Patterns for Building Dependable Systems with Trusted Bases

Eunsuk Kang and Daniel Jackson, Massachusetts Institute of Technology

We propose a set of patterns for structuring a system to be dependable by design. The key idea is to localize the system's most critical requirements into small, reliable parts called *trusted bases*. We describe two instances of trusted bases: (1) the *end-to-end check*, which localizes the correctness checking of a computation to end points of a system, and (2) the *trusted kernel*, which ensures the safety of a set of resources with a small core of a system.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.2.4 [**Software Engineering**]: Design—*Methodologies*

General Terms: Design, Reliability

Additional Key Words and Phrases: Dependability, trusted bases, design patterns

ACM Reference Format:

Kang, E. and Jackson, D. 2010. Patterns for Building Dependable Systems with Trusted Bases. jn 2, 3, Article 1 (May 2010), 14 pages.

1. INTRODUCTION

Conventional approaches for ensuring system dependability involve *ex post facto* techniques such as testing, inspection, and verification. In practice, it is too costly to rely *solely* on these techniques to satisfy a long list of requirements that the customer demands. Resources are limited, and deadlines loom around the corner. Even with unlimited time and effort, it is unlikely that one will be able to construct a large system that is completely free of faults.

A complementary approach is to prioritize the requirements based on their importance, and structure the system so that the most critical requirements are ensured by small, reliable subsets of the system's parts called *trusted bases*. For example, a microkernel-based OS [Tanenbaum et al. 2006] is designed so that only a small, trusted core of the system is responsible for ensuring its safety and security; an erroneous program in the user space may hinder normal system operations, but should not be able to crash the entire OS. Similarly, it may be desirable to design an electronic voting system so that its vote-tallying software, which is known to be susceptible to various security attacks, cannot compromise election integrity. Instead, third-party auditors are entrusted with examining the election process to ensure the correctness of the election outcome [Rivest and Wack 2008].

This approach is not new. Concepts such as the trusted computing base [Lampson et al. 1992], security kernel [Popek and Farber 1978], and safety kernel [Rushby 1989] are well known in the fields of security and safety-critical systems. We believe that experienced designers already do this, structuring a system so that the most critical components are kept to small subsets of the system's parts. However, despite

This research was supported by the Northrop Grumman Cybersecurity Research Consortium, and the National Science Foundation under grants 0541183 (Deep and Scalable Analysis of Software) and 0707612 (CRI: CRD – Development of Alloy Technology and Materials). Author's address: Eunsuk Kang, Computer Science and Artificial Intelligence Laboratory, 32 Vassar St. Room 32-G708, Cambridge, MA, 02139; email: eskang@csail.mit.edu; Daniel Jackson, 32 Vassar St. Room 32-G704, Cambridge, MA, 02139; email: dnj@csail.mit.edu
PLoP'10, October 16-18, Reno, Nevada, USA. ACM 978-1-4503-0107-7

a wealth of informal knowledge about designing systems with small trusted bases, this approach is far from being in widespread use, partly due to a lack of a systematic framework for documenting and reusing the knowledge. This is where patterns can help.

In this paper, we propose a set of patterns for building a dependable system with trusted bases. We begin by introducing **Trusted Base** as a generic pattern for localizing a critical requirement into a small subset of the system's parts (Section 2). We then describe two specialized instances of the Trusted Base pattern—**End-to-End Check** (Section 3) and **Trusted Kernel** (Section 4). We conclude with a discussion of future directions (Section 5).

2. PATTERN: TRUSTED BASE

2.1 Context

Suppose that we are given a task of building a large, complex software system that interacts closely with the environment. Such a system is *dependable* if it can be justifiably trusted to satisfy its critical requirements [Jackson 2009]. For example, an electronic voting system is dependable if it accurately tallies votes, despite potential attempts for sabotage from a malicious voter. A dependable control system for an automobile contains a safety feature that prevents the car from accelerating out of control. A dependable online banking system protects the customer's information from inadvertently being leaked onto a third-party's hands.

2.2 Problem

How do we build a dependable system? Ideally, we would be able to decompose the system into a set of components, implement them independently, and test them to ensure that they together satisfy all of the requirements. In reality, this process is not so straightforward, and we face a variety of challenges:

- We have a limited amount of development resources to spend, and so it is unlikely that we will be able to test or verify every component to the desired level of coverage.
- Most large, modern systems are not built from scratch; we are often given components to work with, components that we must integrate into the system. Some of these components may be inherently unreliable (e.g. a network) or untrustworthy (e.g. a dynamically loaded module). Some may be closed-source proprietary software with poorly documented interfaces, and so understanding and testing their behaviors may be difficult.
- A dependable system must be resilient to failures; it must satisfy its most critical requirements, *despite* potential failures in some of its components.
- A complex software system with poor separation of concerns suffers from tight coupling between its parts; a failure in one component can propagate to the rest of the system and undermine dependability.

The first two challenges suggest that it is often infeasible to ensure that every component is reliable; we must be willing to accept failures in some parts of the system. As the last two points suggest, the complex nature of software poses a challenge in building a flexible, robust system that can withstand the failures.

Given these challenges, how do we build a dependable system out of potentially unreliable components?

2.3 Solution

Our proposed solution is to *construct the system so that the most critical requirements depend on only small, reliable subsets of the system's parts, called* ***trusted bases***.

A *trusted base* for a requirement is the set of components in the system that are sufficient to establish the requirement, *regardless* of how the components outside the trusted base behave. In a poorly designed

system with high coupling, the trusted bases for the most critical requirements will encompass nearly every component in the system, and so the failure of any single component may have catastrophic effects. A robust design achieves localization of critical requirements into small, separate trusted bases so that a failure in an unreliable component does not impair the most critical services.

Achieving small, reliable trusted bases requires effort throughout the entire development process:

- **Requirements**: Negotiate with the customer to devise an explicit prioritization of requirements, so that when failures occur, the system continues to provide the most critical services, even if other less critical ones fail to hold.
- **Design**: Drive the key architectural decisions with the goal of isolating the components that perform the critical services from the others, so that a failure of a low-critical component does not propagate into the trusted bases. When constrained to work with unreliable or untrustworthy components, we must structure the system so that they lie outside the trusted bases, and their misbehaviors do not undermine the critical requirements.
- **Implementation**: Assign the most skilled, experienced programmers to implement the components inside the trusted bases, and encourage them to use a memory-safe language (e.g. Java over C) and simple algorithms that are easy to understand and program.
- **Testing**: Allocate available testing and verification resources in a *non-uniform* manner, depending on the criticality of the components. This may involve applying more rigorous, expensive techniques (e.g. formal verification) to the components inside the trusted bases, and cheaper, readily available ones (e.g. testing, code inspection, etc.) to the rest of the system. The smaller a component to be tested, the easier it is to argue that the component is correct, and so there is an incentive to keep the trusted bases as small as possible during the design phase.

The Trusted Base pattern can be realized through different architectures and designs. In this paper, we describe two specialized patterns—End-to-End Check (Section 3) and Trusted Kernel (Section 4).

2.4 Consequences

The Trusted Base pattern can be used to simplify and strengthen an argument that the system's satisfies its critical requirements. Having isolated the trusted parts from the untrusted ones, we need to ensure only the reliability of the trusted bases, and be confident that the system will satisfy its most critical requirements.

The pattern is not without liabilities. The customer is encouraged to prioritize requirements based on their criticality, and design decisions are driven by this prioritization, and so this inevitably leads to sacrifices in one aspect of the system for another. For example, if safety is the paramount concern of the customer, this may require restricting the functionality of the system to make it simpler and easier to test. A high-security system often includes authentication and monitoring mechanisms, which tend to hinder the system performance.

Additional mechanisms may be necessary to ensure the separation of the trusted parts from the untrusted ones, depending on the expected failure modes of the components. Usually, this involves defining a narrow, controlled interface for each trusted base, and restricting interactions with the untrusted clients. A stronger encapsulation mechanism (e.g. hardware-based protection) may be required to guard against malicious components that deliberately attempt to take down the trusted base. Implementing and testing these mechanisms can be difficult, and often require more effort than a conventional design of the system. But this is part of the cost of dependability.

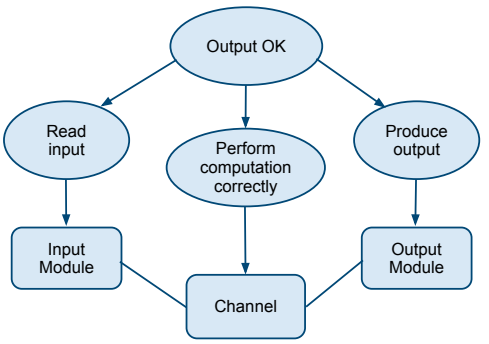

Fig. 1. Property-part diagram for an input-output system. A box represents a part, a circle a property, an arrowed edge a dependency of a property on a part or another property, and a straight edge an interaction between two parts.

2.5 Known Uses

There are a number of known systems that exploit the notion of trusted bases; these are described in the Known Uses sections of the specialized patterns (Sections 3.6 and 4.6).

2.6 Related Patterns

Hanmer's set of patterns for fault-tolerant designs [Hanmer 2007] addresses challenges in building dependable systems, and is most closely related to our work. Although he does not explicitly discuss the notion of trusted bases, some of his patterns, such as Error Containment Barrier and Quarantine, are intended to localize a critical property into a small part of the system, similar to the goal of the Trusted Base pattern.

A number of patterns for building secure systems [Fernandez and Pan 2001; Yoder and Barcalow 1997] have been suggested. Many of these patterns—involving access control and authentication—address the challenge of assigning different levels of trust to participants in the system, and isolating critical, trusted parts from the untrusted ones. We believe that these patterns can be used to implement trusted bases. However, the Trusted Base pattern is not restricted to security, and is intended to cover a wide range of dependability requirements in general (e.g. safety, reliability, etc.).

3. SPECIALIZED PATTERN: END-TO-END CHECK

3.1 Context

Consider a system that accepts an input, performs a computation as a series of processing steps on the input, and produces an output. Figure 1 is a *property-part diagram* [Jackson and Kang 2009] that shows the structure of such a system. A property-part diagram is an extension of a traditional module dependency diagram that shows not only *parts* of the system, but also *properties*, which depend on parts or other properties. In this diagram, the system consists of a module that reads the input (*input module*), a *channel* as a group of modules that carry out some computation on the input and transfer data from one end to another, and finally, a module that takes the result at the end of the channel and outputs it to the user (*output module*).

3.2 Problem

How do we argue that the output is correct with respect to the customer's requirement? One solution is by ensuring that every component carries out its task correctly. Then, the requirement that the actual

output match the expected value ("Output OK") depends on every component in the system behaving as expected and satisfying the property that is assigned to it.

Starting from a property, the set of all parts that are reachable through the edges corresponds to the parts that are responsible for establishing the property. If any of these parts fails, then the property may no longer hold. This set is exactly the *trusted base* for the property. For example, in Figure 1, the trusted base for the requirement "Output OK" includes every component in the system.

In practice, it may be too difficult to achieve high confidence in such a system. Some of the components in the channel may be inherently unreliable, and behave in ways that are poorly understood or difficult to enforce. For example, let us assume that we are given the task of building a file transfer system over an unreliable network medium. Failures are common in such a network; it may occasionally drop packets, deliver them out of order, or corrupt them. We may also be constrained to use commercial-off-the-shelf (COTS) components as a part of the channel. The source code for such a component may be unavailable, and the interface specification poorly defined. As a result, we may not fully understand the behaviors of the component.

How do we ensure the integrity of the outcome from a series of computational steps that are performed by potentially unreliable components?

3.3 Solution

The proposed solution is to perform an *end-to-end check* to ensure that the outcome of computation is correct. A successful application of this pattern involves the following tasks throughout the development:

- **Requirements**: We modify the critical requirement for the new system from "Output OK" to "Output OK *or* failure detected". The end-to-end pattern provides only a way to detect potential failures in the channel; it does not guarantee that the system will always produce an output. If, for example, the network in a file transfer system shuts down, then there is no way to transfer the file across to destination host, until the network is back up again. In this sense, the pattern ensures not that "good things always happen", but only that "bad things never happen", which may be considered more critical than the former (as negotiated with the customer).
- **Design**: We add a component called *checker* to verify the integrity of the output using a *predicate* that relates each input value to the corresponding expected output. If the actual output turns out to be incorrect, then the checker signals a possibility of a channel failure, and warns the user not to rely on the output. The user may then attempt a re-try, or report the problem to the system manager. If the checker returns OK, then the user may safely regard the output to be correct.
- **Implementation**: When given multiple available options, we implement the simplest, most widely used algorithm for implementing the checker. For example, there are a number of different ways for encoding the content of a file; simpler the encoding, higher the confidence that the implementation of the checker will be correct.
- **Testing**: We can consider the channel to be non-critical, and focus available testing effort to ensure that the checker and the input and output modules are implemented correctly.

The property-part diagram in Figure 2 depicts the structure of a system that employs the end-to-end check. The pattern achieves a smaller trusted base for the integrity requirement, which no longer relies on the property of the channel. The requirement will still hold, despite a failure in the channel, as long as the components in the trusted base—the checker, and the input and output modules—behave as expected.

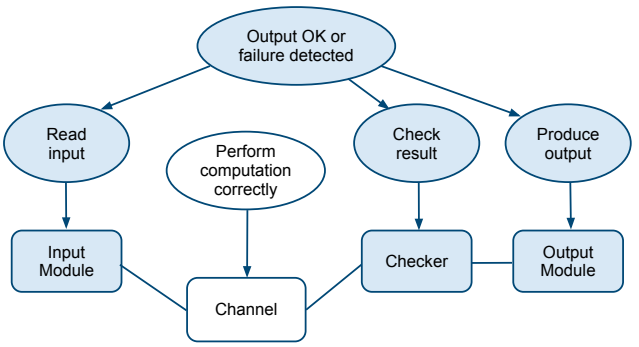

Fig. 2. Property-part diagram for the input-output system with an end-to-end check. As a consequence of the end-to-end check, the requirement no longer depends on the channel performing the computation correctly.

3.4 Example

The seminal paper [Saltzer et al. 1984] by Saltzer and his colleagues introduces the *end-to-end principle*, and uses a reliable file transfer problem as the leading example. This principle has since become the underlying backbone for the communication protocols on the Internet.

The problem is to design a system that transfers a file on one host machine to a destination host over a potentially unreliable network. The network and the file systems may experience a variety of failures, such as data corruption, dropped packets, and packets being delivered out of order. There are different approaches to ensure the reliability of file transfer. One approach is to build extra fault-mechanisms into the network and the file system to increase their reliability. But this may be too costly, as Saltzer and his colleagues argue.

An alternative approach is to move the network and the file systems out of the trusted base, so that the reliability of the transfer can be ensured by the end points of the system alone. Figure 3 shows the property-part diagram for the file transfer system with the end-to-end check. The sender application, taking the role of the *input module*, reads the file from the file system, computes the hash on the file, and sends it over the network, along with the data packets that represent the content of the file. The receiver application, taking the role of both the *checker* and the *output module*, reads the packets off the network and writes the file to its file system. Then, it reads back the same file from the file system and computes a hash on the file. Two hash values now reside on the receiver application; the one received over the network, and the one computed on the receiver host. The predicate that determines the outcome of the transfer is simply the comparison of the two hash values; if they are equal, then the file has been successfully transferred.

As a consequence of applying the pattern, the network and the receiver file system are no longer part of the trusted base. Note that the sender file system is still within the trusted base; if the file system fails and the sender application reads out the file with a bad content in the first place, then the transfer will be unsuccessful, and worse, this failure will go undetected.

3.5 Consequences

The End-to-End Check pattern achieves the goal of pushing the unreliable channel out of the trusted base, but it is not without liabilities.

While the integrity requirement no longer depends on the channel, other parts of the system may need to fulfill more responsibilities. First of all, in order to check the integrity of a particular output value, the checker needs the following three pieces of information: the input value, the actual output

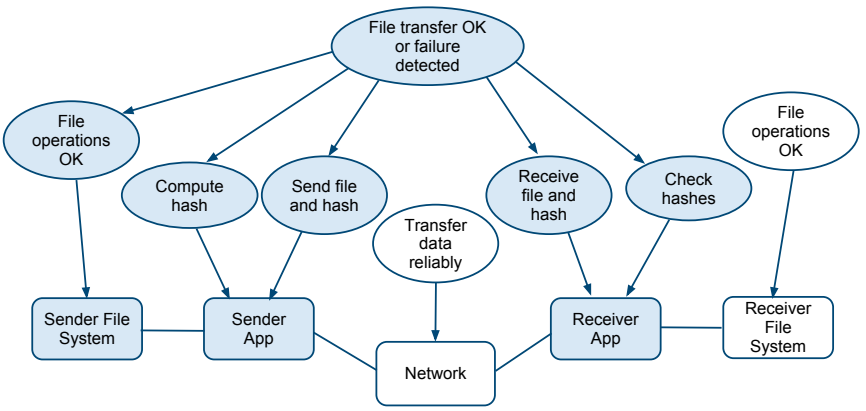

Fig. 3. Property-part diagram for the file transfer system.

value, and the predicate that relates the input to its expected output. To do this, we need to extend the functionality of the input module to pass an encoding of the input value directly to the checker. For example, in the file transfer system, the sender application must compute a hash of the input file and transmit it to the receiver, along with the input file blocks.

The customer may need to accept a lower level of performance as a trade-off for greater reliability. In the file transfer example, the pattern deliberately treats the network as unreliable; this can sometimes lead to undesirable effects on the system performance. For instance, when the checker detects a failure, it may prompt the sender host to resend the entire file from the start. If network failures are frequent, the overhead of multiple retries can add up to a significant cost. Thus, while the network lies outside the trusted base, for performance reasons, it may be a good idea to utilize available resources to ensure that the network is reliable.

3.6 Known Uses

Some of other known uses of end-to-end checks are as follows:

3.6.1 *Voter-Verifiable Voting Systems.* In recent years, electronic voting (e-voting) systems have been gaining popularity in the United States and other parts of the world. The proponents of e-voting point out potential cost saving of automated vote tallying, and greater accessibility for persons with physical conditions (e.g. visual impairment). However, many security experts have expressed concerns about the reliability of e-voting systems, especially due to the dependence on the voting machine, which has been shown to be susceptible to various security flaws [Kohno et al. 2004]. A erroneous or malicious piece of vote-tallying software can drop or modify votes, or tamper with the final tally.

An *end-to-end verifiable* voter verifiable (E2E) system is a type of voting systems in which voters are allowed to verify that their votes were included in the final tally. Researchers have developed several E2E voting systems [Chaum et al. 2008; Popoveniuc and Hosp 2006; Ryan and Schneider 2006] as alternatives to existing e-voting systems. An E2E system shifts the burden of ensuring election integrity from the voting machine to voters and auditors, who are now responsible for checking that the ballots are correctly tallied as cast. The argument for why voters and auditors are more trustworthy than a voting machine involves social factors; for example, auditors are hired by different political parties, and so the likelihood of collusion in the auditing process is small. An E2E system also relies on the assumption that voters are responsible citizens with an interest in preserving election integrity.

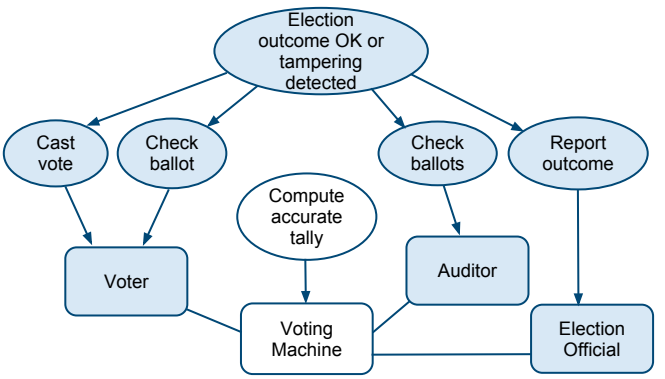

Fig. 4. Property-part diagram for an end-to-end voting system.

Figure 4 shows the property-part diagram for an E2E voting system. This example demonstrates that components in a trusted base may include not only hardware or software that we build, but also parts of the environment, such as voters in an election. This is an important but often overlooked point that has implications on the task of the designer, who must clearly articulate assumptions about the behaviors of the environmental entities and design the system to accommodate their participation.

3.6.2 *Constraint Solvers.* Modern constraint solvers can tackle large problems that were unfathomable a decade ago. They are increasingly being used in a wide variety of applications, such as program analysis, verification, and testing. But how can we be confident that an instance found by a constraint solver is a valid solution? How do we trust a third-party solver?

The answer is that we do not need to trust one. Consider the property-part diagram in Figure 5. When a constraint solver returns an instance as a possible solution to a formula F (in form of variable assignments), we can simply plug the values back into the variables in F. If the formula evaluates to true, then we can be confident that the instance is a valid solution. Otherwise, the constraint solver might be faulty; we can report the problem to the developers of the constraint solver, or simply switch to another solver. For example, the Alloy Analyzer [Jackson 2006] relies on a set of third-party SAT solvers, and performs this check during every analysis run.

This example is particularly nice, because checking the validity of a solution is much easier (in polynomial time) than computing the solution (in NP-complete). Blum and Kannan's work on self-checking programs provides a theoretical discussion of the kinds of computations for which an "easy" checker is available [Blum and Kannan 1995].

3.6.3 *Runtime Assertions.* One of the simplest and most widespread use of the end-to-end check is runtime assertions in programming languages. We can insert an assertion at the end of a module or a computational block to check whether a desired condition holds. If the assertion evaluates to false, then we can examine the program for possible bugs; if it evaluates to true, then we can be confident that, for this particular execution, the program correctly carries out the computation. Again, the pattern is applicable only if we can find a succinct condition (i.e. the predicate) that we can insert inside the assertion.

3.7 Related Patterns

In his set of patterns for fault-tolerant designs [Hanmer 2007], Hanmer introduces *Correcting Audits* as an architectural pattern for checking integrity and well-formedness of data. The End-to-End Check

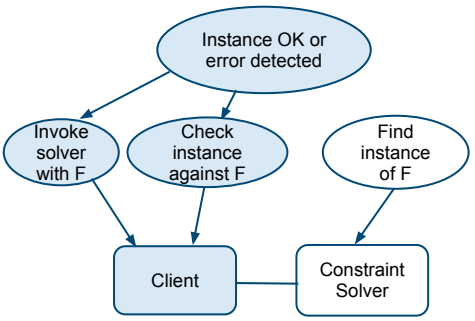

Fig. 5. Property-part diagram for a client-solver interaction.

pattern can be regarded as a specialization of Correcting Audits in which the end points of the system are assigned the responsibility of checking the system output. Hanmer's Checksum pattern can be used to implement the end-to-end check pattern to detect errors in transmission, as we saw in the file transfer example.

The CHECKS pattern language by Cunningham [Cunningham 1995] introduces a set of patterns for validating the user's input for well-formedness (e.g. Exceptional Value), and detecting failures (e.g. Echo Back or Diagnostic Query) that may arise from erroneous input. The end-to-end check pattern is used to validate the relationship between an input-output pair, not the input value itself, and so the input module from Figure 2 may be augmented with the CHECKS patterns to ensure the validity of the user's input before the computation is carried out.

4. SPECIALIZED PATTERN: TRUSTED KERNEL

4.1 Context

Consider the task of building a system that controls a set of *resources*, and provides its *participants* with *services* for accessing these resources. Examples of resources include memory space in an operating system, user profile data on a social network site, and flight plans in an air-traffic control system.

4.2 Problem

On top of the basic functionality for accessing the resources, the system may be required to ensure critical properties about them. For example, a user on the social network should not be able to gather sensitive data about other users, unless given an explicit permission to do so. Inconsistent flight data may cause catastrophic accidents on the runway, and so the air traffic controller must correctly merge real-time updates from multiple flight operators. An OS must guard against a buggy application process that performs dangerous operations and crashes other programs (and in the worst case, the entire OS, itself).

Figure 6 shows a property-part diagram for a design of a system that manages two types of resources, X and Y, with two participants A and B (in general, the system may contain any number of participants and resources). Each resource is represented by an interface that provides functionality for retrieving and modifying the resource. The requirement "Resources safe" holds true as long as each resource interface processes requests correctly, and each participant does not perform unsafe operations on the resources that it accesses. The trusted base for the requirement includes all of the participants and the resource interfaces.

How much confidence can we place on this system? The major challenge arises from the difficulty of ensuring that each participant satisfies its property. Often with systems such as this one, not all

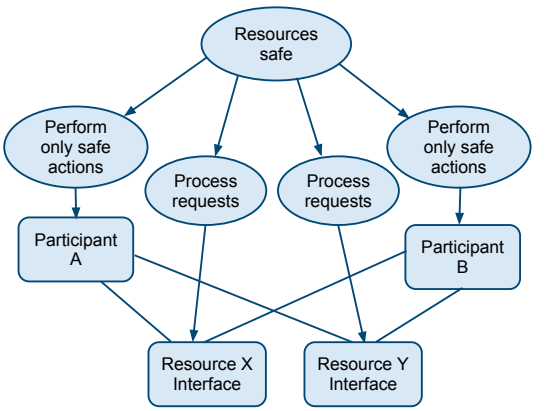

Fig. 6. Property-part diagram for a system that provides a set of resources, which are accessed by various participants.

participants may be known at the time of the development. As a result, it might not be always possible to inspect or test the code inside a participant to ensure that it will behave as expected. For example, an OS in an all-purpose computer must be able to support any number of user applications. Many of these applications are downloaded from the web, and some of them could be malicious software designed to damage the system resources.

Even with absence of malicious components, due the sheer size and number of the participants in a complex system, it may not be possible to ensure that every participant will behave well and share the resources gracefully with others. Most likely, one or more bugs will manage to slip by our testing effort, and cause unexpected behaviors in the participants.

How do we ensure the safety of system resources while providing services to potentially buggy or malicious participants?

4.3 Solution

Our proposed solution is to encapsulate the control of resource access requests into a small component called the *trusted kernel*. To successfully apply this pattern, we must carry out the following tasks throughout the development:

- **Requirements**: We prioritize the safety of system resources to be the primary concern of the system. The trusted kernel is intended to ensure only that the resources remain safe in presence of faulty or malicious participants. Other system requirements, such as functionality, performance, and availability, are still dependent on the correct behaviors of the participants.
- **Design**: We place the trusted kernel as the interface between all of the participants and the resources, so that every request by a participant must go through the kernel. It contains the logic and information to decide whether or not a particular request has potential to damage the system resources, and disallows the request if it is deemed so. For example, the trusted kernel in an OS may deny a request by a user process to write to a memory location outside its allowed boundary.
- **Implementation**: Even a single failure in the trusted kernel may undermine the safety requirement, and so we must employ best available programming practices to ensure that the trusted kernel is implemented correctly. This may involve using a memory-safe language such as Java or Ada, and minimizing the use of risky language features such as dynamic memory allocation.

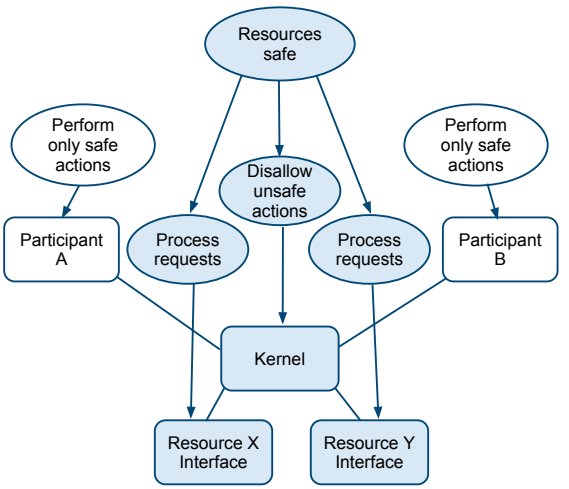

Fig. 7. Property-part diagram for a system with a trusted kernel, which controls how the resources are accessed and prevents potentially dangerous actions by participants.

- **Testing**: By keeping the implementation of the kernel small and simple, we may subject it to extensive testing and formal verification, which can yield high confidence in its correctness.

The property-part diagram in Figure 7 shows the design of the system that is structured with a trusted kernel. One immediate difference from the previous design in Figure 6 is the absence of the dependency of the requirement "Resources safe" on the properties of the participants. A malicious or buggy participant may make a dangerous request that could lead to undesirable effects, but the resources will remain in a safe state, as long as the trusted kernel is able to detect and deny the request. The trusted base for the requirement now includes only the trusted kernel and the interfaces to the resources.

4.4 Example

A *microkernel* refers to software designed to provide applications (e.g. the file system, device drivers, etc) with basic services for accessing system resources, such as memory and CPU scheduling. A microkernel draws a stark contrast with monolithic designs of OS, which allows these applications to freely access the resources. This means that an error in a file system or a device driver can potentially corrupt parts of critical memory, and in the worst case, crash the entire OS.

Figure 8 shows a property-part diagram for a simplified microkernel-based OS. By taking away the privilege of direct resource access from the applications, the microkernel reduces the amount of code that need to be trusted in order to ensure the safety of the system. Several researchers [Klein et al. 2009; Tanenbaum et al. 2006] have advocated the benefits of microkernels in providing greater reliability over monolithic OSes.

Every design decision comes with trade-offs. In an microkernel-based system, every request made by an application must first go through the kernel; this may add overhead to the application performance. Applications in the monolithic OS can arguably perform better, since they can directly obtain the resources without being restricted by the kernel. In fact, poorer performance has been one of the main criticisms against microkernel-based OSes.

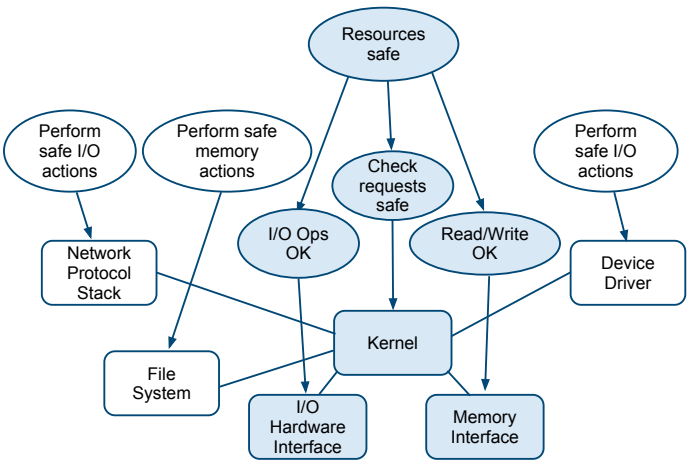

Fig. 8. Property-part diagram for a microkernel-based OS.

Then, how do we choose between the two types of operating systems? The decision, again, comes down to the customer's prioritization of requirements. Microkernels are recommended for use in systems where safety or security is the primary concern. On the other hand, if the user desires optimal performance above all, then a monolithic OS may be more suitable.

4.5 Consequences

The kernel must have sufficient knowledge to decide whether or not a particular request by a participant is dangerous. This may require keeping additional information or states inside the kernel. For example, in order to control access to the memory, an OS kernel must keep track of the portions of the memory addresses that are occupied by different processes. In a high-security system with multiple levels of privileges, the trusted kernel may use an access control list (ACL) [Pfleeger and Pfleeger 2007] to grant or deny access to a particular user. Keeping these kinds of information up-to-date is a challenging problem on its own.

As we observed in the discussion of microkernels in the previous section, placing restriction on resources, and requiring every request to be checked by the trusted kernel, can hinder the system performance. The customer must be willing to accept the lower performance as a trade-off for greater reliability.

4.6 Known Uses

Some of other known uses of trusted kernels are as follows:

- **Security kernel and trusted computing base (TCB)**: A *security kernel* [Popek and Farber 1978] refers to a small core of hardware and software components that are responsible for enforcing security policies of a system[1]. A *TCB* [Lampson et al. 1992] generalizes on this concept to include, as part of the trusted components, software processes that may reside outside the kernel; consequently, a TCB can also be used to check end-to-end security properties that extend beyond the kernel boundary. Security kernels and TCBs have been used in a variety of criticial systems; examples include EROS [Shapiro et al. 1999] and seL4 [Klein et al. 2009].

[1]A security kernel differs from a microkernel in that the latter usually protects OS resources (such as memory or CPU), while the former is mainly concerned with information privacy. However, the two terms are sometimes used interchangeably.

- **Information flow in organizations**: In a typical organization, different types of information are shared among different groups of people. For example, a government agency such as the NSA may restrict the access of highly sensitive information only to a small group of individuals who have the appropriate level of clearance. Of course, these trusted individuals must be reliable enough to protect the information; it takes just one devious or irresponsible individual to leak the information and compromise the privacy.
- **Trusted core in theorem provers**: Theorem proving systems such as Coq [Bertot and Castéran 2004] and HOL [Gordon and Melham 1993] construct proofs of theorems by performing a series of (semi-) automated deductive steps. Many of these systems are complex pieces of software themselves, and possibly contain faults. How can we be confident that a proof returned by a theorem prover is actually sound? A family of provers, referred to as *LCF* [Gordon et al. 1979] provers, are designed to guarantee the soundness of every proof that the tool produces. In an LCF prover, logical statements are represented as a datatype, and inference rules as datatype operations that take one or more statements and produce another; the prover also contains a small set of statements that correspond to axioms. A new statement (i.e. a theorem) can be created only by applying a primitive inference rule to an existing statement. Thus, as long as the set of axioms are consistent, and the core inference rules are implemented correctly, any theorem that the tool claims to have proven must be logically valid.

4.7 Related Patterns

At the architectural level, the Microkernel pattern [Buschmann et al. 1996] has been suggested as a mechanism to encapsulate the basic functionality of an OS. Our Trusted Kernel pattern generalizes beyond an OS architecture, and is applicable in any context that calls for protection of resources. At the same time, Trusted Kernel is a specialization of Reference Monitor [Fernandez 2002] and System Monitor [Hanmer 2007], which may be placed at multiple boundaries across the system to control access, instead of being encapsulated in the core.

A variety of security patterns have been proposed in the past. Among these, patterns for implementing *access control* [Fernandez and Pan 2001; Priebe et al. 2004] can be used at the trusted kernel boundary to manage how resources are accessed.

At the implementation level, the Facade pattern [Gamma et al. 1995] may be used to wrap a set of related resources, and provide an interface that controls access to them.

5. FUTURE WORK

We believe that these patterns are just the beginning of a series of patterns for designing dependable systems with trusted bases. Our conjecture is that many successful system designs employ techniques for reducing trusted bases. In future, we plan to study a variety of such systems, and devise a more extensive catalog of patterns with trusted bases.

Acknowledgments

We would like to thank António Rito Silva for being a wonderful shepherd of our paper, and the members of the Writers Workshop group— Cédric Bouhours, Robert Hanmer, Kiran Kumar, Ernst Oberortner, and Shivanshu Singh—for their detailed feedback, which greatly helped improve the paper. Thanks also to Joseph Near, Aleksandar Milicevic, and Sasa Misailovic, who offered helpful comments on an earlier draft.

REFERENCES

BERTOT, Y. AND CASTÉRAN, P. 2004. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer-Verlag New York Inc.

BLUM, M. AND KANNAN, S. 1995. Designing programs that check their work. *J. ACM 42,* 1, 269–291.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. A System of Patterns: Pattern-Oriented Software Architecture.

CHAUM, D., CARBACK, R., CLARK, J., ESSEX, A., POPOVENIUC, S., RIVEST, R. L., RYAN, P. Y. A., SHEN, E., AND SHERMAN, A. T. 2008. Scantegrity ii: End-to-end verifiability for optical scan election systems using invisible ink confirmation codes. In *EVT*.

CUNNINGHAM, W. 1995. The checks pattern language of information integrity. In *Pattern languages of program design*. ACM Press/Addison-Wesley Publishing Co., 155.

FERNANDEZ, E. 2002. Patterns for operating systems access control. In *Conference on Pattern Languages of Programs*.

FERNANDEZ, E. AND PAN, R. 2001. A pattern language for security models. In *Conference on Pattern Languages of Programs*.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-wesley Reading, MA.

GORDON, M. AND MELHAM, T. 1993. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press New York, NY, USA.

GORDON, M., MILNER, R., AND WADSWORTH, C. 1979. Edinburgh LCF: a mechanized logic of computation, volume 78 of Lecture Notes in Computer Science.

HANMER, R. S. 2007. *Patterns for Fault Tolerant Software*. John Wiley and Sons.

JACKSON, D. 2006. *Software Abstractions: Logic, language, and analysis*. MIT Press.

JACKSON, D. 2009. A direct path to dependable software. *Commun. ACM 52,* 4, 78–88.

JACKSON, D. AND KANG, E. 2009. Property-part diagrams: A dependence notation for software systems. In *ICSE '09 Workshop: A Tribute to Michael Jackson*.

KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. 2009. sel4: formal verification of an os kernel. In *SOSP*. 207–220.

KOHNO, T., STUBBLEFIELD, A., RUBIN, A. D., AND WALLACH, D. S. 2004. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*. 27–.

LAMPSON, B. W. 1984. Hints for computer system design. *IEEE Software 1,* 1, 11–28.

LAMPSON, B. W., ABADI, M., BURROWS, M., AND WOBBER, E. 1992. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst. 10,* 4, 265–310.

PFLEEGER, C. AND PFLEEGER, S. 2007. *Security in Computing*. Prentice-Hall.

POPEK, G. AND FARBER, D. 1978. A model for verification of data security in operating systems. *Communications of the ACM 21,* 9, 737–749.

POPOVENIUC, S. AND HOSP, B. 2006. An introduction to Punchscan. In *IAVoSS Workshop On Trustworthy Elections (WOTE 2006)*. 28–30.

PRIEBE, T., FERNÁNDEZ, E. B., MEHLAU, J. I., AND PERNUL, G. 2004. A pattern system for access control. In *DBSec*. 235–249.

RIVEST, R. L. AND WACK, J. P. 2008. On the notion of "software independence" in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 366,* 1881, 3759.

RUSHBY, J. 1989. Kernels for safety. *Safe and Secure Computing Systems*, 210–220.

RYAN, P. AND SCHNEIDER, S. 2006. Prêt à Voter with re-encryption mixes. *Computer Security–ESORICS 2006*, 313–326.

SALTZER, J. H., REED, D. P., AND CLARK, D. D. 1984. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2,* 4, 277–288.

SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. 1999. Eros: a fast capability system. In *SOSP*. 170–185.

TANENBAUM, A. S., HERDER, J. N., AND BOS, H. 2006. Can we make operating systems reliable and secure? *IEEE Computer 39,* 5, 44–51.

YODER, J. AND BARCALOW, J. 1997. Architectural patterns for enabling application security. In *Conference on Pattern Languages of Programs*.

PLoP'10, October 16-18, Reno, Nevada, USA. Copyright 2010 is held by the author(s). ACM 978-1-4503-0107-7