Electronic Notes in Theoretical Computer Science 70 No. 4 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 16 pages

Combining Monitors for Runtime System Verification

Joshua Levy and Hassen Saïdi and Tomás E. Uribe

System Design Laboratory
SRI International
Menlo Park, CA. 94025
{levy,saidi,uribe}@sdl.sri.com

Abstract

Runtime verification permits checking system properties that cannot be fully verified off-line. This is particularly true when the system includes complex third-party components, such as general-purpose operating systems and software libraries, and when the properties of interest include security and performance. The challenge is to find reliable ways to monitor these properties in realistic systems. In particular, it is important to have assurance that violations will be reported when they actually occur. For instance, a monitor may not detect a security violation if the violation results from a series of system events that are not in its model.

We describe how combining runtime monitors for diverse features such as memory management, security-related events, performance data, and higher-level temporal properties can result in more effective runtime verification. After discussing some basic notions for combining and relating monitors, we illustrate their application in an intrusion-tolerant Web server architecture under development at SRI.

1 Introduction

Most computer systems cannot be completely verified before they are put in service. The high complexity of complete fielded systems places them beyond the reach of exhaustive formal analysis tools, such as model checkers and theorem provers. Furthermore, many systems rely on third-party components, for which a complete specification (or even source code) is not available. Finally, the nature of most formal analysis tools requires that they check a model of the

[1] This research is sponsored by DARPA under contract number N66001-00-C-8058. The views herein are those of the authors and do not necessarily reflect the views of the supporting agency.

system rather than the system itself. If this model is inaccurate, important aspects of the actual system behavior can be overlooked. The specifications used to certify a system may be incomplete, wrong, or make implicit assumptions that are violated by the runtime environment.

Runtime verification addresses some of these problems by checking that the actual system execution satisfies desired properties. Rather than checking that all computations of (a model of) the system satisfy a property, as design-time verification does, runtime verification checks that the particular computation generated when the system is executed is correct, avoiding the state-space explosion that limits the scalability of exhaustive verification tools.

However, important choices must still be made when analyzing the runtime behavior of a system, including the level of abstraction at which the runtime verification is performed, how detailed and comprehensive it is, and what specifications or properties are expected to hold. For example, the comprehensiveness of the checks is often traded off against efficiency.

In many cases it is insufficient to monitor operation at a single level of abstraction. Likewise, monitoring may not be restricted to a single component, or to a single aspect of operation. The correct behavior of the monitoring mechanism may depend on properties that must themselves be monitored separately.

These concerns are particularly acute when the system is intended to maintain security properties. A hostile attacker can try to defeat or circumvent the monitoring mechanisms themselves, and a successful attack may disable the monitors intended to report it.

In this paper, we discuss and illustrate how runtime verification benefits from the combination of multiple monitors at multiple levels of abstraction:

- The use of many specialized monitors results in greater awareness of events in the system.
- By combining the information provided by the monitors, stronger properties that relate different aspects of the system can be checked at runtime.
- The modularity arising from separate monitors facilitates the sharing and reuse of monitoring information and mechanisms.

Outline: We present some preliminary definitions in Sections 2 and 3. Section 4 describes our target application, an intrusion-tolerant server. Section 5 presents a set of examples where monitors are combined. Section 6 presents conclusions, and related and future work.

2 Preliminaries

We are interested in checking runtime properties of distributed systems, which may be composed of multiple machines, each running multiple processes, connected by a network. While we do not attempt a full formalization of this task

in this paper, below we describe some of the aspects that it should address.

System state: Informally, the system state is given by the union of the states of all the machines, together with the state of the network.

For a particular machine, the system state can be described at various levels of abstraction. At the hardware level, it is given by the state of the processor, registers, and memory. At the OS level, it is the state of the OS and the processes currently running. For a given process or thread, the (local) state is given by the value of its program variables, execution stack, and control location. The *system variables* should be sufficient to describe all of the components and abstractions considered.

Definition 2.1 A *system state* is an assignment of values to a set of system variables $\mathcal{V}$.

Fixing the set of system variables is a modeling choice, and implies choosing a certain minimum level of abstraction (such as CPU registers and memory locations), but is useful in defining our terminology below.

Definition 2.2 A *run* r of the system is an infinite sequence of system states

$$r : s_0, s_1, s_2, \ldots$$

States and runs can be mapped to higher levels of abstraction, where an abstract variable is defined in terms of a set of original *concrete* system variables. This raises a number of issues, such as the problem of the scope of the abstract variables [15]: in practice, the definition of the abstract state will be local to a module, process, or subroutine. When the local process is not running, the abstract values will be undefined. In general, the mapping between abstract and concrete variables is not fixed (e.g., a program variable refers to different memory locations from one run of the program to the next, and other values can be dynamically bound).

Observability at different levels: Usually, when we focus on a particular aspect of the system, only a subset of the system state is visible. For example, an application might not know what the full OS state is, including what other processes are running. The OS may know about all the running processes, but may not have direct access to the internal state of a particular process.

Definition 2.3 (Local view) Given a system state s and a set of (possibly abstract) variables $\mathcal{L}$, we write $s|_{\mathcal{L}}$ to indicate the assignment that s induces in the variables in $\mathcal{L}$, that is, the local view of the state. We allow for variables in $\mathcal{L}$ to be undefined (e.g., at a state where they have not been initialized).

Time granularity: In general, a component or monitor can only observe some of the system transitions, and may overlook transitions. Also, a transition at the software abstraction level may correspond to multiple steps at the hardware level, and, furthermore, may be interrupted and resumed. For instance, an application-level monitor will not know when an interrupt is called.

For this reason, we can expect most of the properties being monitored to be *stuttering invariant* [1].

Previous work has identified distinctions between process-level, statement-level, and instruction-level monitoring [11]. Each presents different trade-offs between overhead, implementation complexity, and precision. Our goal is to allow monitors at all of these levels, as well as other more generally defined abstraction levels.

3 Monitors

A *monitor* is a module that observes a given set of system variables at a particular level of abstraction, gathers information about these variables, and makes some of this information available to other monitors and components. In particular, we expect monitors to report the violation of expected system properties.

Definition 3.1 A property φ is a set of runs of the system. For a run r, we say that r is a *model* of φ, and write $r \models \varphi$, if $r \in \varphi$. We write $r \models \neg\varphi$, or equivalently $r \not\models \varphi$, if $r \notin \varphi$.

A common way of describing such properties is using linear-time temporal logic (LTL) [16]. Properties can be classified as belonging to the *safety* or *liveness* classes [2], where safety properties are those whose failure can be detected by a finite prefix of the run. A monitor that halts and issues a warning or alert is, by definition, checking a safety property. (On the other hand, not all safety properties are monitorable [23].)

There are system properties that cannot be expressed as sets of runs, including possibilistic and general branching-time temporal properties [3,6]. This includes a number of security properties, such as various formulations of noninterference [25].

We will focus on the *detection* of property violations rather than property *enforcement* [21].

Definition 3.2 A *monitor* M has a set of *observable state variables* $\mathcal{V}_M$ and a set of *auxiliary variables* $\mathcal{O}_M$, which are visible to outside modules. The monitor operates by sampling the variables in $\mathcal{V}_M$ and updating the variables in $\mathcal{O}_M$, and raises an *alarm* when it considers that a violation has occurred. Given a system run $s_0, s_1, s_2, \ldots$, the monitor sees a subsequence of states

$$s_{i_0}, s_{i_1}, s_{i_2} \ldots$$

restricted to the variables $\mathcal{V}_M$, that is,

$$s_{i_0}|_{\mathcal{V}_M}, s_{i_1}|_{\mathcal{V}_M}, s_{i_2}|_{\mathcal{V}_M} \ldots$$

where $0 \leq i_j \leq i_{j+1}$ for all $j \geq 0$.

The auxiliary variables $\mathcal{O}_M$ are assumed to be part of the global set of system variables $\mathcal{V}$.

In practice, the variables in $\mathcal{V}_M$ are best defined locally, rather than in terms of global but low-level system variables. This is the case, for example, when we annotate high-level code to check invariants.

We do not make any assumptions regarding what mechanism a monitor M may use to observe the variables $\mathcal{V}_M$, or what internal state M may have. Most monitors can be assumed to be finite-state, though software monitors can have an unbounded stack (such as the pushdown automata used in [24]). A simple example of a formally derived monitor is a finite-state automaton that describes a temporal safety property at a particular level of system abstraction [7], and is composed with the system through code annotation.

In practice, the power and reliability of monitors can vary.

Definition 3.3 (Monitor characteristics) We say that a monitor M is *complete* with respect to a property φ if for any run r such that $r \not\models \varphi$, M is guaranteed to raise an alarm. A monitor M is *sound* with respect to a property φ if whenever M reports an alarm, then $r \not\models \varphi$ (no false alarms).

An example of an incomplete monitor is one that can skip steps at the chosen abstraction level, and thus miss states that violate the desired property. Many intrusion detection systems are examples of incomplete and unsound monitors: they can produce false alarms, and fail to report violations. Note also that the property checked by a monitor may be implicit in the monitor implementation, and not available in declarative form (such as a temporal logic formula).

A monitor can be defeated if its execution is halted. Technically, the safety property that the monitor is intended to check might not be violated, if the execution steps that modify the observed variables are halted as well. This is the case for monitors that check the internal consistency of applications through source code annotations. We investigate this issue in more detail below.

Assume-guarantee specifications: We can specify the properties of monitors in an assume-guarantee fashion [1].

Definition 3.4 Given a system $\mathcal{S}$, a monitor M and properties φ_1 and φ_2, we write

$$[\varphi_1]\, M\, [\varphi_2]$$

to indicate that M is sound and complete with respect to φ_2, when restricted to runs that satisfy φ_1, assuming that M is not disabled. That is, M is guaranteed to soundly report a violation of φ_2 for any run where φ_1 holds.

Note the analogy to the partial correctness described by standard Hoare triples, which are conditioned on the termination of the program. That M can reliably monitor φ ("total correctness") is thus decomposed into two properties: the first can be written as $[\mathit{true}]\, M\, [\varphi]$, and includes soundness. The second states that M cannot be disabled in a run of $\mathcal{S}$.

If we write $[\![M]\!]$ to indicate the set of runs for which the monitor M does not raise an alarm, then $[\varphi_1]\, M\, [\varphi_2]$ can be expressed as $(\varphi_1 \cap [\![M]\!]) \subseteq \varphi_2$.

Definition 3.5 (Monitor composition) If M_1 and M_2 are monitors in $\mathcal{S}$, we write $M_1 \oplus M_2$ to indicate the monitor that raises an alarm whenever M_1 or M_2 does so.

Informally, if M_1 checks that property P holds, and M_2 checks Q, then $M_1 \oplus M_2$ checks $P \wedge Q$: if the combined monitor does not raise an alarm, then both P and Q hold. We also have

$$[\![M_1 \oplus M_2]\!] = [\![M_1]\!] \cap [\![M_2]\!] \ .$$

We can then write simple rules such as

$$\frac{[P_1]\, M_1\, [Q_1]\,, \quad [P_2]\, M_2\, [Q_2]}{[P_1 \wedge P_2]\, (M_1 \oplus M_2)\, [Q_1 \wedge Q_2]} \qquad \frac{[P]\, M_1\, [\varphi]\,, \quad [\varphi]\, M_2\, [Q]}{[P]\, (M_1 \oplus M_2)\, [\varphi \wedge Q]}$$

and

$$\frac{[P_1]\, M_1\, [Q_1]\,, \quad P \to P_1, \quad Q_1 \to Q}{[P]\, M_1\, [Q]}$$

where general $\mathcal{S}$-temporal validity is required of the antecedents [16]. In this way, runtime and static verification can be combined.

The above rules are all instances of a single general rule:

$$\frac{\begin{array}{l}[A_i]\, M_i\, [G_i]\,; \\ (\bigwedge_i (A_i \to G_i)) \to A \to G\end{array}}{[A]\, (M_1 \oplus \ldots \oplus M_n)\, [G]}$$

In the following section, we describe ways in which monitors can be tightly combined in practice, by sharing information in addition to conjoining their alarms.

3.1 Combining Monitors

As we have discussed, monitors can be embedded in application code, or in the OS; they can be a separate process, or be composed of separate modules that observe networks and interfaces.

The $\oplus$ composition operator described above means only that the various monitor alarms are conjoined. In practice, monitors can share and exchange other information. The modes of monitor combination that we consider are shown schematically in Figure 1. An arrow from N to M indicates that the observable variables of M include some of the auxiliary variables of N, that is, $\mathcal{O}_N \cap \mathcal{V}_N$ is not empty.[2] In all cases, we assume that the monitor alarms

[2] In general, circular information flows can occur, where both $\mathcal{O}_{M_1} \cap \mathcal{V}_{M_2}$ and $\mathcal{O}_{M_2} \cap \mathcal{V}_{M_1}$ are nonempty. However, it may be simpler to merge M_1 and M_2 into a more complex monitor in this case.

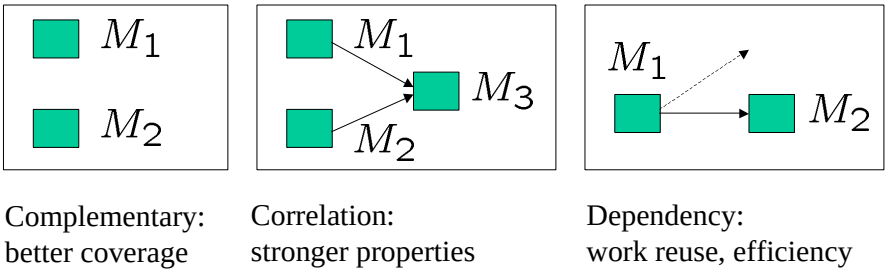

Fig. 1. Combinations of monitors (information flow)

are conjoined. The combination modes depicted are:

- **Complementary monitors:** In this case, monitors M_1 and M_2 independently check their properties. The net effect is to obtain the monitor $M_1 \oplus M_2$ as described in the previous section. Note that even when the monitors do not communicate at runtime, their design can exploit the fact that other monitors are present. For instance, the correct behavior of M_1 may depend on assumptions that are checked by M_2, as we will see in Section 5.
- **Correlation:** Given monitors M_1 and M_2, monitor M_3 relates their activity by including among its observable variables some of the auxiliary variables of M_1 and M_2. This allows checking properties that are stronger than those checked by M_1 and M_2 alone.
- **Dependency:** A special case of correlation, where a monitor checks properties that depend on information passed on by other monitors. This combination can increase the efficiency of the overall monitoring subsystem, allowing the reuse of potentially expensive monitored data.

4 Application: The DIT Server Architecture

Our motivation and sample application is the design and implementation of an intrusion tolerant web server architecture [22]. The Dependable Intrusion Tolerance (DIT) architecture is shown schematically in Figure 2. One or more redundant tolerance proxies manage user requests, and forward them to a redundant application server bank.

The goal of the DIT system is web content distribution with high integrity and availability, at reasonable cost. This is done by using low-cost COTS software, with relatively low assurance, in a high-assurance intrusion-tolerant design. The architecture is based on the observation that COTS web server software is feature filled and complex, and tends to contain security vulnerabilities. However, different programs and operating systems have different vulnerabilities, so a system with diverse web servers on diverse platforms should provide greater assurance of availability and integrity, assuming a reliable mechanism for collecting responses from the redundant servers, validating them, and forwarding them to the clients.

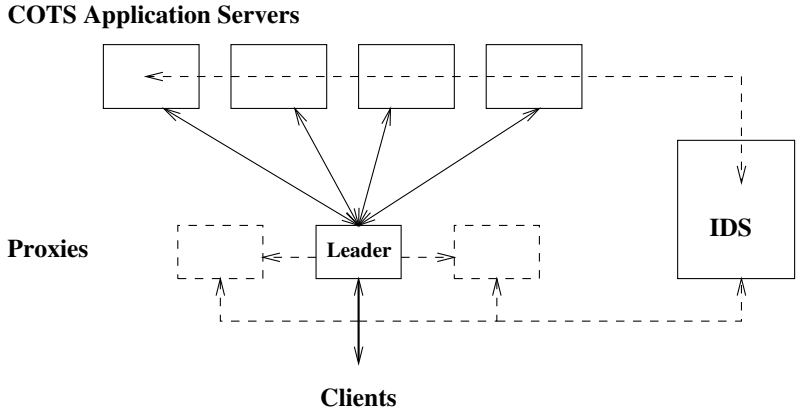

Fig. 2. Schematic view of the intrusion-tolerant server architecture

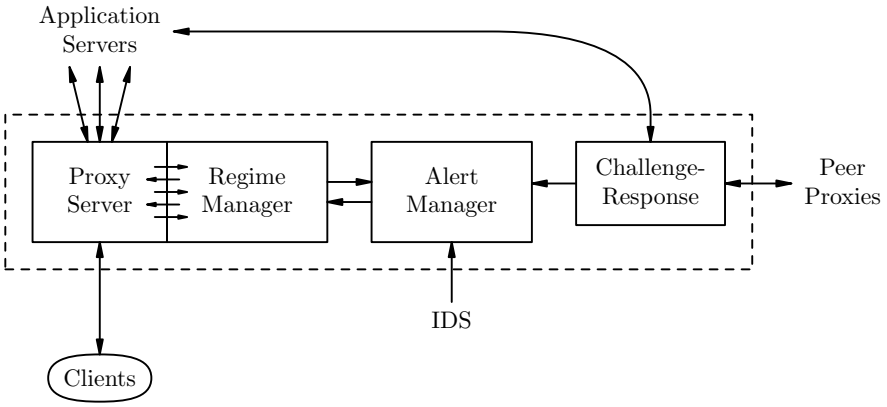

Fig. 3. High-level view of proxy implementation

This function is performed by the proxy, which resides on a hardened platform running a small amount of custom code. The simplicity and customized nature of the proxy software makes it more amenable to hardening than the application servers. The proxy accepts client requests, forwards them to a number of application servers, and compares the content returned by the application servers. If enough agree, the proxy sends the corroborated answer to the client.

An *agreement regime* determines which servers are queried by the proxy for each client request, and how sufficient agreement is determined. For example, a particular regime may specify that each request be forwarded to three different application servers, randomly chosen, and that a simple majority is required among the three replies before the result is sent back to the client.

The proxy and application servers communicate over a private network that is monitored by an intrusion detection system (IDS). The agreement regime changes over time as events such as disagreements and IDS alarms are reported.

Figure 3 shows the main components of our proxy implementation. The regime manager is responsible for executing the content agreement protocol,

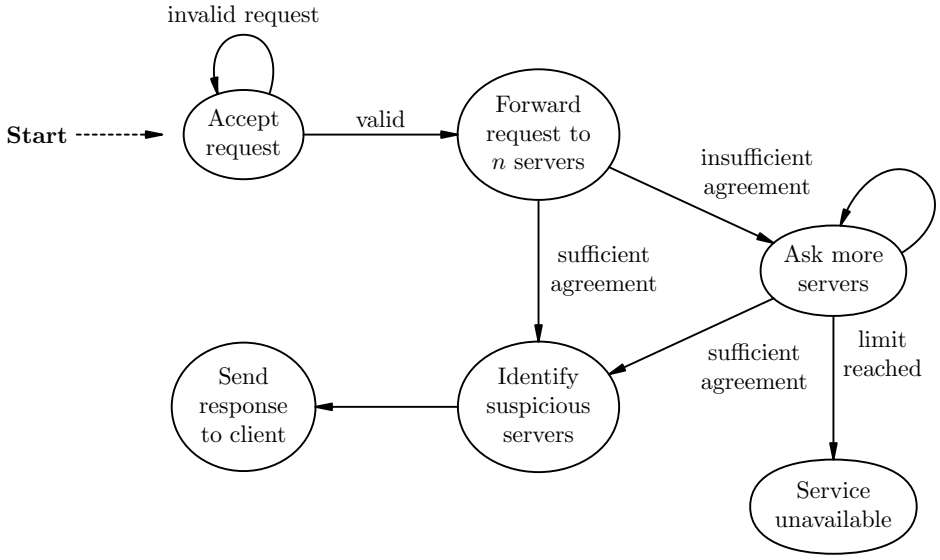

Fig. 4. Generic content agreement protocol

described in Figure 4. The alert manager takes input from the IDS subsystem and the challenge-reponse protocols, and notifies the regime manager when changes are warranted.

A *challenge-response* protocol is constantly executed as an additional integrity check. It provides a relatively high-latency check on the integrity of files and directories on the proxies and servers, by periodically issuing a challenge that must be answered by computing a one-way function of the challenge and the given file.

The challenge-response protocol also provides a liveness check: an alarm is raised if a response is not received. This is an example of a specialized monitor that can help ensure that other monitors are alive and working. (Note that a successful subversion of this protocol is highly likely to be detected by the IDS subsystem.)

5 Examples of combination

We now describe examples of various kinds of runtime monitor combinations.

Code annotation and process monitoring: A common approach to verifying program execution at runtime is source code annotation. This is most applicable to properties that are easily expressed in the source language and in the context of the program design, such as simple assertions, and temporal properties that can be expressed with a few auxiliary variables.

Embedding the runtime verification into the program itself allows close and accurate monitoring, but does not give any assurance that the properties hold for the system as a whole unless there are guarantees that the annotated source is actually running. Verifying this is nontrivial when general-purpose operating systems are used and security compromises are possible. However,

mechanisms to catch such violations do exist, such as those used in host-based intrusion detection systems. Operating system processes can be monitored so that it is difficult for an intruder (or a fault) to stop or kill a program without being detected.

Let P_1 be the source code consistency property, and P_2 be the system property that states that the application program is running. In this example, the code annotation monitor M_A is complete with respect to the internal property P_1, but not with respect to the system-level property $P_1 \wedge P_2$, since M_A cannot guarantee P_2. Thus, we need a complementary combination of monitors that will catch violations of a more global system property.

Such a combination is used in the DIT proxy design. Code annotations check assertions about variables and monitor the execution of the request-handling threads, while a separate monitor M_E is used to verify uninterrupted execution of the entire multithreaded process. The latter monitor accesses fairly tamper-resistant kernel information via the `/proc` filesystem mechanism present in many Unix-type operating systems. A formalization of this argument is

$$\frac{[true]\, M_A\, [P_1]\,, \quad [true]\, M_E\, [P_2]}{[true]\, (M_A \oplus M_E)\, [P_1 \wedge P_2]}$$

In addition, we use StackGuard [4] to ensure that no buffer overflows occur in the proxy code. This can be seen as a third complementary monitor.

Note that M_A and M_E observe different sets of state variables. M_A checks properties of application software variables, such as source code variables and execution stack, while M_E monitors the operating system's state. This illustrates how checking a property that relates many kinds of state variables is made easier by decomposing it into subproperties that each concern a more restricted set of variables, for which monitors are easier to build.

Counting connections: As an example of correlation between monitors, consider a server program that accepts client connections from the network and supplies responses. If the machine running the program is a dedicated server, then the number of network connections to the machine should equal the number of clients being serviced by the program. A violation would indicate a bug in the server software or an unauthorized connection.

To accurately monitor the number of requests being serviced, we need a monitor for the server software, which is obtained by annotating the code. To count the number of actual client connections for the whole system, we need an operating system-based monitor. Finally, a third monitor takes input from the first two and checks that they agree.

We are currently incorporating such a monitor into the DIT proxy, using `/proc` to supply system connection information, as mentioned in the previous example.

The number of connections to the server process could also be counted from

the outside, without altering the program; our approach has the advantage of being able to verify additional properties such as, for instance, that valid responses are being sent for each connection.

Resource and performance monitoring: A more general example of correlation concerns verification of properties that relate resource use and performance. Performance and resource use may each be monitored independently, most likely using different mechanisms. But properties that involve both types of information are also useful. Given a simple server application where client requests are expected to have approximately equal cost, we may want to check that memory and CPU usage are proportional to the number of currently connected clients. Violation of this property could indicate a memory leak, a bug in the server operation, or a client that is exceeding anticipated or acceptable resource allocations.

Combining monitors for network security: Combinations of monitors need not be limited, of course, to systems consisting of a single machine. A networked collection of computers can be monitored, with monitors verifying the state and operation of various platforms, the network, and their combined operation. While the quantity of information flow and the level of trust in communications are usually more limited in a network than between software components on a single computer, the necessity for effectively combining separate monitors is even greater, since different mechanisms are required to monitor different platforms and the network itself.

Network intrusion detection systems that check for possible violations of security in networked systems are already common [19]. Since they are designed for general-purpose networks and because of the volume and complexity of general network data, network IDSs do not verify most aspects of network operation, but instead focus on particular violations, such as the presence of packets containing known attacks, or particular types and sequences of packets. However, in applications where verification of stronger network security properties is desired, network monitoring can be combined with other types of monitors for increased accuracy.

As a simple illustration, consider a database, behind a firewall, accessible via an authenticating web server. Monitoring internal network connections to the database becomes more effective if the monitor is aware of the user associated with each request, and when users log on and log off using the authentication mechanism. By verifying that queries follow login and precede logoff, many requests that arise from penetration or circumvention of the firewall would be caught.

Checking request forwarding in the DIT system: Finally, let us sketch how, in the DIT system design, monitors combine to verify a key property P:

P: The proxy sends client requests to a correct set of application servers.

The proxy server program maintains two key internal values: (i) the list

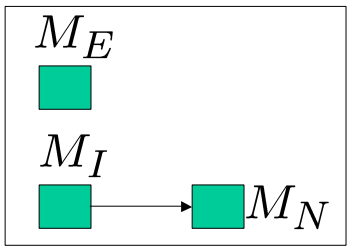

Fig. 5. Information flow for DIT monitors

of servers that are considered uncompromised, and hence may be queried, and (ii) the current regime, which determines the number of uncompromised servers that should be queried for each client request. P means that a request is sent to the proper number of distinct application servers that are considered to be uncompromised, or is not sent at all if there are not enough such servers. The property P can be expressed in terms of several other properties:

P_1: The proxy server is running.

P_2: Each regime transition is allowable, according to a design specification.

P_3: The current regime is actually followed (i.e., the proxy sends a number of requests equal to what is specified in the current regime).

P_4: Servers considered compromised are never queried.

Together, P_1–P_4 imply P (assuming that the regime at system startup is correct).

Property P_1 is verified by the OS-based `/proc` monitor M_E described earlier. The interface to the alert manager allows regime transitions to be monitored, using an interface monitor M_I, which works as long as the proxy server is running. M_I is able to verify P_2 by checking each transition against a specification that describes when regimes are allowed to change. The specification states how the regime is increased when disagreement is observed, and how the regime can be decreased if the number of uncompromised servers drops below the number needed by the regime. The events that trigger the transitions are observable on the interface.

To verify that the number of requests actually matches the current regime, a network monitor M_N, running on a separate machine connected to the internal network, sniffs traffic to see which application servers are the recipients of each forwarded request. The monitor compares this with the current regime and the current list of uncompromised servers, which it receives in network messages from M_I. (M_N also detects the absence of such messages.) Thus M_N verifies both P_3 and P_4.

The relevant types of system variables monitored in this example are

(i) OS variables on the proxy (`/proc`)

(ii) Application software variables (proxy code)

(iii) Protocol status variables (current regime and current list of uncompromised servers)

(iv) Network variables (packets being sent, their sources and destinations)

P_1 concerns the first set of system variables, P_2 concerns the second, and P_3 and P_4 both concern the last two.

To summarize, Figure 5 shows schematically the information flow between the three monitors. The formal argument to show that the three monitors combine to verify P is

$$\dfrac{P_1 \wedge P_2 \wedge P_3 \wedge P_4 \rightarrow P, \quad \dfrac{[true]\, M_N\, [P_3 \wedge P_4], \quad \dfrac{[true]\, M_E\, [P_1], \quad [P_1]\, M_I\, [P_2]}{[true]\, (M_E \oplus M_I)\, [P_1 \wedge P_2]}}{[true]\, (M_E \oplus M_I \oplus M_N)\, [P_1 \wedge P_2 \wedge P_3 \wedge P_4]}}{[true]\, (M_E \oplus M_I \oplus M_N)\, [P]}$$

Of course, there remains the task of ensuring that the three monitors will not be disrupted, which in this case we assume will be established by other formal or informal arguments.

6 Conclusions

We have argued that in practical systems, runtime monitors at different levels of operation must be combined. The design of the system, and the choice of monitoring levels and mechanisms, should take the effect of monitor combination into account. The advantages of combining monitors can be summarized as follows:

(i) Additional monitors can cover missing properties, such as unchecked assumptions (cooperation)

(ii) By sharing information, stronger properties can be monitored (correlation)

(iii) Greater efficiency and modularity in the monitoring mechanism

There are many practical challenges to the effective implementation and combination of monitors. For instance, it would be desirable to have a secure kernel module that supports the exchange of information between monitors implemented at different levels of abstraction.

6.1 Related Work

The combination of runtime monitors has not been addressed in previous work, though much work has been done to develop and test individual monitoring mechanisms. We mention a few that are relevant to the techniques and applications we have discussed.

Intrusion detection can be seen as an important class of runtime monitoring and verification, particularly in the case of *specification-based* IDS [12,13]. Given a specification of potential intrusions, design-time verification can help identify security vulnerability and exploits as well.

The lack of type safety in C programs leads to notorious security flaws, such as buffer overflows, as exploited by the Code Red worm [18]. Several tools address the need for checking type safety properties of C and C++ programs at runtime. The CCured type system [17] uses a combination of static and runtime pointer analysis to catch type errors. Cyclone [10] is a more strongly typed dialect of C that inserts additional runtime checks. StackGuard [4] is a compiler security mechanism that catches many buffer overflow attacks (which we use for the proxy compilation).

The MaC language [15,11] provides a formal language for describing monitors and automatic tools for instrumenting code. The Alamo monitor architecture [9,8] instruments C code in more detail, resulting in less efficient code. Techniques for instrumenting C and C++ code and analyzing the results are presented in [14].

Future work: Our formal framework may be extended to explicitly include abstraction layers, including abstraction and refinement relations between variables, components, and monitors. The assume-guarantee proof notation can be similarly extended.

We have not discussed design issues, such as how a designer knows what monitors to use and how they should be assembled. However, this approach to runtime verification may nicely complement design methodologies that include design-time verification as well. Off-line verification and analysis could be applied as thoroughly as possible, and runtime monitors could then be chosen and assembled to check all remaining critical properties. The choice of monitors can also follow the *design assurance argument*, which describes why the architecture meets the desired operational and security requirements [5].

As we noted in Section 2, some important security properties are not sets of traces [25,20]. Hence, runtime checks alone cannot guarantee these properties. Runtime verification must be combined with design-time analysis to ensure such properties.

Another direction for future work concerns extending our framework to include property *enforcement*, as done by the security automata of [21]; our current focus is on detecting safety property violations.

Finally, another challenge is to extend our framework to include real-time properties and monitors, with access to a global or local system clocks. Real-time constraints allow expressing many untimed progress properties as timed safety properties, which can therefore be monitored.

Acknowledgement

We thank Magnus Almgren, Nikolaj Bjørner, Steven Cheung, Yves Deswarte, Bruno Dutertre, and Alfonso Valdes for their comments and feedback.

References

[1] Abadi, M. and L. Lamport, *Composing specifications*, ACM Transactions on Programming Languages and Systems **14** (1992), pp. 1–60.

[2] Alpern, B. and F. B. Schneider, *Recognizing safety and liveness*, Distributed Comp. **2** (1987), pp. 117–126.

[3] Ben-Ari, M., Z. Manna and A. Pnueli, *The temporal logic of branching time*, Acta Informatica **20** (1983), pp. 207–226.

[4] Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang and H. Hinton, *StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks*, in: *Proc. 7th USENIX Security Conference*, 1998, pp. 63–78.

[5] Dawson, S., J. Levy, R. Riemenschneider, H. Saïdi, V. Stavridou and A. Valdes, *Design assurance arguments for intrusion tolerance*, in: *Workshop on Intrusion Tolerant Systems, DSN 2002*, 2002, pp. C–8–1 – C–8–5.

[6] Emerson, E. A. and J. Y. Halpern, *'Sometimes' and 'not never' revisited: On branching time versus linear time*, Journal of the ACM **33** (1986), pp. 151–178.

[7] Geilen, M., *On the construction of monitors for temporal logic properties*, Electronic Notes in Theoretical Computer Science **55** (2001), RV'01—First Workshop on Runtime Verification.

[8] Jeffery, C., *The Alamo execution monitor architecture*, Electronic Notes in Theoretical Computer Science **30** (2000).

[9] Jeffery, C. L., W. Zhou, K. Templer and M. Brazell, *A lightweight architecture for program execution monitoring*, in: *ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, 1998, pp. 67–74.

[10] Jim, T., G. Morrisett, D. Grossman, M. Hicks, J. Cheney and Y. Wang, *Cyclone: A safe dialect of C*, in: *USENIX Annual Technical Conference*, 2002.

[11] Kim, M., "Information Extraction for Run-time Formal Analysis," Ph.D. thesis, CIS Dept. Univ. of Pennsylvania (2001).

[12] Ko, C., "Execution Monitoring of Security Critical Programs in a Distributed System: A Specification-Based Approach," Ph.D. thesis, Computer Science, University of California at Davis (1996).

[13] Ko, C., M. Ruschitzka and K. Levitt, *Execution monitoring of security critical programs in distributed systems: A specification-based approach*, in: *IEEE Symposium on Security and Privacy*, 1997, pp. 175–187.

[14] Kortenkamp, D., T. Milam, R. Simmons and J. L. Fernandez, *Collecting and analyzing data from distributed control programs*, Electronic Notes in Theoretical Computer Science **55** (2001), RV'01—First Workshop on Runtime Verification.

[15] Lee, I., S. Kannan, M. Kim, O. Sokolsky and M. Viswanathan, *Runtime assurance based on formal specifications*, in: *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[16] Manna, Z. and A. Pnueli., *Temporal verification diagrams*, in: *International Symposium on Theoretical Aspects of Computer Software (TACS'94)*, LNCS **789** (1994), pp. 726–765.

[17] Necula, G. C., S. McPeak and W. Weimer, *CCured: Type-safe retrofitting of legacy code*, in: *Principles of Programming Languages* (2002).

[18] Permeh, R. and M. Maiffret, `.ida` *"Code Red" worm*, Security Advisory AL20010717, eEye Digital Security (2001), `http://www.eeye.com/html/Research/Advisories/AL20010717.html`.

[19] Porras, P. and P. Neumann, *EMERALD: Event monitoring enabling responses to anomalous live disturbances*, in: *Proceedings of the 20th National Information Systems Security Conference*, Baltimore, MD, 1997, pp. 353–365.

[20] Rushby, J., *Security requirements specifications: How and what?*, in: *Symposium on Requirements Engineering for Information Security (SREIS)*, Indianapolis, IN, 2001.

[21] Schneider, F. B., *Enforceable security policies*, Information and System Security **3** (2000), pp. 30–50.

[22] Valdes, A., M. Almgren, S. Cheung, Y. Deswarte, B. Dutertre, J. Levy, H. Saïdi, V. Stavridou and T. E. Uribe, *An adaptive intrusion-tolerant server architecture*, Technical report, System Design Laboratory, SRI International, CA (2001).

[23] Viswanathan, M., "Foundations for the Run-time Analysis of Software Systems," Ph.D. thesis, Computer and Information Science, University of Pennsylvania (2000).

[24] Wagner, D. and D. Dean, *Intrusion detection via static analysis*, in: *IEEE Symposium on Security and Privacy*, 2001, pp. 156–169.

[25] Zakinthinos, A., "On the Composition of Security Properties," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Toronto (1996).