August 28, 2000

# Solutions for Trust of Applications on Untrustworthy Systems

**Jonathan C. Hill, Jack Davidson, John Knight**
**<jch8f, davidson, knight>@cs.virginia.edu**

**Distributed systems rely on non-local applications. At the same time, non-local applications can only be trusted as far as a non-local systems can be trusted. This is inadequate for the purpose of monitoring and maintaining critical infrastructure that relies on a distributed computer system. We require a distributed, flexible, and reliable application system to act non-locally throughout a network. Flexibility encourages a model that utilizes application level processes, dispatched from a trusted source system to untrustworthy non-local systems in the network. Reliability requires that the local system be aware of the state of operation of its dispatched application on the inherently untrustworthy non-local system. Unfortunately, these requirements lead to a scenario of a trust gap, in which a dispatched application level process must correctly function while relying on non-local (to the dispatcher, local to the application) system services which cannot be trusted. This is the inverse problem of the untrustworthy incoming application, in which a trusted system is asked to support an untrustworthy application. As such, a trust gap comes with a critical, unique, and difficult set of properties of great importance for the development of fault tolerant distributed systems. In this paper we will consider hardware and low-level software solutions to the trust gap problem. We develop a taxonomy of possible solutions and investigate the promise of each approach. Of these, we find two solution approaches with potential and applicability to today's distributed computing environments.**

## 1.0 Introduction

Critical distributed infrastructures must react to faults; reconfiguring applications, distributing new tasks and priorities, and preparing to defend against additional faults. Without such capability, the survival potential for critical networks is highly questionable. Yet today's critical distributed infrastructures are not prepared to handle non-local faults in an automated manner. As it stands, what happens to one part of a network is either watched helplessly by the remaining network, or only adapted to locally by some simple distributed algorithm. This is regardless of the fault's impact on overall network functionality. What is needed is a network that can respond in a more intelligent manner. The ideal network would react more as organism than architecture, responding to faults with corrections, healing, and posturing, rather than passively crumbling and awaiting manual repairs upon each deterioration.

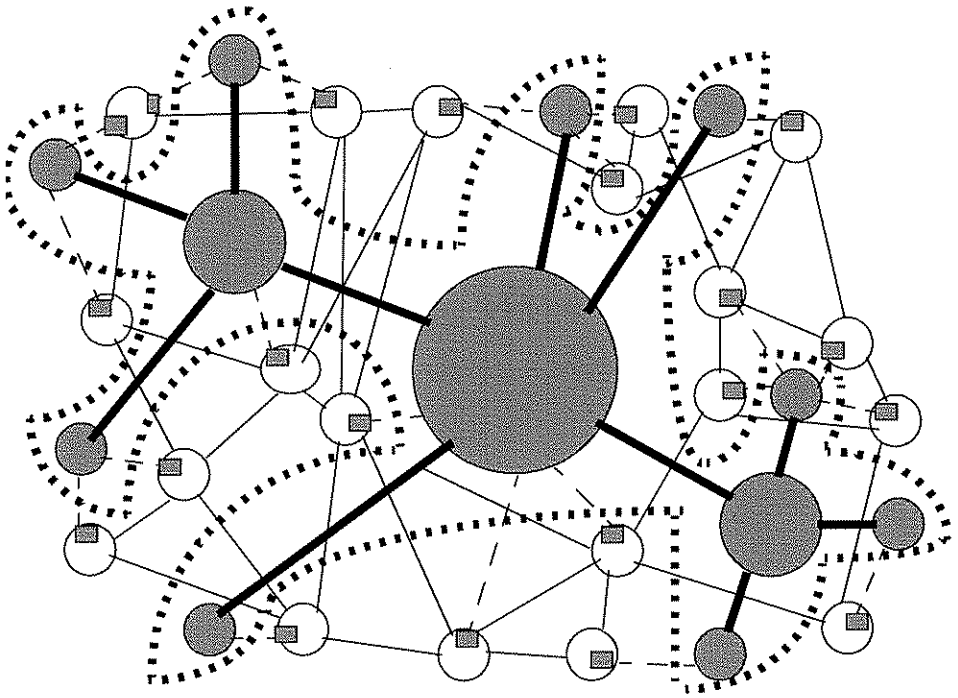

**FIGURE 1.** A graph of a fault-tolerant distributed network consisting of an untrustworthy work network (clear nodes and thin connections) with a hierarchical reactive "nervous system" support network (shaded nodes with thick connections.) Work nodes are monitored by the support nodes. The dashed curved line indicates the trust boundary between the support network and the work network. The small shaded rectangles within work nodes are support system applications which must maintain clarity of their functionality while executing on the untrustworthy work nodes.

A critical infrastructure composed of a large, heterogeneous distributed system must include network wide, non-local fault tolerance in order to maintain survivability. What fails for one portion of the network must not be ignored by the rest of the network.We

propose a network architecture that contains a "nervous system", and hence contains a highly guarded, reactive, information propagating and hierarchical control infrastructure. Such a network is depicted in figure 1.

The network depicted in figure 1 contains "work" nodes and "support" nodes. Work nodes perform the work of the distributed system, while "support" nodes are responsible for maintaining the clarity of the distributed system. The support nodes of the network are carefully maintained and highly guarded. As a result they are trusted by the network, much as a nervous system is highly guarded and trusted by an organism. Meanwhile, the work nodes are numerous and not intrinsically guarded. Thus the clarity of work nodes is assumed suspect by the network as a whole.

The support nodes act as a nervous system for the network. They report and organize conditions on the network, as well as respond to changing network conditions. Support nodes detect non-local faults in the network and react, reconfiguring the network in the presence of damage, responding preemptively to threats, and rearranging distributed services as needed. In general, it is the support node system which is responsible for the effective arrangement of the distributed system's work and services.

**For the remainder of this paper, internal trust of the nervous system is a stated assumption of the network. In other words, the nervous system, as a network, trusts itself.**

Due to the trusted nature of the support nodes and the untrustworthy nature of the work nodes, a trust boundary exists at the interface to the work nodes. This trust boundary is depicted in figure 1. From the perspective of the nervous system network,

**we cannot assume that the work nodes are trusted.**

In particular, it is network wide non-local faults in the work nodes and their network that we intend to tolerate. It is the natural relationship between the fault-catcher (nervous system) and the fault-capable network (work network) that renders the work nodes inherently untrustworthy. Yet,

**it is of critical importance for the reactive system that trustable information gathering and reconfiguration be accomplished on the untrustworthy work nodes.**

If the network is to survive an attack (as an example,) then it must obtain accurate information from potentially compromised work nodes, as well as enforce actions on these nodes. As a result,

**the nervous system network is required to enforce non-local action where trust cannot be given.**

This is a principal problem of fault-tolerant network design. How do we maintain clarity of sensor/actuator applications dispatched by a trusted nervous system across the trust boundary to work nodes? Complicating this question are goals such as portability and reconfigurability. Our fault-tolerant network must be able to reconfigure and manage application distribution over the work network, dynamically, with maximum flexibility. Meanwhile, the work node network may consist of a large collection of COTS compo-

nents as well as legacy systems. For this reason, the application level for dispatched sensor/actuators best suits flexibility and dynamic capability.

Work at the application and high-system level has yielded interesting methods of protecting applications [Wang, Hill, Knight, Davidson 00.] These techniques harbour protection from static analysis of the application within the application itself, with the original static analysis information of the program being a secret held by the trusted nervous system. While this hinders intelligent tampering and impersonation attacks, we still require a solution that protects us from the root causes that allow protection violations and attacks in the first place. The root cause is the lack of trust provided by the underlying system on the work node. Therefore, this paper will focus on these low-level system components in its analysis and its solution space. The remainder of this paper is devoted to finding solutions to maintaining the trust of applications dispatched from the nervous system to work nodes on the system. What is needed is an examination of hardware and low-level software in light of an untrustworthy operating system. Section 2 defines application clarity and a compositional operation for system clarity, as a formalization of what is meant by observable trust. We consider currently existing system architectures and discover the "trust and clarity gap"; a natural product of network trust models attempting to extend local trust onto a non-local system. The gap between high and low-levels on the remote system further suggests the importance of low-level system solutions. From these observations, section 3 proposes a taxonomy of possible solutions. Sections 4, 5, and 6 explore this solution space. Section 7 summarizes our work, considers hybrid solution models, and discusses the direction of further research.

## 2.0 The Trust and Clarity Gap

### 2.1 A Definition of Clarity

A process can be described by many properties, such as real-time guarantees, which relate the operation of a process to a system and the process itself. In this report we seek to maintain the clarity of applications running on untrustworthy systems. We formally define clarity below.

- **Application Clarity** over property P is the extent to which reliable observation that an application obeys or does not obey a liveness or safety property P can be obtained.

Informally, as an integral part of a system, a component with clarity of P is one that can, at time t, be reliably observed to determine its relation to P. This differs from the traditional notion of trust, in which trust is the degree of confidence in the consistent adherence (or violation) of a system to a property P. Informally, we define clarity of a system as the extent to which an observer can obtain the necessary information to compute trust of the system. Trust implies clarity, but clarity does not imply trust. That is, clarity is a necessary precondition for un-assumed trust.

### 2.2 A Sketch of Some Important Liveness Properties

We will be considering application clarity for several liveness or juxtaposed safety properties, defined as

- **Operational Liveness** indicates that an application:
  1. contains only code that the application author intended it to contain to run on a system;
  2. contains only the data that the application author intended it to contain to run on a system;
  3. maintains its contained data state such that the only actions having been performed upon it are those intended by the application code (including registers);
  4. is initialized in a manner semantically equivalent to the application initialization protocol agreed to (between the application author and the run-time system) prior to run-time instantiation of the application; and
  5. as a process, performs instruction streams, in serial or parallel execution, only as intended by (semantically equivalent to) the original application implementation.
- **Privacy Liveness** indicates that an application is maintained so as to receive correct enforcement of memory and data privacy services from a security policy on the run-time system, as stated to the application author just before the application is run-time instantiated. (includes register privacy)
- **Reservation Liveness** indicates that an application receives process scheduling and other resource allocations in a manner consistent with the policy of the run time system stated to the application author just prior to the run-time instantiation of the application. If any change is to take place in the scheduling and other resource allocation policies of the system to the application, they must be stated to the author prior to being put into effect.
- **Resource Assignment Liveness** indicates that an application receives the actual resources it requests, rather than spoofed resources, or alternate resources that were not intended by the application design.

We will be concerned with operational liveness, privacy liveness, fair reservation and assignment liveness clarity. We will utilize these forms of clarity to maintain an adequate level of trust for an application residing on the untrustworthy system-enough to meet our purposes in crossing the trust boundary in the first place.

## 2.3 Clarity Model

Let clarity(P,A,t) be a measure from 0 to 1, where 1 is absolute knowledge that a liveness or safety property P is obeyed or not for application A up to time t. 0 is absolute knowledge that P is either obeyed or not obeyed by A up to time t. Intermediate measures are a confidence rating in application A obeying or not obeying P up to time t. Therefore we have two ways to utilize quantify clarity in a system. We can utilize a scalar measure from 0 to 1, or simply utilize Boolean clarity, where a system either maintains clarity (1), or does not (0.)

### 2.3.1 Clarity Composition

Given a system of components, we can compose knowledge of clarity for the system as a whole from the components. Let our model consist of **n** properties, for which we wish to model clarity propagation. Let each component **i** of our system depend upon the resources and services of components in the set $S_i$. Let the internal clarity of a compo-

nent (without considering dependences) be $\mathbf{J_i}$. Let component **i** have clarity measure for property **P** up to time **t** of **I(P,i,t).** Let it be that the clarity of property $\mathbf{P_a}$is transferred from resource or service of component **k** to a dependant component by function $\mathbf{C_k(P_a,I1,...In)}$ where $\mathbf{I_1}$ to $\mathbf{I_n}$ are clarity values of the **n** properties at component **k** at time **t-1**. Then the clarity of property $\mathbf{P_a}$ at time **t** at component **i** is given by

$$I(P_a, i, t) = J_i \prod_{k \in S_i} C_k(P_a, i, I(P_1, k, t-1), \ldots, I(P_n, k, t-1))$$

In words, a function $\mathbf{C_k}$ at component **k** describes how the clarity of each property at **k** just prior to our next clarity determination effects our clarity knowledge of our property for an observing application. Since this function is a variable of the observing component, the component being observed, and previous clarity values for all properties of our model, it is a function very specific to the individual system being explored.

We will also be interested in components which actively determine the clarity of other components. Such components will generally monitor other components to determine the status of some liveness or safety property P on the observed component.

## 2.4 Trust and Clarity Gaps

Consider the typical modern, layered system architecture, as depicted in figure 2. Each higher layer relies on lower levels and itself for services and resources. There is a direct correlation between intrinsic clarity, and the depth of a layer. Applications tend to have increased complexity over the more general services at lower levels. Also, higher levels tend to have more specialized services and resources, and therefore tend to not be as well tested and used. As a result, less clarity information is naturally collected about higher level functionality than lower level functionality. In summary, clarity tends to decrease with increasing dependence, complexity, and decreased testing or use. This leads to a natural trust vector on isolated computer systems- with each system layer looking for trust from the layers beneath it, which happily, correlates with the services and resources on which it relies and for which clarity information is more readily available.

In today's networked world, as depicted in figure 1, nodes cannot afford to assume the trust of one another. Figure 3 represents two nodes of a system. The left hand node represents a trusted dispatching node which will send a trusted application to run on a remote node. The right hand node represents the remote node. From the dispatching node's perspective (left node of figure 3), the dispatched application code is trusted, but the services on which it relies cannot be trusted at the remote (right hand) node.

The software of the remote node is assumed to be untrustworthy, as the network is large and faults (including attacks) will occur. This is the required model for the left node to be able to provide non-trivial, non-local fault tolerance for the remote system. For example, if the left hand node is to detect insidious deliberate faults, it must start with the assumption that the right hand software cannot be trusted. As a result, the left node can-

not rely on software on the right node to say "Everything is fine" as this reporting is untrustworthy. **Non-local clarity requires some trusted component on the non-local machine.**

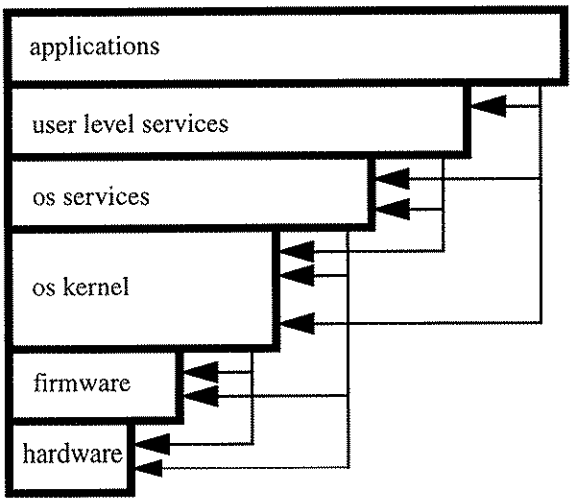


**FIGURE 2.** The clarity and trust hierarchy of the typical system architecture. Arrows indicate service and resource dependencies between layers of the system.

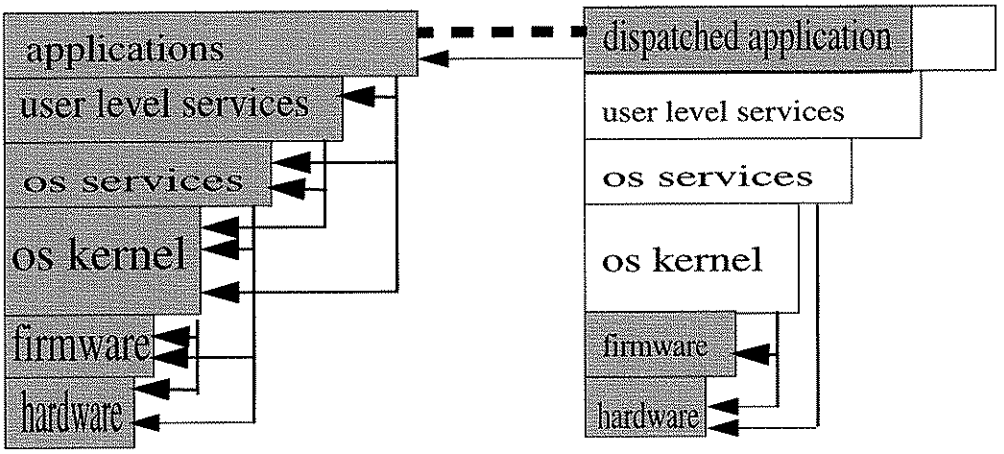


**FIGURE 3.** Trust system model for a trusted dispatching (left) sending an application to an untrustworthy work node (right.) Shading indicates Boolean trust, with clear boxes indicating a lack of trust. Arrows indicate trust dependence. The dashed line indicates a trusted link between the two nodes.

The application is built and verified on the left hand node, and is thus a trustable component (internal clarity, J=1) before it is dispatched. The application, which must remain trusted, is now present on top of untrustworthy services and resources. In other words, a natural extension of the trust hierarchy on the local node (left) becomes unreliable when transported to utilize components at the remote node (right.)

This is unfortunate, because without trust for the dispatched application system, we must actively monitor the services and resources on which the application is dependant in order to maintain its trust. However, just as there is a trust gap, there is also an clarity gap for the application. It cannot monitor resources and services for clarity without relying upon these resources and services.

- We utilize the right hand node of figure 3 to visualize the **trust and clarity gap** scenario.
- It is clear that trust and clarity gaps occur naturally on an inherently untrustworthy distributed system with dispatched trusted applications.
- **Trust Relativity**: Figure 3 represents the standard large network scenario for trust, namely **trust relativity**. Each node trusts itself but cannot trust the resources or services of other nodes. We can strengthen our arguments by not assuming "global" or "absolute reference frame" of trust in the system. However, since our system maintains a trusted nervous system, we will often be able to eliminate trust relativity conditions.

## 3.0 Solution Taxonomy

In this section we will consider ways to maintain trust of the dispatched application in the presence of a trust and clarity gap. Our solution taxonomy will rely on the observation that an clarity gap (and hence trust gap) is a necessary precondition for untrustworthiness of a trusted application dispatched to run on a remote, untrustworthy node.

Assume that an application is prepared at the left hand node of figure 3. It is verified so that its internal components in composition are trusted. Thus we have full clarity of the internals of the program before we ship it off to the remote system. It is then dispatched to the remote node in the network over a trusted link. Since the behavior of the program when it receives its required resources and services is trusted, then on any remote node where services and resources are trusted, the execution of the application can always be trusted back at the dispatching node (from our trust composition operation.) It is precisely because we do not assume trust of a remote node's services that we require clarity from the remote node's services. In the presence of an clarity gap, we cannot establish trust. Hence a trust gap remains for levels of the system on which the formerly trusted application relies for trusted execution. As a result, the trusted application cannot execute in a trustworthy manner on the remote system.

Traditionally, there are two methods to keep a necessary precondition out of the picture: prevention, and avoidance. Detection is also an effective solution so long as reacting to a positive detection of a necessary precondition can lead to acceptable action to remove the precondition instance. We will divide up our solution space by solutions which we will define as trust gap prevention, trust gap avoidance, and trust gap detection.

- **Trust Gap Prevention**: This technique attempts to eliminate trust gaps for all remote nodes (and hence any clarity gaps.) This can be done by inserting our dispatched application at a base trusted level of a remote system, or by rasing the level of trust of the remote system. Both techniques have been pursued by other research groups and are discussed in section 4. We investigate shortcomings of the Pentium and modifications to its system architecture that will help raise trust of address space protection to the application level.
- **Trust Gap Avoidance**: Trust gap avoidance relies on there not being an clarity gap on the remote system in order to dynamically compute trust of the software layers of the remote system. We will consider such approached in section 5. One solution is to have clarity local to the application when it arrives at the remote node, and allow it to determine trust of the system.
- **Trust Gap Detection**: Trust gap detection does not rely on the absence of an clarity gap. It asserts trust by running a remote trusted automata to detect trust safety violations. The monitoring device is attached to the highest level of the remote system with clarity (for our purposes, hardware.) It observes the behavior of the hardware to detect safety violations committed against our application from untrustworthy levels of software. By detecting any safety violations of untrustworthy layers at run time, trust of our remote application can be maintained. Trust gap detection uses lower level clarity without intermediate clarity. This is possible because necessary process resources for the monitor are maintained within the trusted monitor device.

### 3.1 Hardware Assumptions

In the following sections we will consider the advantages and disadvantages of each approach. Although only stated for one section, our implicit assumption is a system architecture similar to modern day commercial architectures such as the Pentium III system.

At some level we must trust a remote node. If we trust nothing of the remote node, then we have nothing at the remote site with clarity. We will make the assumption that the hardware, and perhaps firmware, at a remote node, are trusted. This assumption means that we will be primarily concerned with detecting trust gaps in the software components of the system, from the operating system kernel on up to the application level services. We will specifically state circumstances in which hardware is not trusted.

## 4.0 Trust Gap Prevention

Trust gap prevention relies on system properties that do not allow a trust gap to occur. We can prevent trust gaps by making certain that wherever we send a trusted application, trust can be maintained on all of the services and resources the application will utilize. Figure 4 shows the two general paths available in trust gap prevention. We can either raise the trust level up to our application, or lower our application to the existing trust level on a system.

There are natural advantages to trust gap prevention. It is efficient to be able to simply dispatch an application to a foreign system we know we can trust. In other words, we would have no need for clarity information at run-time. It is also simple, from the sys-

tem design perspective, in that application dispatching over secure links is sufficient to maintain application trust from the servicing node.

On the other hand, moving the application position in the hierarchy, or moving the trust boundary in a remote node may introduce difficulties in implementation, limitations in application, or security concerns for a node that receives an application from a foreign dispatcher.

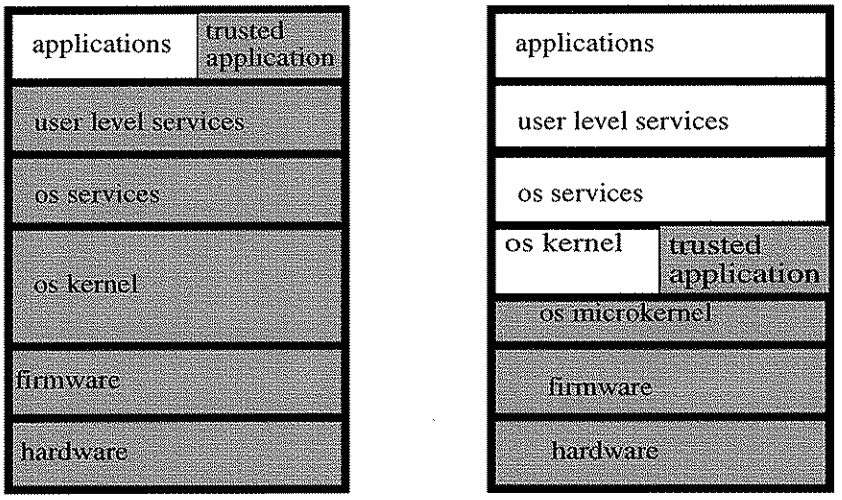


**FIGURE 4.** An illustration of two methods for trust gap prevention. Trust ascent raises the inherent level of trust for all nodes to the boundary with the application level. Application descent lowers the level of a trusted application to interface the boundary of a trusted microkernel.

### 4.1 Clarity Ascent through Hardware Run-Time Trust Checks and Enhanced Clarity Capabilities

Static verification of core operating system trust properties has proven to be a difficult and expensive task. Establishing a trusted code base from static observation is the traditional method of trust ascent. However, it is expensive, requires full access to code in order to establish proofs, and requires reanalysis for non-incremental system changes. In other words, clarity of trust properties at system construction time is available, but expensive to utilize. In a heterogeneous reconfiguring network, COTS components and legacy code will form numerous combinations that may change rapidly over time. As a result, static proofs for a trusted code base are inadequate for our purposes.

In order to implement trust ascent to the application level, we must derive a technique for which clarity is less expensive to utilize, but which continues to guarantee that a trust

gap cannot arise as an inherent property of the system. Run-time checking and local fault tolerance for trust properties can eliminate the need to verify such properties at system construction time. Fault tolerance for run-time trust faults is beyond the scope of this work, but we can at least bolster clarity with run-time detection of trust faults. Software run-time checks are unlikely to be sufficient as they will sample at the rate of execution of the checking software. They are also subject to tampering. Our solution for clarity ascent is to augment hardware to guarantee trust properties at run-time which cannot be guaranteed statically or in software with sufficient speed or clarity. While we are modifying hardware, we can also provide mechanisms to enhance run-time clarity for an operating system.

We will illustrate this approach by guaranteeing properties of a hardware protection mechanism. Memory protection is an important portion of operational, privacy, resource reservation, and resource assignment liveness properties. Without memory protection, we cannot guarantee operational clarity of code or data state for a process. Likewise, privacy requires memory protection disallow accessibility between processes. Resource reservation and correct assignment require that assignment tables maintained in memory are adequately protected.

The Pentium processor, as with any current general purpose processors, relies on the operating system to set up and control data tables utilized by the hardware for protection mechanisms. Figure 5 shows critical components of a typical system required to establish protection trust for an application, with focus on the basic model of the Intel P6 processor series system. Arrows point from a component towards those on which it relies.

- Notice that almost all components rely on memory (and register) protection in order to be protection trusted.

At the bottom of the graph are the data tables that the hardware requires in order to function correctly. Memory protection requires these data tables to be trustable. However, for these data tables to be trustable, memory protection (from the operating system) must be trustable.

**What we observe is that there is a circular trust dependence between the memory protection mechanisms in the operating system, and the hardware-required protection tables that the operating system must maintain.**

In the typical isolated system scenario (no network) the memory protection system is trusted as well as the hardware-required data tables. As a result, component trust easily propagates to higher level components. We cannot afford to make such assumptions about non-local node operating systems. Therefore, we will augment the Pentium hardware with:

- an isolated kernel address space
- hardware run-time check rules

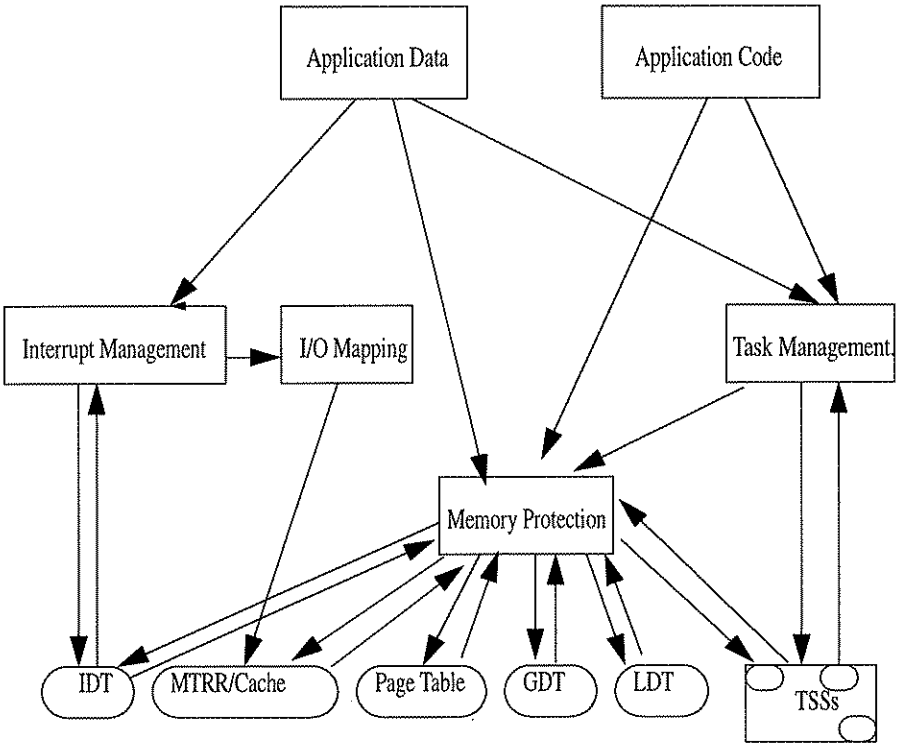


**FIGURE 5.** A simplistic trust dependence graph for basic P6 system components. An arrow is drawn from the trusting object to the object that is to be trusted.

#### 4.1.1 Isolated Kernel Address Space

The Pentium maintains a separate address space for I/O. We will extend this concept by maintaining a separate address space for the operating system kernel. The operating system kernel will be defined by any code that is running at privilege level 0 (most privileged.) The address space contains two components.

1. A ROM code address space
2. A RAM data address space

Access to address spaces is defined as follows:

- A process with current privilege level is 0 will fetch instructions from the separate ROM address space.
- A process with current privilege level is 0 will fetch data from the separate RAM data address space if the address range is within a hardware prescribed range (the upper 2 GB of a 32 bit physical address.)

- Any process with current privilege level greater than 0 will always access code and data from the regular address space.

Thus access to the kernel address spaces is an implicit operation. We have implicitly isolated a kernel code and data space. We have also intrinsically forced the code space of the kernel to be execute only, even to the kernel processes.

Kernel-Only RAM space is an address range of RAM that represents a different RAM bank for level 0 processes than other higher level processes. It can only be accessed by privilege level 0, kernel mode operations. The immediate effect of this limitation is that regardless of protection table information, only kernel mode process can modify this memory region. The hardware-required data tables are to be put in this address range, as well as other critical kernel data information.

It is assumed that the Kernel Only RAM space be large enough to accommodate all TSS and page tables the system may require, as well as space for other kernel data.

At the same time, this introduces a data address range in the standard address space that is inaccessible to directly read or write by the operating system kernel although, it is free to allocate or deallocate it.

### 4.1.2 Additional Memory Protection Data and Run-Time Checks

In addition to the address spaces presented above, the hardware can be extended to perform checks on what sorts of processes have accessed given memory regions as follows:

- Hardware data table access faults if the table is outside the kernel RAM address range.
- Page table entries and segment table entries maintain a bit that is set whenever NON kernel ROM code (level 0 processes) accesses a page.
- If ROM Kernel code accesses data in a page or segment that has been accessed by NON ROM Kernel code (the bit is set) a fault occurs.

Note that the kernel can clear the bit in a segment or page table entry in order to access the segment or page. The Pentium currently has a reference bit in page and segment tables. We merely wish to have a separate bit for reference by non-kernel level processes.

The run-time checks allow the processor to keep track of whether other memory regions of memory space have been accessed by non-kernel level processes. Furthermore, they prevent the kernel from "unknowingly" accessing memory that is assumed to be sheltered from non-kernel process access. In other words, to access memory that may have been accessed by non-kernel processes, the privileged mode process must first clear the segment or page table bit (in kernel RAM space) which indicates whether a segment or page has been touched by non-kernel processes. This increases the availability, with careful kernel coding, of run-time protection clarity information, allowing additional trust properties to be deferred to run-time maintenance.

### 4.1.3 Properties of Isolated Kernel Address Spaces and Run-Time Checks

Adding the isolated kernel address space and run-time checks provides us with the following run-time guaranteed protection trust and clarity features which do not exist in a traditional Pentium system:

- Only kernel processes may access or modify hardware-required table information. (Trust)
- Kernel code cannot be modified or read. (Trust)
- The kernel may store additional data in an address space only kernel level processes can modify. (Trust)
- Only the physically installed kernel code in kernel ROM can be executed as level 0 privilege code, and hence execute privileged mode instructions. (Trust)
- Instructions fetched from outside the range of legal kernel ROM by level 0 processes result in a memory access fault. (Clarity)
- Kernel processes cannot access data that has been modified by processes other than those of the kernel without at least acknowledging this possibility by clearing the page and segment table info of non-kernel process accessed bits. (Clarity)

Our run-time enforced trust and clarity mechanisms have reduced the burden of verification in operating system software design. So long as hardware is trusted, our additional checking can be trusted.

We cannot guarantee with run-time checks of our magnitude that kernel level data will not be attackable by stack smashing, and such, due to poor kernel or kernel interface programming. However, we can guarantee that no such attack will change kernel code. As a result, the most prominent problems of a kernel attack, namely replacement of kernel code with some other code to execute with privilege, has been eliminated. What remains is the possibility that poor kernel design will lead to abuse of kernel routines through abuse of the parameters or non-determinism errors in such functions. Kernel data might be corrupted in this way. If the protection tables are corrupted, protection could still be compromised for the system. Verification of kernel properties in a design and implementation is still required, though more properties are hardware trustable, and others can be made simpler to prove by use of our enhanced clarity checks. If such proof of algorithms and implementations in the static kernel ROM can be established, then memory protection can be trusted. The result would be propagation of component trust for memory protection to higher level components. The resulting picture of trust is as described in figure 6, in which trusted required dependencies are made bold. We can see that the hardware-required data tables can now trust memory protection, as it is enforced by hardware. As a result, there is no longer a circular dependence of trust between memory protection and the tables which enforce it. Apart from weakening attack strategies, we have made it simpler to prove remaining properties of memory protection.

Our architecture adjustments impose certain restriction on the way a kernel should be designed and interact with the surrounding system. For one thing, kernel level system calls on a Pentium maintain a level 0 stack for use by a process that passes through a call gate to level 0 privilege mode (this avoids task switch overhead.) A good implementation for trust propagation would utilize kernel RAM address range for the level 0 stack

of the process. The kernel can copy level > 0 stack information to the level 0 stack as needed. Furthermore, some system functionality that might fall outside a kernel by modern standards, such as some task management, might be required to be considered, once again, part of the kernel. This is due to the fact that privileged mode instructions can only execute from the ROM kernel code address space. It should be noted that the more run-time hardware trust bolstering we do, the more restrictive the operating system kernel design becomes.

We have described how trust can be ascended through modifications of Pentium hardware. However, we only demonstrated this for memory protection. Other required features, such as interrupt management, I/O mapping, and task management, are also important parts of operational and resource reservation and assignment liveness.

1. Attempt to establish hardware systems for system-design-time proof of additional component trust.
2. Attempt to establish run-time hardware systems to help manifest trust of these components at run-time.
3. Rely on external observation to determine some resource reservation and assignment trust. Some trust can be obtained by observing an output signature of the application over a trusted link to the dispatcher node. As an example, some trust of fair scheduling can be obtained by observation of a trusted link signal from the application.

One solution to the I/O mapping problem might be to have I/O mapping be committed in firmware to be unadjusted by software tampering. Current system BIOS's, which cannot be overridden by an operating system and are password protected would be adequate for this task. A more permanent I/O mapping would also be a solution. In the section on Trust Gap Detection we will discuss an important alternative to the techniques discussed here.

### 4.1.4 Conclusions about Trust Ascension

Trust ascension has the advantage that it does not require us to create special privileges for trust for our dispatched application on the remote node. It also does not upset the traditional balance of centrally based trust, relative to each node of the distributed system.

Our solution approach is to provide guarantees of trust properties at run-time through hardware, thus reducing the burden of proof from static code analysis and model checking of large and complicated operating system components. While we have demonstrated memory protection bolstering through run-time checks, higher level resources might require additional hardware to achieve the same effect, if trust propagation cannot be reasonably maintained through static proof.

Further investigation of this avenue, through analysis of operating system components by the trusted code base community, would be valuable. While they attempted to statically verify properties of trust, our approach, in summary, is to guarantee them at run-time through hardware and reduced software verification requirements.

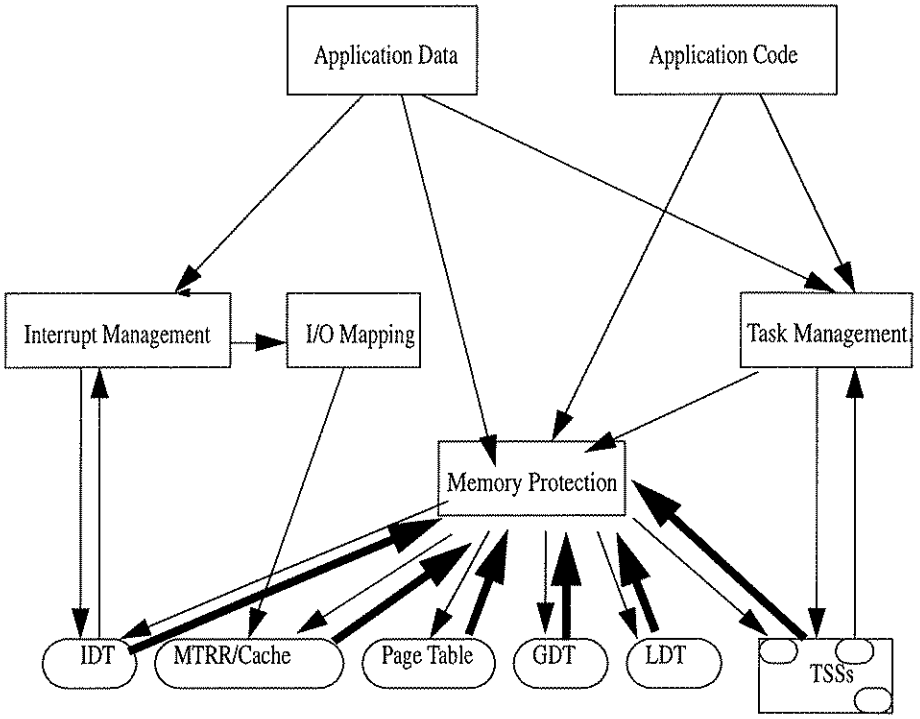


**FIGURE 6.** A reworking of figure 4 for our modified architecture. Trust dependences which now carry trust are drawn as thicker arrows. We have eliminated the trust dependence cycle present in figure 4.

### 4.2 Application Descent through PCC Applications on a Trusted Micro-Kernel

In this solution, out goal is to have the dispatched application rest on the trust border at the remote system. The remote system must have a remotely trusted computing base consisting of a micro-kernel. Such a micro-kernel may be difficult to prove, but should only provide enough functionality to hand control of the hardware over to the trusted application, with a little abstraction of hardware resources. The micro-kernel would be maintained in ROM to avoid susceptibility to overwrite attacks. It is possible that we could simplify the task of trusting a micro-kernel through the hardware mechanisms of section 4.1.

The result of this application descent is that the application must be compiled to run at a level of abstraction and responsibility far beneath what a typical application must handle. This can be done at the trusted dispatching node, knowing the micro-kernel protocol at the remote node. The net result, however, is a net loss in application portability, and

the requirement of smarter, more elaborate compilers and linkers on the trusted node. We would also loose the ability to dispatch trusted COTS applications.

Furthermore, the micro-kernel would need to provide adequate resources and services such that the application could maintain operational, reservation and assignment, and privacy liveness properties. Such a micro-kernel would be required to allow the trusted application to schedule itself, or provide a scheduler within the micro-kernel. Likewise, either priority would need to be given to the application for interrupt assignment, or the micro-kernel would need to handle interrupt reservation assignment. In other words, the micro-kernel model would need to handle very little and "hand over the system" to the application. Otherwise, the micro-kernel would rapidly balloon into a complex traditional operating system.

Since the computer will be handing over substantial power to an application, at issue with this technique is the extent to which the incoming application can be trusted by the remote node. The remote node would require a sophisticated proof carrying code analysis technique in order to justify handing over low-level system capabilities to the incoming application. PCC is an active area of research/ However, if we allow that the network assumes trusted of the support nodes, then such proof carrying code is not required. Existing micro-kernel infrastructures, if trust verified, might be sufficient to implement this technique.

## 5.0 Trust Gap Avoidance

Trust gap avoidance relies on there being no clarity gap, in order to maintain trust for a dispatched applications required services and resources. Trust is maintained by actively monitoring the system layers for trust, using the available system clarity. Rather than seeking to detect trust faults, however, we seek to determine a proof of trust upon insertion of the application. In other words, upon receipt of a trusted application, a work node utilizes extensive clarity to guarantee to the dispatching node that trust will be maintained for the application.

One technique would be to have any services and resources required by the trusted application, such as fair scheduling, memory protection, registers, i/o, etc., be a required to present proof to the dispatching node, that they can be trusted. Such proof carrying resources and services, represent a solution parallel to the inverse problem of proof carrying code for untrustworthy applications. We do not discuss this technique further in this report.

## 6.0 Trust Gap Detection and Application Specific Trust Detection

Trust gap detection is an attempt to detect a trust gap when clarity at intermediate levels of a system hierarchy cannot be obtained or utilized efficiently. Instead we obtain clarity for higher levels of the system by monitoring system behavior at a low, trusted, level. If clarity for higher levels is not too expensive to extract from low-level state information, then we have a method of detecting trust violations (existence of a trust gap) at higher

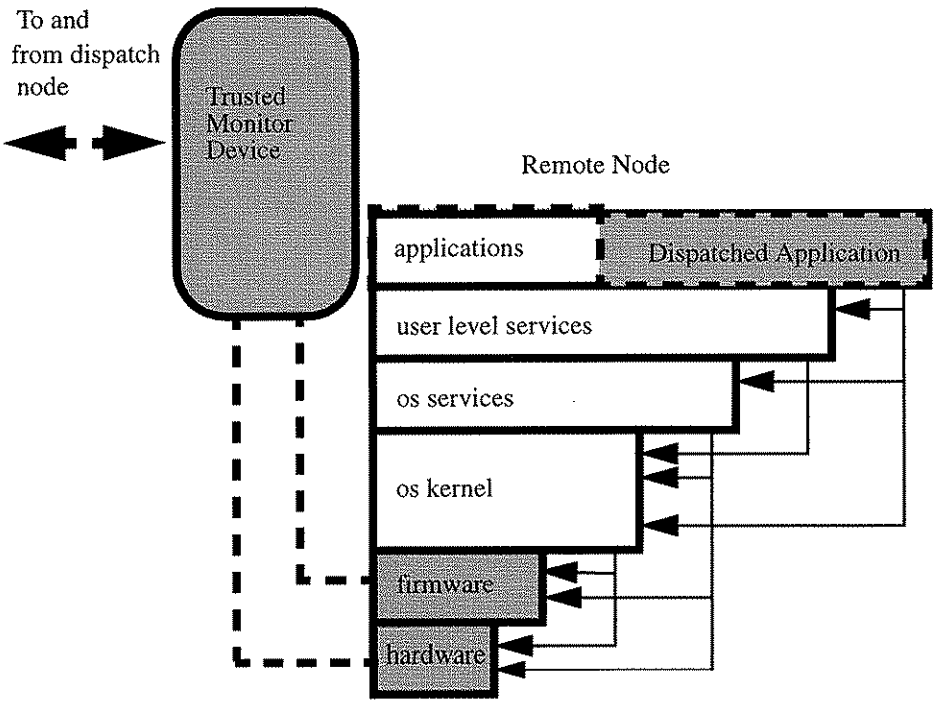


**FIGURE 7.** A trust gap monitoring device is attached to a remote node, where a dispatched application is present. Trusted components appear with shading, while components for which trust-violations must be monitored are clear. Trusted links are drawn with dashed lines. Arrows indicate resource and service dependencies.

levels of the system. In this section we will demonstrate how observation of hardware state by a trusted monitor is sufficient to detect a wide range of higher-level trust faults.

Our approach requires that we extend monitoring of our dispatched application to beyond the application's output signature. We observe the behavior of the remote system more directly. Figure 7 shows the scenario that takes shape. A trusted monitor system is paired with the remote node. The monitor system might be a any form of trustworthy monitoring device. For a start, imagine a specialized embedded device with wires tapped into key components of the remote node system.

While figure 7 depicts the monitor at the hardware level, the monitor could sit at any level that exists before the trust gap where total system state can be observed within the level. Clearly, hardware is a prime example of such a system level.

Through the monitoring of the required trust properties of the dispatched application, we can determine whether, for example, memory protection trust is being maintained. Furthermore, we can determine whether I/O trust is maintained, as well as other components.

### 6.1 Available Data Streams

We can obtain the following data from the system quite easily

1. Contents of memory bus
2. physical I/O ports
3. external cache state and hits

It would be more difficult to tap into the following

1. Processor state
2. internal cache state and hits
3. TLB results and branch predictor results

Even further, we may desire to not only monitor state, but utilize hardware to actively search state. For example, we might want to occasionally take control of the memory bus so as to obtain a snapshot of some data region. (This might allow us to sample the memory bus less frequently than at every instruction.)

### 6.2 Protection Trust Fault Detection through Memory Reference Monitoring: Bus and Cache Snooping

If our trusted monitor is aware of the memory space devoted to our dispatched application, and we had access to the data bus as well as data and instruction caches on the Pentium system, then we could determine whether any other process violated the memory protection requirements of our dispatched process as follows:

- Watch all memory bus traffic and cache request hits and branch table hits.
- Parse the bus traffic as follows:
  - Determine a division of elements of the stream into instruction and data references.
- Use the following rules to detect memory protection trust inversion:
  - If a data access is to a region outside the application data area and the previous instruction reference is to an instruction within the applications instruction area, generate an untrustworthy data reference fault.
  - If a data access is to a region within the application data areaand the previous instruction reference is to an instruction outside the applications instruction area, generate an untrustworthy data reference fault.

There are many complications to this model we have yet to consider. With the presence of growing and shrinking stacks, and dynamic heap usage, this becomes more difficult to clearly define, and we will need additional analysis of these circumstances. Also, virtual memory and run-time segmentation are problematic, as the monitor must know with certainty where the data and instruction memory of our dispatched process reside. If we could have our monitor determine the initial segment and page state of the applications memory, and verify page faults and such, then virtual memory is not a problem. However this may be beyond the scope of a reasonable experimental implementation. Segmented real-address memory is not as difficult a problem to tackle. Linked libraries, such as DLLs are a difficulty, as they append shared address spaces to an applications

instruction stream, and sometimes result in data space sharing. We also require that the monitor is aware of system calls, and of data sharing over system calls. The Pentium processor maintains separate stacks for separate protection levels, and thus data copying to system calls is an issue the monitor would need to be aware of.

Using this technique, we should be aware of any access to our trusted applications data, and which instruction stream is doing it! We even would know whether the instruction stream of our application is executing and getting a fair shake.

As with trust inversion prevention, we showed how memory protection can be rendered trusted. We also showed how clarity for applications can be increased. Just as with prevention, we leave as future work ways in which to show that other required resources, such as task and interrupt management can be trusted. If we have access to memory, it should not be hard to check the hardware-required tables of the remote node, to maintain trust for I/O and interrupts.

## 6.3 Processor State Monitoring

It may also be possible to monitor a system through the processor. We should be able to observe memory requests and register content to understand which process is executing instructions, and where data is being fetched from. However, since we would effectively only gain knowledge of registers over our previous model of bus snooping and cache snooping.

- Is there a reason to desire knowledge of general purpose registers?
- Would knowledge of segment registers, and the processor state flag register, yield additional useful information?

## 6.4 I/O Device Monitoring

There is no reason why we cannot extend our model to observe references to other devices on the computer system. For example, a hard disk drive plays a significant role in the storage of application executables and data files. Just as we monitor the memory bus for operations on memory that violate trust for an application, so can we monitor the commands sent to a hard disk drive, or other storage device, to detect violation of trust on the image of a trusted executable stored at an untrustworthy node.

Furthermore, with detection capability for temporary/permanent storage devices, we may be able to re-incorporate virtual memory into a system model supported by our monitoring device.

At issue is whether our monitor device can perceive enough information about the file system to maintain clarity of trusted application images and data on disk. Clearly, placing trusted applications and files within a fixed track or sector range, and declaring such a range as not to be written to, allows us to enforce write-protection trust for trusted process executable images. Clearly, we would need to have prior knowledge of operating system functions (and there code positions in a fixed kernel) that would be allowed to write trusted executable images into this disk space. Such considerations make a hybrid model of trust preemption with a fixed ROM kernel, and trust detection with a monitoring device, more attractive.

With sufficient knowledge of code that is allowed to read and write certain processes, we can guarantee read and write trust of trusted applications on disk storage.

## 6.5 Further Property Clarity with Active Memory Queries

We have demonstrated how passive monitoring can enforce protection trust for applications in both memory and more permanent storage systems. However, this is just the tip of the iceberg! Another possible monitoring technique is "active" memory probing, rather than just passive snooping of bus and cache activity. By placing the processor of a system and any other bus devices (such as DMA channelers) into a memory wait state, we should be able to take control of the memory bus, for short periods of time, with out greatly affecting processor usage. As we will show, these techniques can help deter intelligent tampering and spoofing. In addition, they further bolster protection and I/O routing by allowing us to probe hardware required data-tables, such as an interrupt vector table or I/O mapping table.

### 6.5.1 Application Specific Property Clarity with Active Memory Queries

We have the opportunity to occasionally scan critical data and code regions of our trusted application in order to make sure they are intact. The following methods for active queries might be combined with the passive techniques given above:

- Provide the monitor device with an image of the application or of critical application regions. Also send the monitor information on how and what parts of these images to verify, and how often.
- Probe specific data regions and check data values to see that they are in accordance with the processes output.

For example, we might occasionally scan a critical loop of our application, and then occasionally check a loop invariant exit data value, or a loop variant data value that has some functional relation to program output. **Active memory monitoring allows us to monitor an application for compliance with static and dynamic properties (application specific properties) that are initially known only to the dispatching node and the monitor.** Complex properties of application provide the opportunity for the dispatcher and monitor to maintain a secret (which properties of the application are being monitored) that would render tampering less effective.

## 6.6 The Resulting Application Specific Detection Capabilities Are Strong

We will now suggest that a combination of passive and active memory/IO bus and cache monitoring will be highly effective deterring tampering and faults with respect to our dispatched application. In fact, it will be highly effective in stopping spoofing and intelligent tampering attacks!

Our trust gap detection technique can be utilized to fight intelligent tampering and spoofing. We can have knowledge of relationships between our dispatched processes internal data state and its output, that only careful program analysis at the remote node could identify. Furthermore, in order to "spoof" output of the trusted application, the spoofer would need to also update the internal data state of the application in accordance with the relationship that we have knowledge of. Considering we have passive

monitoring to detect protection violations, this will be difficult for an adversary to conduct from the outset. In addition, we may utilize the code transform techniques developed by Wang, Hill, Knight and Davidson on our dispatched application to make static analysis (to determine invariant and variant relationships between internal process data state and output) more difficult or effectively useless. An intelligent tampering attack would have to be careful to maintain the relationships between the dispatched application's internal process data and process output. The adversary would be required to:

1. Determine which internal data the monitor is observing and correlating with which output of the application.
2. Get around the monitor's observation of data and code protection for the application.
3. Statically or dynamically analyze the application to determine the internal behavior of the application, and the relationships between internal data and output (to spoof the dispatcher node.) Also, information about important static relationships within the program must be understood in order to spoof a monitor observing application specific properties.
4. Create an alternate form of the application to spoof our dispatching node and monitor, while at the same time, updating the data state (and instruction stream reference) of the actual application. Or, manipulate the application so as to modify its behavior while not disturbing the properties of the application observed by the dispatcher node and monitor.

Since the information from step 1 will be private to the trusted application monitor and monitoring device, this will be difficult for an adversary to obtain. Step 2 will require the adversary to sneak past the monitor's passive observation of instruction and data streams. Step 3 will require the adversary to statically and dynamically analyze the program in an attempt to regain the information that the original author's have about the application. Since the monitor and dispatcher can change which properties of the application they utilize about the application, this will be extremely difficult. If we repeatedly change the internal details of the application and redispatch it, this will be even more difficult for the adversary. Finally, step 4 requires that the adversary perform careful spoofing or tampering of the program.

## 6.7 Sampling and Processing Speed

One issue that must be considered is granularity of sampling over time. It must be assumed that the remote system operates at a particular cycle rate. Our monitor must be fast enough to not miss any cycle-to-cycle state information (in case of very short trust faults.) Our monitor must be at least as fast as, if not several orders of magnitude faster than, the instruction cycle rate of a processor in order to process passive information for the hardware state of the system. Meanwhile, our active monitoring can only occur occasionally, as it must take over the memory bus in order to probe memory. Careful timing with memory bus activity should allow more extensive memory probing with less disturbance of processes' use of memory.

### 6.8 Eliminating the Some Hardware Trust Assumptions

A robust trusted monitoring device with sufficient trusted links to a systems hardware might be able to detect a wide range of hardware faults. Although beyond the scope of this report, such detection capabilities are an advantage for fault-tolerant critical infrastructure concerned with more than just software faults. It should be noted that as we increase the number of trusted links to remote hardware the more robust the monitor device must be made in order to maintain trusted link to all monitored hardware.

### 6.9 Conclusions about Low-Level Trust Detection

We have suggested a model for low-level monitoring of hardware state and an algorithm for detection of memory protection violations from this state. This information allows us to detect any trust violation by levels of a system for which we have no clarity. We leave research into how other hardware components can be utilized to establish trust properties for future research. Furthermore, we demonstrated how active monitoring of memory can further aid detection of trust faults through monitoring of hardware-required data tables in memory. Active monitoring of memory also provides us with the opportunity to scan applications for violations of complex, application specific properties known, initially, to only the dispatcher and the monitor device. If the monitor's activity is not observable by an adversary, then an adversary cannot obtain the secret of which static relationships the monitor is observing. Since the number of application specific properties between code and data are large, and application implementations can be occasionally modified and re-dispatched, we have a technique for monitors to detect attempts to intelligently tamper with, or spoof, application behavior.

## 7.0 Summary and Conclusions

### 7.1 A Summary of Findings

We defined trust as the confidence we have about whether a property is obeyed or not. We defined clarity as the confidence we have about our ability to obtain information about trust of a property. We then showed how traditional system architectures lead to a condition we defined as a trust and clarity gap in a distributed application system, in which applications are dispatched from a trusted source node to an untrustworthy destination node. In our analysis we discovered two techniques to help eliminate the trust gap problem in dispatched applications over a large untrustworthy network.

The first is a form of trust gap prevention. It involved the modification of architecture rules to eliminate some of the cyclical trust dependences in today's hardware designs (in order to simplify the propagation of trust to higher levels on remote machines) where at least the hardware is trusted. The result is increased reliability of key trust requirements for foreign nodes that may be tampered with or are faulty at the software level. We demonstrated how this technique could be applied to the critical trust component of memory protection. We still require that kernel functionality be verified before being physically installed in the kernel execute only code space; but we have reduced the verification burden of the kernel. Future investigation can consider other crucial properties of application trust with relation to specific system trust goals.

The second is a form of trust gap detection. It involves the contractual physical placement of a trusted monitoring device at each remote node to which applications will be dispatched. It allows us to carefully detect memory protection violations, intelligent tampering and spoofing attempts, and potentially such behaviors as stack an buffer overrun attacks. In exchange for this monitoring, the dispatching node gains clarity information from a low-level of the foreign node, where sufficient state is present to detect a large number of trust faults. We can devise small monitors to detect only memory usage trust faults, or more extensive systems to detect trust faults on other devices, such as hard disks, as well. With active memory monitoring, we can even monitor faults in trust of application specific properties. This will help us provide trust against intelligent tampering and spoofing attacks. In general, monitoring for trust faults at a low-level allows us to detect a faults in adherence to a wide range of properties; from general trust properties to application specific properties.

Our detection technique does not depend on understanding "typical" or "atypical" data reference or instruction reference streams. Our monitoring criteria is formally specified by the internal implementation of the trusted application, or in the case of memory protection violations, simple rules of memory accesses by processes. It should be noted that a successful monitor device could eliminate the need for the dispatched application to maintain constant communication with the dispatching node to maintain trust. Thus, while sensors would be effective, "output quiet" actuators would also be instrumentable as trustable on an otherwise untrustworthy software system.

Our prevention and detection techniques defend against system errors, intelligent attacks, and critical software failures. It should be noted that trust gap prevention does not protect against hardware failure. Trust gap detection can defend against some hardware failures, when some still clarity-worthy piece of hardware contradicts the information of a clarity-less faulty hardware device. However, it cannot defend against intelligent tampering or spoofing at the hardware level. Thus our system provides extensive trust for many forms of software failure, and a good selection of non-adversarial hardware failures.

## 7.2 A Hybrid Approach

In combination, our two techniques may provide redundancy in providing application trust. In particular, the change in the nature of protection in hardware that we have suggested, through an isolated kernel space, is more of a revision on the model of trust in a pervasively networked world, than a specific technique for protecting remotely dispatched applications. It provides a new level of robustness at the hardware level, for general trust providing capability of networked machines with dispatched applications.

On the other hand, our monitoring technique provides very specific solutions to the question of trusting applications dispatched to other nodes where software is not trusted. Together, our trust gap prevention can help bolster the ability of the trust gap detection monitor to easily detect trust faults. Also, we have many remaining trust properties that we did not provide proof of solution by to either of these techniques. Depending on the power of trust gap detection, more or less trust ascension techniques may be required in order to make trusted applications relying on untrustworthy system software a reality.

As we saw when examining hard disk trust fault detection, having more knowledge of a fixed operating system makes determination of some faults less expensive.

### 7.3 Conclusions

We have identified two techniques to significantly reduce the problem of trust and clarity gaps for trusted applications distributed to systems with untrustworthy software. The result is the ability to create more responsive, fault tolerant network infrastructures, capable of reacting to non-local faults in a rapid, intelligent manner. In the short term, we have provided a method for bolstering trust of applications in any environment in which hardware is trusted. In an increasingly networked computing environment, issues of software trust will continue to arise, and the nature of who is trusted and who is not, will more readily make apparent the need for trusted software to run on systems where other critical software is inherently untrustworthy.

## 8.0 References

[1] Amoroso E. Nguyen T. Weiss J. Watson J. Lapiska P. Starr T. "Toward an Approach to Measuring Software Trust," Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, USA, 1991, pp 198-218.

[2] Di Vito, BL. Palmquist PH. Anderson ER. Johnston ML. "Specification and Verification of the ASOS Kernel," Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, USA, 1990, pp. 61-74.

[3] Forrest, S. Hofmeyr, SA. Somayaji A. "Computer Immunology" Communications of The ACM, Vol 40, No. 10. October 1997. ACM, 1997, pp 88-96.

[4] Genier G-L. Holt RC. Furkenhauser M. "Policy vs. Mechanism in the Secure TUNIS Operating System," Proceedings. 1989 IEEE Symposium on Security and Privacy. IEEE Computer Society Press, Washington, DC, USA 1989, pp. 84-93.

[5] Hu AJ, Vardi MY. "A Formal Method Experience at Secure Computing Corporation," Computer Aided Verification 10th International Conference, CAV'98, Springer-Verlag, Berlin, Germany, 1998, pp. 49-56.

[6] Johnston M., Stiriou V. "Testing a Secure Operating System," 13th National Computer Security Conference. Proceedings, Vol. 1, pp. 253-65.

[7] Keedy, JL. Vosseberg, K. "Persistent Protected Modules and Persistent Processesses as the Basis for a More Secure Operating System," Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, IEEE Computer Society Press, Los Alamitos, CA, USA, Jan 1992, Vol. 1, pp. 747-56.

[8] Knight JC, Elder MC, Du, X. "Error Recover in Critical Infrastructure Systems," Computer Security, Dependability, and Assurance: From Needs to Solutions. IEEE Computer Society, 1999, pp 49-71.

[9] Knight, JC, Sullivan KJ, Elder MC, Wang C. "Survivability Architectures: Issues and Approaches," Procedings. Darpa Information Survivability Conference and Expose. IEEE Computer Society, 1999, Vol 2, pp 157-71.

[10] Korelsky T. Sutherland D. "Formal Specification of Multilevel Secure Operating System," Proceedings of the 1984 Symposium on Security and Privacy, IEEE Computer Society Press, Silver Spring, MD, USA, pp. 209-18.

[11] Kozyrakis, CE, Patterson DA. "A New Direction for Computer Architecutre Research," Computer, November 1998, IEEE Computing Society Press, Los Alamitos, CA, USA, 1998. pp 24-32.

[12] Sullivan KJ, Knight JC, Du, X, Geist S. "Information Survivability Control Systems," Proceedings. 21st International Conference of Software Engineering. IEEE Computer Society, Los Alamitos, CA, USA, 1999, pp 184-192.

[13] Waldhart NA. "The Army Secure Operating System," Proceedings. 1990 IEEE COmputer Society Symposium on Research in Security and Privacy. IEEE Computer Society Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA, USA 1990, pp. 50-60.

[14] Wismuller, R. Trinitis, J. Ludwig, T. "OCM - A Monitoring System for Interoperable Tools," SPDT 98, Welches OR, USA. ACM, 1998. pp 1-9.