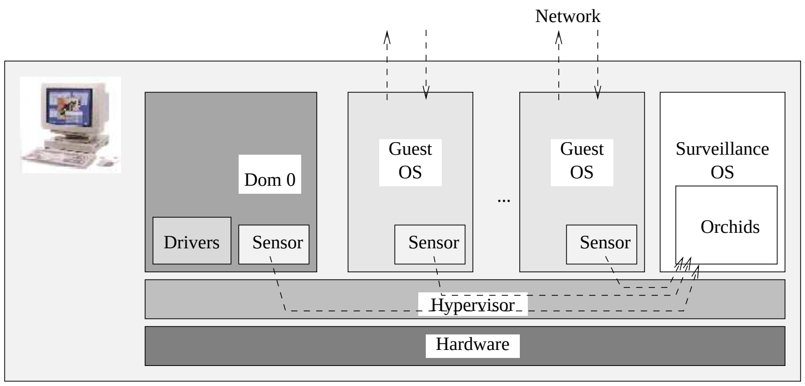

Fig. 3. Simulating Decentralized Supervision, Locally

One can see this architecture as an implementation of a decentralized supervision architecture on a single physical host.

We argue that this solution has several advantages. First, there is a clear advantage in terms of cost, compared to the decentralized architecture: we save the cost of the additional supervision host M.

Second, compared to a standard, unsupervised OS, the user does not need to change her usual environment, or to install any new security package. Only a small sensor has to run on her virtual machine to report events to Orchids. Orchids accepts events from a variety of sensors. In our current implementation, each guest OS reports sequences of system calls through the standard `auditd` daemon, a component of SELinux [22], which one can even run without the need for installing or running SELinux itself. (Earlier, we used Snare, however this now seems obsolescent.) The bulk of the supervision effort is effected in a different VM, thus reducing the installation effort to editing a few configuration files, to describe the connections between the guest OSes and the supervision OS mainly. In particular, we do not need to recompile any OS kernel with our architecture, except possibly to make sure that `auditd` is installed and activated.

A third advantage, compared with interception architectures, and which we naturally share with decentralized architectures, is that isolating the IPS in its own VM makes it resistant to attacks from the outside. Indeed, Orchids runs in a VM that has no other network connection to the outside world than those it requires to monitor the guest OSes, and which runs no other application that could possibly introduce local vulnerabilities.

Orchids should have high privileges to be able to retaliate to attacks on each guest OS. For example, we use `ssh` connections between Orchids and each VM kernel to be able to kill offending processes or disable offending user accounts. (The necessary local network links, running in the opposite direction as the sensor-to-Orchids event reporting links shown in Figure 3, are not drawn.)

However, running on a virtualized architecture offers additional benefits. One of them is that Orchids can now ask domain zero to *kill* an entire VM. This is necessary when a guest OS has been subject to an attack with consequences that we cannot assess precisely. For example, the `do_brk()` attack [23] and its siblings, or the `vmsplice()` attack [14] allow the attacker not just to gain root access, but direct access to the *kernel*. Note that this means, for example, that the attacker has immediate access to the whole process table, as well as to the memory zones of all the processes. While current exploits seem not to have used this opportunity, such attack vectors in principle allow an attacker to become completely stealthy, e.g., by making its own processes invisible to the OS. In this case, the OS is essentially in an unpredictable state.

The important point is that we can always revert any guest OS to a previous, safe state, using virtualization. Indeed, each VM can be *checkpointed*, i.e., one can save the complete instantaneous state of a given VM on disk, including processes, network connections, signals. Assuming that we checkpoint each VM at regular intervals, it is then feasible to have Orchids retaliate by killing a VM in extreme cases and replacing it by an earlier, safe checkpoint.

Orchids can also detect VMs that have died because of fast denial-of-service attacks (e.g., the double `listen()` attack [3], which causes instant kernel lock-up), by pinging each VM at regular intervals: in this case, too, Orchids can kill the VM and reinstall a previous checkpoint. We react similarly to attacks on guest OSes that are suspected of having succeeded in getting kernel privileges and of, say, disabling the local `auditd` daemon.

Killing VMs and restoring checkpoints is clearly something that we cannot afford with physical hosts instead of VMs.

It would be tempting to allow Orchids to run *inside* domain zero to do so. Instead, we run Orchids in a separate guest OS, with another `ssh` connection to issue VM administration commands to be executed by a shell in domain zero. I.e., we make domain zero *delegate* surveillance to a separate VM running Orchids, while the latter trusts domain zero to administer the other guest VMs. We do so in order to sandbox each from the other one. Although we have taken precautions against this prospect, there is still a possibility that a wily attacker would manage to cause denial-of-service attacks on Orchids by crafting events causing blow-up in the internal Orchids surveillance algorithm (see [7]), and we don't want this to cause the collapse of the whole host. Conversely, if domain zero itself is under attack, we would like Orchids to be able to detect this and react against it. We discuss this in Section 4

To our knowledge, this simple architecture has not been put forward in previous publications, although some proposals already consider managing the security of virtualized architectures, as we have already discussed in Section 2.

4 The Weak Link: Domain Zero Administration

The critical spots in our architecture are the VMM (hypervisor) itself, domain zero, and the surveillance VM running Orchids.

Attacking the latter is a nuisance, but is not so much of a problem as attacking the VMM or domain zero, which would lead to complete subversion of the system. Moreover, the fact that Orchids runs in an isolated VM averts most of the effects of any vulnerability that Orchids may have.

Attacks against the VMM are much more devastating. Wojtczuk's 2008 attacks on Xen 2 [26] allow one to take control of the VMM, hence of the whole machine, by rewriting arbitrary code and data using DMA channels, and almost without the processor's intervention... quite a fantastic technique, and certainly one that breaks the common idea that every change in stored code or data must be effected by some program running on one of the processors. Indeed, here a separate, standard chip is actually used to rewrite the code and data.

Once an attacker has taken control over the VMM, one cannot hope to react in any effective way. In particular, the VMM controls entirely the hardware abstraction layer that is presented to each of the guest OSes: no network link, no disk storage facility, no keyboard input can be trusted by any guest OS any longer. Worse, the VMM also controls some of the features of the processor itself, or of the MMU, making memory or register contents themselves unreliable.

We currently have no idea how to prevent attacks such as Wojtczuk's, apart from unsatisfactory, temporary remedies such as checkpointing some selected memory areas in the VMM code and data areas.

However, we claim that averting such attacks is best done by making sure that they cannot be run at all. Indeed, Wojtczuk's attacks only work provided the attacker already has *root access* to domain zero, and this is already quite a predicament. We therefore concentrate on ensuring that no unauthorized user can gain root access to domain zero.

Normally, only the system administrator should have access to domain zero. (In enterprise circles, the administrator may be a person with the specific role of an administrator. In family circles, we may either decide that there should be no administrator, and that the system should self-administer; or that one particular user has administrator responsibilities.) We shall assume that this administrator is *benevolent*, i.e., will not consciously run exploits against the system. However, he may do so without knowing it.

One possible scenario is this: the administrator needs to upgrade some library or driver, and downloads a new version; this version contains a trojan horse, which runs Wojtczuk's attack. Modern automatic update mechanisms use cryptographic mechanisms, including certificates and cryptographic hashing mechanisms [27], to prevent attackers from running man-in-the-middle attacks, say, and substitute a driver with a trojan horse for a valid driver. However, there is no guarantee that the authentic driver served by the automatic update server is itself free of trojans. There is at least one actual known case of a manufacturer shipping a trojan (hopefully by mistake): some Video iPods were shipped with the Windows `RavMonE.exe` virus [16], causing immediate infection of any Windows host to which the iPods were connected.

5 A Line of Defense: MCC, LTL, and Orchids

Consider the case whereby we have just downloaded a device driver, and we would like to check whether it is free of a trojan. Necula and Lee pioneered the idea of shipping a device driver with a small, formal proof of its properties, and checking whether this proof is correct before installing and running the driver. This is *proof-carrying code* [9]. More suited to security, and somehow more practical is Sekar *et al.*'s idea of *model-carrying code* (MCC) [20]. Both techniques allow one to accept and execute code even from untrusted producers.

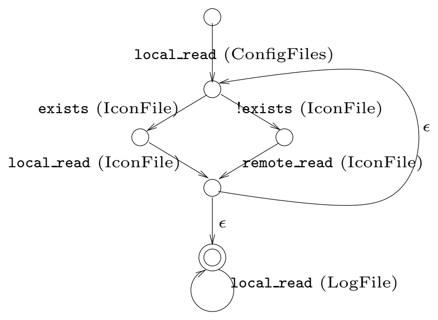

Fig. 4. An EFSA Model, after Sekar *et al.* [20]

Let us review MCC quickly. A *producer* generates both the program D to be downloaded (e.g., the device driver), and a *model* of it, M. The *consumer*, which in our case is software running in domain zero, checks the model against a local *policy* P. Instead of merely rejecting the program D if its model M does not satisfy, the consumer computes an *enforcement model* M' that satisfies both M and P, and generates a monitor that checks whether P satisfies M' at run-time. Any violation is flagged and reported.

In [20], models, as well as policies and enforcement models, are taken to be extended finite-state automata (EFSA), i.e., finite state automata augmented with finitely many state variables meant to hold values over some fixed domain. A typical example, taken from op. cit., is the EFSA of Figure 4. This is meant to describe the normal executions of D as doing a series of system calls as follows. Each system call is abstracted, e.g., the first expected system call from D is a call to `local_read` with argument bound to the ConfigFiles state variable. Then D is expected to either take the left or the right transition going down, depending on whether some tested file (in variable IconFile) exists or not. In the first case, D should call `local_read`, in the second case, D should call `remote_read`. The transitions labeled ϵ are meant to be firable spontaneously.

Typical policies considered by Sekar *et al.* are invariants of the form "any program should either access local files or access the network but not both" (this is violated above, assuming obvious meanings for system calls), or that only files from the `/var/log/httpd/` directory should be read by D. Policies are again expressed as EFSA, and enforcement models can be computed as a form of synchronized product denoting the intersection of the languages of M and P.

The EFSA model is sufficiently close to the automaton-based model used in Orchids (which appeared about at the same time, see the second part of [17]; or see [7] for a more in-depth treatment) that the EFSA built in the MCC approach can be fed almost directly to Orchids. Accordingly, we equip domain zero with a sensor as well (Figure 3, left box), which reports to Orchids for EFSA checking.

However, we feel that a higher-level language, allowing one to write acceptable policies for automatic updates in a concise and readable manner, would be a plus. There have been many proposals of higher-level languages already, including linear temporal logic (LTL) [17], chronicles [8], or the BMSL language by Sekar and Uppuluri [21], later improved upon by Brown and Ryan [2]. It is not our purpose to introduce yet another language here, but to notice that a simple variant of LTL *with past* will serve our purpose well and is efficiently and straightforwardly translated to Orchids rules—which we equate with EFSA here, for readability, glossing over inessential details.

Consider the following fragment of LTL with past. We split the formulae in several sorts. $F^\bullet$ will always denote *present tense* formulae, which one can evaluate by just looking at the current event:

$$\begin{array}{rll} F^\bullet ::= & P(\boldsymbol{x}) \mid cond(\boldsymbol{x}) & \text{atomic formula} \\ \mid & \bot & \text{false} \\ \mid & F^\bullet \wedge F^\bullet & \text{conjunction} \\ \mid & F^\bullet \vee F^\bullet & \text{disjunction} \end{array}$$

Atomic formulae check for specific occurrences of events, e.g., `local_read` (IconFile) will typically match the current event provided it is of the form `local_read` applied to some argument, which is bound to the state variable IconFile. In the above syntax, $\boldsymbol{x}$ denotes a list of state variables, while $cond(\boldsymbol{x})$ denotes any computable formula of $\boldsymbol{x}$, e.g., to check that IconFile is a file in some specific set of allowed directories. This is as in [21, 2]. We abbreviate $P(\boldsymbol{x}) \mid \top$, where $\top$ is some formula denoting true, as $P(\boldsymbol{x})$.

Note that we do not allow for negations in present tense formulae. If needed, we allow certain negations of atomic formulae as atomic formulae themselves, e.g., `!exists` (IconFile). However, we believe that even this should not be necessary. Disjunctions were missing in [21], and were added in [2].

Next, we define *past tense* formulae, which can be evaluated by looking at the current event and all past events, but none of the events to come. Denote past tense formulae by $F^{\leftarrow}$:

$$\begin{array}{rll} F^{\leftarrow} ::= & F^{\bullet} & \text{present tense formulae} \\ \mid & F^{\leftarrow} \wedge F^{\leftarrow} & \text{conjunction} \\ \mid & F^{\leftarrow} \vee F^{\leftarrow} & \text{disjunction} \\ \mid & F^{\leftarrow} \searrow F^{\bullet} & \text{without} \\ \mid & \text{Start} & \text{initial state} \end{array}$$

All present formulae are (trivial) past formulae, and past formulae can also be combined using conjunction and disjunction. The novelty is the "without" constructor: $F^{\leftarrow} \searrow F^{\bullet}$ holds iff $F^{\leftarrow}$ held at some point in the past, and since then, $F^{\bullet}$ never happened. Apart from the without operator, the semantics of our logic is standard. We shall see below that it allows us to encode a number of useful idioms. The past tense formula Start will also be explained below.

Formally, present tense formulae $F^{\bullet}$ are evaluated on a current event e, while past tense formulae $F^{\leftarrow}$ are evaluated on a stream of events $\boldsymbol{e} = e_1, e_2, \ldots, e_n$, where the current event is e_n, and all others are the past events. (We warn the reader that the semantics is meant to reason logically on the formulae, but is not indicative of the way they are evaluated in practice. In particular, although we are considering past tense formulae, and their semantics refer to past events, our algorithm will never need to read back past events.) The semantics of the without operator is that $\boldsymbol{e} = e_1, e_2, \ldots, e_n$ satisfies $F^{\leftarrow} \searrow F^{\bullet}$ if and only if there is an integer m, with $0 \leq m < n$, such that the proper prefix of events $e_1, e_2, \ldots, e_m$ satisfies $F^{\leftarrow}$ for some values of the variables that occur in $F^{\leftarrow}$ ("$F^{\leftarrow}$ held at some point in the past"), and none of $e_{m+1}, \ldots, e_n$ satisfies $F^{\bullet}$ ("since then, $F^{\bullet}$ never happened")—precisely, none of $e_{m+1}, \ldots, e_n$ satisfies $F^{\bullet}$ *with the values of the variables obtained* so as to satisfy $F^{\leftarrow}$; this makes perfect sense if all the variables that occur in $F^{\leftarrow}$ already occur in $F^{\bullet}$, something we shall now assume.

The past tense formula Start has trivial semantics: it only holds on the empty sequence of events (i.e., when $n = 0$), i.e., it only holds when we have not received any event yet. This is not meant to have any practical use, except to be able to encode useful idioms with only a limited supply of temporal operators. For example, one can define the formula $\blacksquare\neg F^{\bullet}$ ("$F^{\bullet}$ never happened in the past") as Start $\searrow F^{\bullet}$.

$$\begin{array}{rcll}
\blacksquare\neg F^\bullet & \overset{\text{def}}{=} & \text{Start} \setminus F^\bullet & \text{“}F^\bullet \text{ never happened in the past”} \\
\blacklozenge F^\leftarrow & \overset{\text{def}}{=} & F^\leftarrow \setminus \bot & \text{“}F^\leftarrow \text{ was once true in the past”} \\
F^\leftarrow \to F^\bullet & \overset{\text{def}}{=} & \blacklozenge F^\leftarrow \wedge F^\bullet & \text{“}F^\leftarrow \text{ was once true, and now } F^\bullet \text{ is”} \\
F^\leftarrow \to F_1^\bullet \to F_2^\bullet \to \ldots \to F_n^\bullet & \overset{\text{def}}{=} & (\ldots((F^\leftarrow \to F_1^\bullet) \to F_2^\bullet) \to \ldots) \to F_n^\bullet & \\
F_1^\bullet; F_2^\bullet; \ldots; F_n^\bullet & \overset{\text{def}}{=} & \text{Start} \to F_1^\bullet \to \ldots \to F_n^\bullet \text{ Chronicle} &
\end{array}$$

Fig. 5. Some Useful Idioms

The without operator allows one to encode other past temporal modalities, see Figure 5. In particular, we retrieve the *chronicle* $F_1^\bullet; F_2^\bullet; \ldots; F_n^\bullet$ [8], meaning that events matching $F_1^\bullet$, then $F_2^\bullet$, ..., then $F_n^\bullet$ have occurred in this order before, not necessarily in a consecutive fashion. More complex sequences can be expressed. Notably, one can also express disjunctions as in [2], e.g., disjunctions of chronicles, or formulae such as $(\texttt{login}(Uid) \setminus \texttt{logout}(Uid)) \wedge \texttt{local\_read}(Uid, ConfigFile)$ to state that user Uid logged in, then read some $ConfigFile$ locally, without logging out inbetween.

Let us turn to more practical details. First, we do not claim that only Start and the without ($\setminus$) operator should be used. The actual language will include syntactic sugar for chronicles, box ($\blacksquare$) and diamond ($\blacklozenge$) modalities, and possibly others, representing common patterns. The classical past tense LTL modality $\mathcal{S}$ (“since”) is also definable, assuming negation, by $F \; \mathcal{S} \; G = G \setminus \neg F$, but seems less interesting in a security context.

Second, as already explained in [17, 21, 2], we see each event e as a formula $P(fld_1, fld_2, \ldots, fld_m)$, where $fld_1, fld_2, \ldots, fld_m$ are taken from some domain of values—typically strings, or integers, or time values. This is an abstraction meant to simplify mathematical description. For example, using `auditd` as event collection mechanism, we get events in the form of strings such as:

```
1276848926.326:1234 syscall=102 success=yes a0=2 a1=1 a2=6 pid=7651
```

which read as follows: the event was collected at date 1276848926.326, written as the number of seconds since the epoch (January 01, 1970, 0h00 UTC), and is event number 1234 (i.e., we are looking at event e_{1234} is our notation); this was a call to the `socket()` function (code 102), with parameters `PF_INET` (Internet domain, where `PF_INET` is defined as 2 in `/usr/include/socket.h`—`a0` is the first parameter to the system call), `SOCK_STREAM` (= 1; `a1` is connection type here), and with the TCP protocol (number 6, passed as third argument `a2`); this was issued by process number 7651 and returned with success. Additional fieldss that are not relevant to the example are not shown. This event will be understood in our formalization as event e_{1234}, denoting `syscall` $(1276848926.326, 102, \texttt{"yes"}, 2, 1, 6, 7651)$. The event e_{1234} satisfies the atomic formula `syscall` $(Time, Call, Res, Dom, Conn, Prot, Pid) \mid Res = \texttt{"yes"}$ but neither `audit` (X) nor `syscall` $(Time, Call, Res, Dom, Conn, Prot, Pid) \mid Time \leq 1276848925$.

Third, we need to explain how we can detect when a given sequence of events $\boldsymbol{e}$ satisfies a given formula in our logic, algorithmically. To this end, we define a translation to the Orchids language, or to EFSA, and rely on Orchids' extremely efficient model-checking engine [7]. The translation is based on the idea of *history variables*, an old idea in model-checking safety properties in propositional LTL. Our LTL is *not* propositional, as atomic formulae contain free variables—one may think of our LTL as being first-order, with an implicit outer layer of existential quantifiers on all variables that occur—but a similar technique works.

It is easier to define the translation for an extended language, where the construction $F^{\leftarrow} \setminus F^{\bullet}$ is supplemented with a new construction $F^{\leftarrow} \setminus^{*} F^{\bullet}$ (*weak without*), which is meant to hold iff $F^{\leftarrow}$ once held in the past, *or holds now*, and $F^{\bullet}$ did not become true afterwards.

The *subformulae* of a formula F are defined as usual, as consisting of F plus all subformulae of its immediate subformulae. To avoid some technical subtleties, we shall assume that Start is also considered a subformula of any past tense formula. The *immediate subformulae* of $F \wedge G$, $F \vee G$, $F \setminus^{*} G$ are F and G, while atomic formulae, $\perp$ and Start don't have any immediate subformula. To make the description of the algorithm smoother, we shall assume that the immediate subformulae of $F \setminus G$ are not F and G, but rather $F \setminus^{*} G$ and G. Indeed, we are reproducing a form of Fischer-Ladner closure here [4].

Given a fixed past-tense formula $F^{\leftarrow}$, we build an EFSA that monitors exactly when a sequence of events will satisfy $F^{\leftarrow}$. To make the description of the algorithm simpler, we shall assume a slight extension of Sekar *et al.*'s EFSA where state variables can be assigned values on traversing a transition. Accordingly, we label the EFSA transitions with a sequence of *actions* $\$x_1 := e_1; \$x_2 := e_2; \ldots; \$x_k := e_k$, where $\$x_1$, $\$x_2$, ..., $\$x_k$ are state variables, and e_1, e_2, ..., e_k are expressions, which may depend on the state variables. This is actually possible in the Orchids rule language, although the view that is given of it in [7] does not mention it. Also, we will only need these state variables to have two values, 0 (false) or 1 (true), so it is in principle possible to dispense with all of them, encoding their values in the EFSA's finite control. (Instead of having three states, the resulting EFSA would then have $3\ 2^k$ states.)

Given a fixed $F^{\leftarrow}$, our EFSA has only three states q_{init} (the initial state), q, and q_{alert} (the final, acceptance state). We create state variables $\$x_i$, $1 \leq i \leq k$, one per subformula of $F^{\leftarrow}$. Let F_1, F_2, ..., F_k be these subformulae (present or past tense), and sort them so that any subformula of F_i occurs before F_i, i.e., as F_j for some $j < i$. (This is a well-known *topological sort*.) In particular, F_k is just $F^{\leftarrow}$ itself. Without loss of generality, let Start occur as F_1. The idea is that the EFSA will run along, monitoring incoming events, and updating $\$x_i$ for each i, in such a way that, at all times, $\$x_i$ equals 1 if the corresponding subformula F_i holds on the sequence $\boldsymbol{e}$ of events already seen, and equals 0 otherwise.

There is a single transition from q_{init} to q, which is triggered without having to read any event at all. This is an *ϵ-transition* in the sense of [7], and behaves similarly to the transitions `exists` (IconFile) and `!exists` (IconFile) of Figure 4.

It is labeled with the actions $\$x_1 := 1; \$x_2 := 0; \ldots ; \$x_k := 0$ (Start holds, but no other subformula is currently true).

There is also a single ϵ-transition from q to q_{alert}. This is labeled by no action at all, but is guarded by the condition $\$x_k == 1$. I.e., this transition can only be triggered if $\$x_k$ equals 1. By the discussion above, this will only ever happen when F_k, i.e., $F^{\leftarrow}$ becomes true.

Finally, there is a single (non-ϵ) transition from q to itself. Since it is not an ϵ-transition, it will only fire on reading a new event e [7]. It is labeled with the following actions, written in order of increasing values of i, $1 \leq i \leq k$:

$\$x_1 := 0$	(Start is no longer true)
$\$x_i := P(\boldsymbol{x}) \wedge \mathit{cond}(\boldsymbol{x})$	(for each i such that F_i is atomic, i.e., F_i is $P(\boldsymbol{x}) \mid \mathit{cond}(\boldsymbol{x})$)
$\$x_i := 0$	(if F_i is $\perp$)
$\$x_i := \mathit{and}(\$x_j, \$x_k)$	(if $F_i = F_j \wedge F_k$)
$\$x_i := \mathit{or}(\$x_j, \$x_k)$	(if $F_i = F_j \vee F_k$)
$\$x_i := \mathit{or}(\$x_j, \mathit{and}(\mathit{not}(\$x_k), \$x_i))$	(if $F_i = F_j \diagdown^* F_k$)
$\$x_i := \mathit{and}(\mathit{not}(\$x_k), \$x_\ell)$	(if $F_i = F_j \diagdown F_k$, and $F_j \diagdown^* F_k$ is F_ℓ, $\ell < i$)

Here, *and*, *or* and *not* are truth-table implementations of the familiar Boolean connectives, e.g., $\mathit{and}(0, 1)$ equals 0, while $\mathit{and}(1, 1)$ equals 1. We assume that $P(\boldsymbol{x})$, i.e., $P(x_1, \ldots, x_n)$ will equal 1 if the current event is of the form $P(s_1, \ldots, s_n)$, and provided each x_j that was already bound was bound to s_j exactly, in which case those variable x_j that were still unbound will be bound to the corresponding s_j. E.g., if x_1 is bound to 102 but x_2 is unbound, then $P(x_1, x_2)$ will equal 1 if the current event is $P(102, 6)$ (binding x_2 to 6), or $P(102, 7)$ (binding x_2 to 7), but will equal 0 if the current event is $Q(102, 6)$ for some $Q \neq P$, or $P(101, 6)$. We hope that this operational view of matching predicates is clearer than the formal view (which simply treats $x_1, \ldots, x_n$ as existentially quantified variables, whose values will be found as just described).

The interesting case is when F_i is a without formula $F_j \diagdown F_k$, or $F_j \diagdown^* F_k$. $F_j \diagdown F_k$ will become true after reading event e whenever $F_j \diagdown^* F_k$ was already true before reading it, and F_k is still false, i.e., when $\$x_\ell = 1$ and $\$x_k = 0$, where ℓ is the index such that $F_j \diagdown^* F_k$ occurs in the list of subformulae of $F^{\leftarrow}$ as F_ℓ. So in this case we should update $\$x_i$ to $\mathit{and}(\mathit{not}(\$x_k), \$x_\ell)$, as shown above. This relies on updating variables corresponding to weak without formulae $F_j \diagdown^* F_k$: $F_j \diagdown^* F_k$ becomes true after reading event e iff either F_j becomes true ($\$x_j = 1$), or $F_j \diagdown^* F_k$ was already true before ($\$x_i$ was already equal to 1) and F_k is false on event e ($\$x_k$ equals 0), whence the formula $\$x_i := \mathit{or}(\$x_j, \mathit{and}(\mathit{not}(\$x_k), \$x_i))$ in this case.

This completes the description of the translation. We now rely on Orchids' fast, real-time monitoring engine to alert us in case any policy violation, expressed in our fragment of LTL, is detected.

6 Conclusion

We have proposed a cost-effective implementation of a decentralized supervision architecture on a single host, using virtualization techniques. This offers at least the same security level as a classical decentralized intrusion detection architecture based on a modern IPS such as Orchids, and additionally allows for killing and restoring whole VMs to previous safe states.

In any security architecture, there is a weak link. We identify this weak link as domain zero administration, where even a benevolent administrator may introduce trojans, possibly leading to complete corruption, not just of the VMs, but of the VMM itself. There is no complete defense against this yet. As a first step, we have shown that Orchids can again be used, this time to implement an elegant instance of Sekar *et al.*'s model-carrying code paradigm (MCC).

References

1. J. Briffaut. *Formalisation et garantie de propriétés de sécurité système: Application à la détection dintrusions.* PhD thesis, LIFO Université d'Orléans, ENSI Bourges, Dec. 2007.
2. A. Brown and M. Ryan. Synthesising monitors from high-level policies for the safe execution of untrusted software. In *Information Security Practice and Experience*, pages 233–247. Springer Verlag LNCS 4991, 2008.
3. H. Dias. Linux kernel 'net/atm/proc.c' local denial of service vulnerability. BugTraq Id 32676, CVE-2008-5079, Dec. 2008.
4. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18:194–211, 1979.
5. T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed Systems Security Symposium*, San Diego, CA, Feb. 2003.
6. I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, July 1996.
7. J. Goubault-Larrecq and J. Olivain. A smell of Orchids. In M. Leucker, editor, *Proceedings of the 8th Workshop on Runtime Verification (RV'08)*, Lecture Notes in Computer Science, pages 1–20, Budapest, Hungary, Mar. 2008. Springer.
8. B. Morin and H. Debar. Correlation of intrusion symptoms: an application of chronicles. In *Proceedings of the 6th International Conference on Recent Advances in Intrusion Detection (RAID'03)*, pages 94–112, 2003.
9. G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. *SIGOPS Operating Systems Review*, 30:229–243, Oct. 1996.
10. Nettop, 2004. `http://www.nsa.gov/research/tech_transfer/fact_sheets/nettop.shtml`.
11. J. Olivain and J. Goubault-Larrecq. The Orchids intrusion detection tool. In K. Etessami and S. Rajamani, editors, *17th Intl. Conf. Computer Aided Verification (CAV'05)*, pages 286–290. Springer LNCS 3576, 2005.
12. K. Onoue, Y. Oyama, and A. Yonezawa. Control of system calls from outside of virtual machines. In R. L. Wainwright and H. Haddad, editors, *SAC*, pages 2116–1221. ACM, 2008.

13. N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, Aug. 2003.
14. W. Purczyński and qaaz. Linux kernel prior to 2.6.24.2 '`vmsplice_to_pipe()`' local privilege escalation vulnerability. http://www.securityfocus.com/bid/27801, Feb. 2008.
15. Qemu, 2010. `http://www.qemu.org/`.
16. Small number of video iPods shipped with Windows virus. `http://www.apple.com/support/windowsvirus/`, 2010.
17. M. Roger and J. Goubault-Larrecq. Log auditing through model checking. In *14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236. IEEE Comp. Soc. Press, 2001.
18. R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference*, Tucson, AZ, Dec. 2005.
19. R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.
20. R. Sekar, C. Ramakrishnan, I. Ramakrishnan, and S. Smolka. Model-carrying code (MCC): A new paradigm for mobile-code security. In *Proceedings of the New Security Paradigms Workshop (NSPW 2001)*, Cloudcroft, NM, Sept. 2001. ACM Press.
21. R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *SSYM 1999: Proceedings of the 8th conference on USENIX Security Symposium*, Berkeley, CA, 1999.
22. S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Technical report, NSA, 2001.
23. P. Starzetz. Linux kernel 2.4.22 `do_brk()` privilege escalation vulnerability. `http://www.k-otik.net/bugtraq/12.02.kernel.2422.php`, Dec. 2003. K-Otik ID 0446, CVE CAN-2003-0961.
24. Virtualbox, 2010. `http://www.virtualbox.org/`.
25. Vmware, 2010. `http://www.vmware.com/`.
26. R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat'08*, Las Vegas, NV, 2008.
27. [ms-wusp]: Windows update services: Client-server protocol specification. `http://msdn.microsoft.com/en-us/library/cc251937(PROT.13).aspx`, 2007–2010.
28. Xen, 2005–2010. `http://www.xen.org/`.
29. J. Zimmerman, L. Mé, and C. Bidan. Introducing reference flow control for detecting intrusion symptoms at the OS level. In *Proceeedings of the Recent Advances in Intrusion Detection Conference (RAID)*, pages 292–306, 2002.
30. J. Zimmerman, L. Mé, and C. Bidan. Experimenting with a policy-based hids based on an information flow control model. In *ACSAC '03: Proceedings of the 19th Annual Computer Security Applications Conference*, page 364, Washington, DC, USA, 2003. IEEE Computer Society.
31. J. Zimmerman, L. Mé, and C. Bidan. An improved reference flow control model for policy-based intrusion detection. In *Proceeedings of the European Symposium On Research in Computer Security (ESORICS)*, pages 291–308, 2003.