Dependability Arguments with Trusted Bases

Eunsuk Kang and Daniel Jackson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA USA
{*eskang, dnj*}*@mit.edu*

***Abstract*—An approach is suggested for arguing that a system is dependable. The key idea is to structure the system so that critical requirements are localized in small, reliable subsets of the system's components called *trusted bases*. This paper describes an idiom for modeling systems with trusted bases, and a technique for analyzing a *dependability argument*—the argument that a trusted base is sufficient to establish a requirement.**

***Keywords*-dependability; requirements and design; trusted bases; formal modeling and analysis;**

I. Introduction

Traditional approaches to dependability focus on *ex post facto* methods, such as verification, testing, and inspection. Despite advances in these methods, achieving dependability in a complex system still poses a formidable challenge. Testing and inspection yield limited confidence, and verification is too costly unless carefully focused in scope.

An alternative approach is to design the system so that its dependability is guaranteed largely by construction, with only a limited amount of analysis required to ensure that the design has been faithfully realized in the code. Designs that follow this approach use techniques such as isolating critical components [1], employing interlocks to guard against dangerous actions [2], and performing end-to-end checks [3] to ensure data integrity. Our premise is that these techniques have a common goal: reducing the size of a *trusted base*, which comprises the parts of the system that are responsible for fulfilling a critical requirement.

The idea of localizing critical properties for dependability is not new. The *trusted computing base* (TCB) [4], [5] and the *security kernel* [6] are well-known notions in the security literature. Rushby proposed an analogous version for safety-critical systems, called the *safety kernel* [7], which ensures safety by encapsulating the control of dangerous actions. The value of this idea has been recognized more widely [8], beyond the realm of security. But although many software engineers are familiar with it, few seem to apply the idea systematically and to exploit it in all the contexts in which it might be useful, in part due to a lack of ways to document and reason about trusted bases.

Furthermore, existing approaches to building TCBs and kernels focus on the design of software, and take place after the requirements stage. Often, though, this will be too late. The satisfaction of a requirement usually relies on assumptions about the environment, in addition to the behaviors of software components. The trusted components may include not only *machines* (designed components, such as software modules) but also *domains* (given, environmental components, such as human operators, business processes, and the physical world). Therefore, establishing trusted bases for critical requirements should begin as early as the requirements analysis phase. We prefer to use the term *trusted base*, instead of *trusted computing base*, to make clear the important role of non-computer domains in dependability.

In this paper, we propose a framework for designing a system for dependability with trusted bases. We present an idiom for modeling a system so that the relationships between requirements and trusted bases are made explicit. We describe a technique for analyzing a *dependability argument*—the argument that a trusted base is alone sufficient to establish a requirement; that is, it is not missing a component that is also necessary for the requirement to hold.

The paper begins with an outline of our proposed approach to achieving dependability with trusted bases (Section II). We then present the idiom for modeling systems with trusted bases (Section III), and a technique for analyzing a dependability argument (Section IV). We report on a case study of two electronic voting systems (Section V): an optical scan system and the *Scantegrity* system [9], which is intended to yield a strong guarantee that the failure of an unreliable component (such as a scanner) does not compromise the integrity of the election. We discuss the related work (Section VI), and conclude (Section VII).

II. Overview of the Approach

In this section, we provide an informal discussion of our proposed approach to achieving dependability with trusted bases. As a running example throughout this section, we use the problem of reliable file transfer, based on the description in the seminal paper by Saltzer, Reed, and Clark [3]. A file resides on the disk of a computer, which is linked to another computer through a potentially unreliable communication network. The requirement is to faithfully copy the file from the sender to the receiver. The software engineer's task is to design a file transfer protocol (FTP) application to provide the transfer capability.

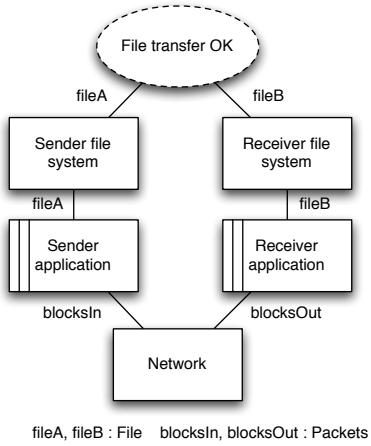


Figure 1. A problem diagram for a design of the file transfer system. Striped boxes represent machines, boxes domains, and a dotted oval a requirement. Labels on the edge between two domains or machines represent *shared* phenomena; labels on the edge from a requirement to a domain represent *referenced* phenomena.

We begin the section by introducing the *problem frames* [10] approach as the underlying framework for structuring a system and its requirements. We define the notion of a trusted base, and describe a particular design of the file transfer system, whose trusted base includes every component in the system. We then present an alternative design, where the trusted base no longer depends on the unreliable network, and therefore, is arguably more dependable than the previous design.

A. Problem Frames

In our approach, we use Michael Jackson's *problem frames* [10] as the underlying modeling framework to articulate the structure of a system and its relationships to requirements. A hallmark of problem frames is the division of the system into *application domains*—parts of the world that already exist—and *machines*—components to be built in order to solve the problem.

Figure 1 shows an example of a *problem diagram*, which illustrates how machines interact with one or more domain to satisfy a requirement. In this particular diagram, the two file systems and the network are examples of domains, because they are given as parts of the problem. Sender and receiver FTP applications are machines that will be built to interact with these domains in order to carry out the file transfer.

Machines and domains interact by sharing *phenomena*—for example, files stored on the file systems, and data packets that are generated by the sender application and sent over the network. In Figure 1, the sender file system shares *fileA* with the FTP application, which then splits the file into blocks of data packets. In turn, the application shares the packets (*blocksIn*) with the network by requesting their transmission.

Each domain has an associated set of *assumptions*. We might assume, for example, that the read and write operations on the file systems work correctly; or that the communication channel in the network is reliable (that is, does not corrupt or drop packets) and secure (resilient against security attacks).

Each machine has a *specification* that it must fulfill. The sender application's specification, for example, might dictate that it correctly read the file from the disk, split its content into data packets, and pass them onto the network for transmission. The receiver application's specification might say that it receives the packets off the network and writes them to disk appropriately.

A requirement is a problem to be solved to satisfy the customer. Usually, the requirement is expressed in terms of phenomena that appear within domains. That the customer's requirements often exist within the environment, not within machines, is one of the key insights of the problem frames approach. For example, the requirement for the reliable file transfer expresses a desired relationship between the contents of the files on the sender and receiver file systems; it does not mention anything about the behaviors of the file transfer applications. It is the requirements analyst's task to derive machine specifications that are sufficient to establish the requirement, in conjunction with domain assumptions[1]. The specifications of the sender and receiver applications, together with the assumptions about the file systems and the network, must satisfy the requirement that the content of the file on the sender matches that of the file on the receiver.

B. Trusted Bases and Dependability Arguments

A *trusted base* for a requirement is the set of all domains and machines that are responsible for establishing the requirement, regardless of how the components outside the trusted base behave.

Consider the design of the file transfer system in Figure 1. We identify the trusted base for the reliable file transfer requirement as the set that consists of: the sender and receiver file systems, the sender and receiver FTP applications, and the network. Then, the argument for the satisfaction of the requirement, also called the *dependability argument*, can be stated as follows:

> If the file operations work correctly, the network transmission is reliable and secure, and the FTP applications are implemented to their specifications, then the system will guarantee reliable file transfer between the two computers.

The analysis of a dependability argument involves two aspects of a trusted base:

[1]Beyond the scope of this paper, some of the techniques for deriving specifications from requirements are discussed in [11], [12].

1) Sufficiency: The proposed trusted base must be sufficient for the desired requirement; that is, if every domain in the trusted base satisfies its assumptions, and every machine satisfies its specification, then the requirement must hold. This amounts to checking that the dependability argument is valid; that is, the conjunction of its premises implies the conclusion. Note that it also implies that failure of the other components to fulfill their specifications or assumptions cannot compromise the requirement.

2) Degree of Confidence: An argument is only as strong as its weakest premise. We must show not only that the trusted base is sufficient, but also that we can be confident that these components will fulfill their responsibilities. This step may involve consulting a domain expert to ensure that each domain assumption is a reasonable characterization of the reality. For each machine, we must provide evidence that the machine conforms to its specification (e.g. a proof, a testing report, etc.). If a component in the trusted base is deemed unreliable, we must consider redesigning the system in order to exclude the component from the trusted base.

As an example, we might evaluate our confidence in the trusted base for the file transfer design in Figure 1 as follows:

- **File systems**: *Can we assume that file operations are reliable?* Not necessarily. Even though file systems have been used and tested for many years, occasional hardware failures do occur, and we may not assume that the stored or read data will always be consistent.
- **FTP applications**: *Can we implement the FTP software with high confidence?* Yes. The required functionality is arguably simple enough such that a reliable, well-tested implementation should be manageable.
- **Network**: *Can we rely on the network to be free of data corruption and security bleaches?* Not necessarily. The network is susceptible to a variety of hardware and communication failures, as well as security attacks.

Based on this assessment, we conclude that the file systems and the network undermine the confidence in the trusted base. As one option, we may attempt to make these components more reliable—for example, by adding extra fault-tolerance mechanisms. But this approach is not always feasible; the cost of these mechanisms may be prohibitive, and a given domain may not be amenable to modification.

C. Design for small, reliable trusted bases

If the trusted base for a critical requirement contains components that we cannot rely on with confidence, we must rethink the design of the system. This step may involve restructuring the system to exclude the unreliable components from the trusted base.

An alternative design of the file transfer system follows the *end-to-end principle* by Saltzer, Reed, and Clark [3]. The key idea is to regard the network and the file systems as being inherently susceptible to failures, and embed extra checks at the application level to detect a transfer failure.

The alternative design works as follows. After reading the file from the disk, the sender application computes the hash for the file, using standard algorithms such as SHA-2. Then, it requests transfer of the file blocks along with the hash. As in the previous design, the receiver application retrieves the data packets from the network and writes the blocks onto the file system. To detect a failure, the receiver reads the same file from its file system and computes the hash for the file. The receiver compares the two hash values—the received and computed ones; if the two match, then it may conclude that the file transfer was successful. If they do not match, then the receiver reports a failure.

The consequence of the end-to-end design is that the network and the file systems no longer belong to the trusted base for the requirement. In other words, the two machines that represent the FTP applications should, by themselves, be sufficient to ensure the reliability of the transfer, despite potential failures in the network or the file systems[2].

Some caveats are worth mentioning. First, the end-to-end design promises a weaker requirement than the original file transfer requirement. It no longer states that the transfer is always successful; instead, it says that either the transfer succeeds, or if not, then the receiver application will report an error. The full functional requirement of the system still relies on the properties of the network and the file systems. For example, if the network repeatedly drops packets over multiple retries, then the transfer is unlikely to ever succeed. In other words, the end-to-end design guarantees only that "bad things do not happen" (i.e. a failure does not go undetected), not that "good things happen" (i.e. transfer is always successful).

The system is relying on mathematical properties of a good hashing function; that is, it should be computationally infeasible to modify a piece of data without altering its hash or to find two different pieces of data with the same hash. These properties ensure that an erroneous or malicious part of the network or the file system cannot corrupt the data in a way that would result in a hash identical to the original hash. Also, the network should not be able to contrive the hash such that it matches the hash of another piece of data.

Lastly, during the process of redesign, other parts of the system—the extra logic for computing and checking hashes—have become more complex, and will incur a cost not only in construction but also in verification. This is part of the cost of achieving dependability, but in general the design aim is to keep the trusted base small enough that the overall cost is still much lower than it would be if a system-wide verification were called for.

III. Modeling Idiom

Reasoning about trusted bases and dependability arguments can be done informally. However, formal modeling

[2]As we will see in Section IV, analysis of the dependability argument reveals that the requirement actually does depend on the sender file system.

can help the analyst to state assumptions and specifications in an unambiguous manner, and enable rigorous analysis, which can reveal subtle flaws in the reasoning. In this section, we introduce our modeling idiom, which explicitly represents the relationships between requirements and trusted bases. In Section IV, we describe analysis for checking the validity of a dependability argument.

A. Alloy

We use Alloy [13], a modeling language based on first-order relational logic, as the underlying formalism for our idiom. Alloy is suitable for this purpose because: (1) its declarative style is natural for describing relationships between domains and machines; (2) its flexibility, and lack of built-in idioms, allows an unconventional structuring; and (3) its analysis engine, the Alloy Analyzer, provides automated consistency checking. But our approach does not prescribe the use of a particular formalism, and other formalisms may well be suitable.

B. Basic Framework

A system consists of a set of domains $\mathcal{D}$, a set of machines $\mathcal{M}$, and a set of requirements $\mathcal{R}$. Each one of domains, machines, and requirements is associated with a *property*. For a domain, this property is the conjunction of the assumptions; for a machine, it comprises the specification; and for a requirement, it expresses the customer's desire. We define a function $\mathcal{TB} : \mathcal{R} \rightarrow \mathbb{P}(\mathcal{D} \cup \mathcal{M})$, which maps each requirement to the set of domains and machines that constitute its trusted base.

These basic elements can be modeled in Alloy as follows:

```
abstract sig Property {}
abstract sig Domain extends Property {}
abstract sig Machine extends Property {}
abstract sig Requirement extends Property {
  tb : set (Domain + Machine)
}
```

The Alloy keyword *signature* declares a set of elements, and *extend* defines a subtyping relation. An *abstract* signature cannot be instantiated, and must be extended by another non-abstract signature; therefore, every element of type *Property* is an element in *Domain*, *Machine*, or *Requirement*. Enclosed in the declaration of *Requirement*, *tb* is a *signature field* that represents the trusted base function, so that $\mathcal{TB}(r)$ will be denoted in Alloy as $r.tb$, where r is a particular instance of *Requirement*.

We introduce an additional variable $\mathcal{OK} \subseteq (\mathcal{D} \cup \mathcal{M} \cup \mathcal{R})$ representing the set of all domains, machines, and requirements that satisfy their properties. In Alloy, we declare this set to be a subset of *Property* using the keyword *in*:

```
abstract sig OK in Property {}
```

An element of type *Property* belongs to *OK* if and only if the property of the element is satisfied. For example, a machine *M* with a specification (represented by a logical formula *S*) is a member of *OK* if and only if *S* holds true:

```
sig M extends Machine {
  ...
}{
  // signature constraint
  this in OK iff S
}
```

A *signature constraint* appears as an appendix of a signature declaration, and is a constraint that holds for every element in that signature; the keyword *this* represents the archetypal element of the signature. So the preceding fragment of Alloy says that the machine *M* is in *OK* if and only if it satisfies its specification. A similar style of specification can be used to classify domains into those that behave as expected, and ones that misbehave; and requirements into those that are satisfied by the system, and ones that are violated.

Having defined the set $\mathcal{OK}$, we formulate a traditional argument for the satisfaction of requirements as follows:

$$(\mathcal{D} \cup \mathcal{M}) \subseteq \mathcal{OK} \Rightarrow \mathcal{R} \subseteq \mathcal{OK} \tag{1}$$

This argument says that if all domains behave as expected and all machines meet their specifications, the satisfaction of desired requirements must follow.

In comparison, our proposed notion of a dependability argument takes the following form:

$$\forall r : \mathcal{R} \cdot \mathcal{TB}(r) \subseteq \mathcal{OK} \Rightarrow r \in \mathcal{OK} \tag{2}$$

That is, for each requirement r, if all of the assumptions and specifications of the components in its trusted base hold, then the requirement is satisfied, *regardless* of how the other components behave. In Alloy, we use an *assertion* to state the dependability argument:

```
assert DependabilityArgument {
  all r : Requirement | r.tb in OK implies r in OK
}
```

The distinction between (1) and (2) has significant implications on the development process. It suggests that key design decisions should be driven with the goal of minimizing the trusted bases for the most critical requirements. During implementation, it may mean assigning the most competent programmers to build the trusted machines, and using a safe language and simple algorithms that are easy to understand and implement. Finally, the distinction allows one to allocate available testing and verification resources more effectively, by focusing them on components in the trusted bases.

C. Model of the End-to-End File Transfer System

We demonstrate the usage of our idiom with a model of the end-to-end file transfer system. We begin by modeling *phenomena*, the elementary building blocks of the world. The purpose of the system is to transfer data packets between two computers. There are two types of packets: blocks and file hashes. Each file consists of a set of blocks, and is associated with a hash; we represent these two as fields of the signature *File*:

```
abstract sig Packet {}
sig Block extends Packet {}
sig Hash extends Packet {}
sig File {
  blocks : set Block,
  hash : Hash
}
```

The file system on the sender contains a file to be sent over the network. It interacts with the sender FTP application; we express this relationship by declaring an element of type *SenderApp* as a field of the signature *SenderFileSys*. The signature constraint encodes a domain assumption that the content of the file received by the application matches the content in the file system; the assumption holds if and only if the file system is OK (i.e. it successfully performs the read operation):

```
sig SenderFileSys extends Machine {
  file: File,
  app: SenderApp
}{
  this in OK iff
    // read operation works correctly
    app.readFile = file
}
```

The specification of the sender application has two aspects: computing the hash on the file that it reads off the file system; and requesting transfer of the blocks to the network:[3]:

```
sig SenderApp extends Machine {
  network: Network,
  readFile : File,
  hash: Hash
}{
  this in OK iff
    // app computes the hash of the file to be sent
    hash = readFile.hash and
    // app requests transfer of both the file blocks & hash
    network.packetsIn = readFile.blocks + hash
}
```

The network domain is associated with two sets of packets: the ones coming in from the sender computer, and the ones going out to the receiver. A desirable (but perhaps too optimistic) assumption about the network is that data transfer is completely free of errors—i.e. every packet going into the network arrives at the destination exactly as it is. This assumption holds true if and only if the network is reliable:

```
sig Network extends Domain {
  packetsIn, packetsOut : set Packet
}{
  this in OK iff packetsIn = packetsOut
}
```

In all these descriptions, there is no commitment to the stated property holding: instead, the model requires that the property hold if and only if the component is OK. The receiver application must take the data packets off the outgoing port of the network and store them as a hash (*receivedHash*) and a file (*receivedFile*), which is then written to the receiver file system. The application reads back the same file into its memory and computes a separate hash value (*computedHash*). The two hashes are then compared against each to check whether or not the transfer was successful:

```
sig ReceiverApp extends Machine {
  network: Network,
  receivedFile, readFile : File,
  receivedHash: lone Hash,
  computedHash : lone Hash
}{
  this in OK iff
    // app reads packets off the network & stores them
    receivedFile.blocks + receivedHash = network.packetsOut and
    // app computes a hash based on the file read back
    computedHash = readFile.hash
}

sig ReceiverFileSys extends Machine {
  file: File,
  app: ReceiverApp
  }{
  this in OK iff
    // the file received by app is written correctly
    file = app.receivedFile and
    // the same file is read back by the app
    app.readFile = file
  }
```

Note that this model does not describe dynamic behavior, such as the particular sequence of file read and write operations and the state changes they produce. Instead, assumptions and specifications are written as static constraints on the phenomena between the interacting components. This style allows a simpler and more natural description, corresponding, for example, to the intuition that in a reliable network, incoming and outgoing packets match. However, our approach does not preclude the analyst from modeling dynamic behavior explicitly, should that seem more useful.

In order to describe hashing, we declare as an Alloy *fact* some properties that are assumed always to hold[4]:

```
fact HashingProperties {
  all f1, f2 : File | f1.hash = f2.hash iff f1.blocks = f2.blocks
  // network cannot contrive a hash
  all n : Network |
    all h : Hash & n.packetsOut |
      h in n.packetsIn or (no f : File | f.hash = h)
}
```

The first formula in the fact states that files have the same hash if and only if their contents are equivalent. The second formula says that every hash is transferred without corruption, or does not match the hash of an existing file; in other words, errors in the network should not be able to contrive the hash in such a way that it matches the hash of another file. We point out two caveats. First, we distinguish these properties from a domain assumption, since they are mathematical properties that are universally regarded to be true, regardless of the problem context. Second, a

[3]In Alloy, + is the operator for set union.

[4]In Alloy, & is the operator for set intersection.

hash function provides only a probabilistic guarantee of the properties, but in practice stronger assumptions such as this are always made. Vulnerabilities in cryptographic primitives are unlikely to be the weakest link in the dependability chain and are therefore not the focus of a system-level analysis.

The last piece of the model is the reliable file transfer requirement. The two file systems and the receiver application contain phenomena that are referenced by the requirement, and appear as the fields of the signature *ReliableTransferReq*[5]. We state that the requirement is satisfied only when the transfer is successful, and if not, then the receiver application is able to detect the failure by comparing the hash values:

```
sig ReliableTransferReq extends Requirement {
  senderFileSys: SenderFileSys,
  receiverFileSys: ReceiverFileSys,
  receiverApp : ReceiverApp
}{
  this in OK iff
    // the transfer is successful or the hashes do not match
    (senderFileSys.file.blocks = receiverFileSys.file.blocks or
     receiverApp.receivedHash != receiverApp.computedHash)
  // receiverApp is the correct instance
  receiverApp = receiverFileSys.app
  // the trusted base includes the two applications
  tb = {senderFileSys.app + receiverFileSys.app}
}
```

The second signature constraint ensures that *receiverApp* is the instance of type *ReceiverApp* that interacts with the receiver file system. Finally, the last constraint indicates that the trusted base for the requirement consists of the sender and receiver applications.

The flexibility of the *signature* construct in Alloy allows reification of meta-level concepts, which we heavily exploit in our modeling idiom. These include the set *Property* and its subset *OK*, which are used to describe and constrain behaviors of domains, machines, and requirements. Technically, the set *OK* is not necessary for analyzing a dependability argument; we could conjoin the assumptions and specifications of the components in the trusted base, and check whether the resulting formula implies the desired requirement. However, *OK* allows us to encode the effect on the system when a domain or a machine fails to satisfy its property. For example, we can generate and visualize a scenario in which a component failure leads to the violation of a requirement, but still preserves the satisfaction of another, more critical requirement. We illustrate such analyses in Section IV.

IV. Analysis

The Alloy Analyzer [13] is a model finder: it solves formulas to find satisfying assignments. The standard applications are animation (generating sample scenarios) and checking (generating counterexamples to a claim). In this section, we describe how checking can be used to analyze a dependability argument.

Recall the assertion for the generic dependability argument from the end of Section III-B:

```
assert DependabilityArgument {
  all r : Requirement | r.tb in OK implies r in OK
}
```

The assertion states that as long as the components in the trusted base for the requirement are *OK*, then the requirement holds. A refutation to this assertion would be a scenario in which the components in a trusted base behave as expected, but the requirement fails to hold.

We use a *check* command in Alloy to check an assertion:

```
check DependabilityArgument for 5
```

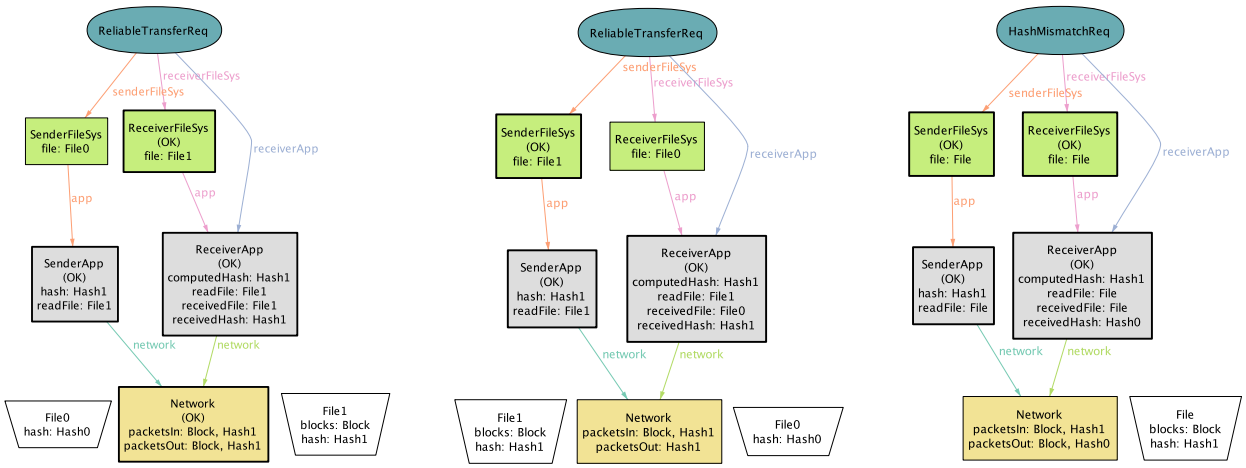
The analysis in Alloy is automatic, and exhaustive up to the specified bound. In this example, it is guaranteed to find a counterexample if one exists involving at most five objects of each signature. The absence of a counterexample does not necessarily imply that the claim is valid, but by increasing the scope, the analyst can gain further confidence. Counterexamples returned by the analyzer are usually small, making it easier to trace them back to flaws in the model.

When executed with the above check command, the Alloy Analyzer finds a counterexample, and generates a diagram using its built-in visualizer, as shown in Figure 2(a). This particular counterexample illustrates a scenario in which the sender application reads an incorrect file content (*File1* instead of *File0*) from the unreliable file system. As a result, the file transferred onto the receiver does not match the content of the original file, even though all other components behave correctly. From this observation, we conclude that the requirement depends on the correctness of the read operation in the sender file system, and revise its trusted base to include the additional component:

```
sig ReliableTransferReq extends Requirement {
  ...
}{
  ...
  // the trusted base includes the two applications as well as
  // the sender file system
  tb = {senderFileSys.app + receiverFileSys.app +
        senderFileSys}
}
```

When we re-run the *check* command, the Alloy Analyzer returns another counterexample, as shown in Figure 2(b). This counterexample represents a scenario in which the the unreliable network fails to transfer the only block (*Block*) in *File1*. As a result, the content of the file (*File0*) written onto the receiver file system is incorrect. However, when the receiver application reads back the same file, the erroneous read operation happens to return a file that has the same content as the original file on the sender—highly unlikely, but still not impossible! The hash value computed on this

[5]One might note that the requirement references the phenomena in *ReceiverApp*. To simplify our example, we departed slightly from the standard problem frames approach, in which a requirement always references domain (and not machine) phenomena. We could instead add a domain *TransferReport* that displays whether a transfer was successful; the requirement would then reference phenomena in this domain.



(a) Erroneous file read (b) Data transfer failure (c) Hash corruption

Figure 2. Counterexamples to the assertions *DependabilityArgument*. The components that satisfy their properties are labeled with *(OK)* in the visualizer.

file matches the received hash (*Hash1*), and the receiver application fails to detect the network transfer error, violating the reliability requirement.

One way to fix this problem would be to include the receiver file system as a part of the trusted base. But placing trust into a component must be done carefully, as such a design-level decision can have impact on the subsequent implementation and testing activities. We must consider the likelihood of the failure that the counterexample illustrates, and make a judgement on whether or not the component needs to be trusted. In this case, we assume that the likelihood of the file system contriving the content of a file such that it yields a particular hash value is extremely small, given a well-designed hash function. Then, we modify the fact *HashingProperties* to include a constraint that rules out the same class of failures:

```
fact HashingProperties {
  all f1, f2 : File | f1.hash = f2.hash iff f1.blocks = f2.blocks
  // network cannot contrive a hash
  all n : Network |
    all h : Hash & n.packetsOut |
      h in n.packetsIn or (no f : File | f.hash = h)
  // file system cannot contrive a file content for a particular hash
  all fsys : ReceiverFileSys | let app = fsys.app |
    fsys.file != app.readFile implies
      app.readFile.hash != app.receivedHash
}
```

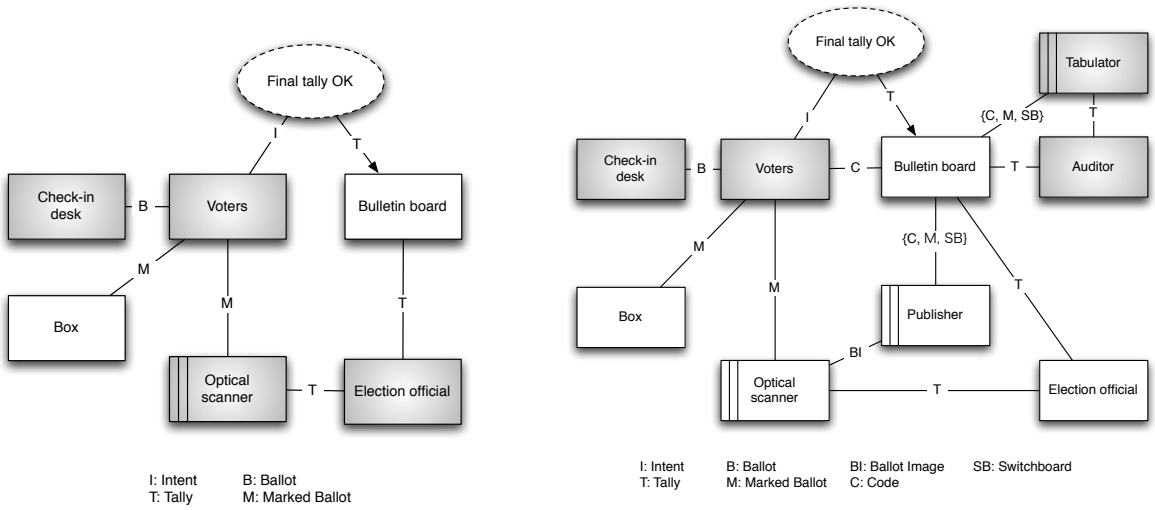
When we re-run the *check* command with the modified fact, the analyzer no longer returns a counterexample. Facts are like axioms in a proof; their validity must be discharged with a careful examination by a domain expert, such as a designer of a cryptographic hash function. One of the benefits of formal analysis is that it forces their documentation, so that they can be subject to an examination.

Let us consider another requirement for the file transfer system, that "a mismatch between the received and computed hash values on the receiver application indicates a file transmission failure". This requirement is desirable, since if it does not hold, the receiver application may mistakenly believe that the file transfer has failed, and request of the sender a gratuitous retransmission. We initially posit that the trusted base for *ReliableTransferReq* is also sufficient to ensure *HashMismatchReq*, and indicate as such using a signature constraint:

```
sig HashMismatchReq extends Requirement {
  senderFileSys : SenderFileSys,
  receiverFileSys : ReceiverFileSys,
  receiverApp : ReceiverApp
}{
  this in OK iff
    // hash mismatch means a transmission error has occurred
    (receiverApp.receivedHash != receiverApp.computedHash
     implies
     senderFileSys.file.blocks != receiverFileSys.file.blocks)
  receiverApp = receiverFileSys.app
  tb = {senderFileSys.app + receiverFileSys.app +
        senderFileSys}
}
```

When we re-run the *check* command with the new requirement, the Alloy Analyzer generates a counterexample, as shown in Figure 2(c). The diagram shows a scenario in which the two hashes on the receiver application do not match, but *File* has been transferred successfully. A close look reveals that the original hash (*Hash1*) has been corrupted during the transfer. Further analysis reveals that the trusted base needs to include both the network and the receiver file system to establish this requirement:

```
sig HashMismatchReq extends Requirement {
  ...
}{
  ...
  tb = {senderFileSys.app + receiverFileSys.app +
        senderFileSys + receiverFileSys +
        senderFileSys.app.network}
}
```



(a) Optical scanner system

(b) Scantegrity

Figure 3. Problem diagrams for the two electronic voting systems. The highlighted components constitute the trusted base for the requirement.

When the analyzer checks the assertion again, it no longer returns a counterexample. This example shows that different requirements have different trusted bases. The requirement *HashMismatchReq* has a larger trusted base, extending to the entire system. Since this requirement is less critical, however, one would expect to devote fewer resources to this trusted base, so identification of the smaller trusted base for the more critical requirement still results in a cost reduction.

V. Case Study: Electronic Voting Systems

As a case study, we used our approach to construct and analyze dependability arguments for two types of electronic voting system: an optical scan system, which is used widely in the U.S., and the Scantegrity system [9], one of several proposals that employ cryptographic methods to improve dependability. The goal of the case study was to demonstrate how trusted bases can be used to compare two different system designs, and argue why one is more dependable than the other in achieving a critical requirement. Due to limited space, we provide only an informal discussion of the systems; the description of complete Alloy models is available in the thesis on which this paper is based [14].

We began by building a model of an optical scan system, based on the description of an existing system in [15]. Figure 3(a) shows the problem diagram for the optical scan system, with the components in the trusted base highlighted. The requirement is to ensure that the final tally of the election match the choice of the voters for a particular candidate[6]. Following our approach, we identified the trusted base that consists of the check-in desk, voters, the optical scanner, and the election official. We constructed the following dependability argument:

> If each voter receives exactly one ballot at the check-in desk, the voter casts the ballot as intended, the scanner correctly records the choice on the ballot and computes the final tally, and the election official correctly reports the scanner's tally on the bulletin board, then the outcome of the election will accurately reflect the choice of the voters.

The argument is valid, but a major weakness lies in one of the trusted components—in this case, the optical scanner. Researchers have shown that existing optical scanners are vulnerable to a range of security attacks [15], [16]. Further compounding the problem, the software in most scanners is proprietary, and not subject to public inspection (which could help uncover such flaws).

Scantegrity [9] is a recently proposed design that is not vulnerable to scanner failure; it maintains election integrity without reliance on the scanner or election official. Figure 3(b) shows the problem diagram for Scantegrity. The system is deployed as an add-on to an existing optical scanner system, and includes three additional components—the publisher, the tabulator, and the auditor.

With Scantegrity, the voter retains a *ballot receipt* after casting a vote. The receipt contains a code that the voter can enter into a website, after the election, to verify that the vote was counted in the final tally. This step provides a way for the voters to detect a failure in which the scanner ignores or incorrectly records their ballots.

To support this verification, Scantegrity constructs a *switchboard*, a table that maps a code on each ballot to a corresponding candidate. All of the codes, as well as

[6]Every voting system must also ensure *privacy* of the voters; that is, it should not be possible to trace a marked ballot to a particular voter. For our purpose, we omit the discussion of the privacy requirement.

the switchboards, are published onto the bulletin board, following the election. Any third-party auditor can construct tabulating software that examines the switchboard and computes a tally that is independent from the scanner's result. By checking whether the two tallies match or not, the auditor can detect a failure in the scanner's computation of the tally.

The trusted base for the tally requirement in Scantegrity includes: (1) the check-in desk, which must ensure that each voter receive exactly one ballot, (2) the voters, who must verify the inclusion of their ballots on the bulletin board, in addition to casting the ballots as intended, (3) the tabulator, which must correctly compute an independent tally using the switchboard, and (4) the auditor, who must compare the two independent tallies to ensure that they match. The dependability argument says that the election outcome is correct as long as these components satisfy their properties; a failure in the scanner cannot give a false election outcome. We checked the validity of the argument in Alloy, and consulted a designer of Scantegrity to confirm that the trusted base that we identified matched their intuition.

A quick glance at Figure 3 might give an impression that Scantegrity is no more dependable than the optical scanner system, because the two trusted bases have the same number of components. But the crucial difference is the *degree of confidence* in the trusted bases. We concluded that the optical scanner system is not reliable, due to the difficulty of achieving high confidence in the scanner. In contrast, the trusted base in Scantegrity is supported by informal but reasoned justifications that provide greater confidence. For example, we feel confident in the outcome of an election audit, because in principle, anyone (including candidates) can act as an auditor, and the probability of all independent auditors colluding is extremely low. Similarly, the switchboard is publicly available, and any interested person can write their own tabulator to compute an independent tally; this open nature arguably leads to higher confidence in the tabulator than in the scanner.

VI. Related Work

Trusted computing bases (TCBs) [4], [5] and security kernels [6] are well-known concepts for building a system around a small core of software and hardware components that are responsible for overall system security. Safety kernels [7] are an analogous concept for safety-critical systems. Our notion of trusted bases differs from these in two respects. First, a trusted base exists for a particular requirement; typically, different requirements depend on different sets of components, and a system therefore has multiple trusted bases. In comparison, only a single TCB or kernel is responsible for enforcing *all* security or safety policies in the system. For example, in the end-to-end file transfer design, we saw how the reliability requirement has a smaller trusted base than the requirement that a hash mismatch imply a failure. Second, a trusted base need not be encapsulated in the manner of a TCB or a kernel. A critical requirement may be established by the cooperation of components that are distributed throughout the system—like, for example, the sender and receiver applications in the trusted base for the reliability requirement.

Haley, Laney, Moffett, and Nuseibeh propose the notion of *trust assumptions* to describe expected properties of environmental components for the satisfaction of security requirements [17]. Like us, they use problem frames to structure a system into domains and machines, and assign a trust assumption to each domain. One may *reject* a trust assumption if it is deemed too risky or unrealistic, and add other domains or phenomena in order to satisfy a requirement. Although they discuss only security requirements, it seems that their approach can be extended to handle safety-critical systems as well. Our approach considers a trusted base consisting of not only domains, but machines as well, and so the satisfaction of a requirement relies additionally on each trusted machine satisfying its specification.

Goal-oriented approaches [18], [19], [20] treat *goals* as essential part of requirements and design activities, and advocate an explicit documentation of the structure of goal decomposition. From one perspective, requirements in our approach are like goals, and assumptions and specifications are like subgoals that together imply a higher-level goal. A goal-oriented notation might be used to crystallize the structure of our dependability argument. We are already exploring this synergy in ongoing work [14], [21].

Our analysis for checking the sufficiency of a trusted base resembles *obstacle analysis* [22]. An obstacle is a behavior of an environmental entity (e.g. an operator) that leads to violation of a goal. Once found, obstacles are resolved by various measures, such as weakening the goal or substituting the bad entity with another; this step is similar to the activity of redesigning a system for a more reliable trusted base.

Assurance-based development (ABD) [23] proposes the construction of a dependability argument hand-in-hand with the development of the system in a particular style. The argument is expressed in Goal Structuring Notation (GSN) [19]. Top-level goals are decomposed into smaller subgoals, which are subsequently discharged using various *strategies* (e.g. testing, verification, etc.). Lutz and Patterson-Hine [24] propose an approach to building an argument that the system is able to detect and handle safety-related contingencies. Like ABD, the structure of their argument is based on the GSN, but they discharge subgoals using analysis of fault models. The focus of these two approaches is on *how* to go about discharging claims in a dependability argument. In contrast, we emphasize identifying *what* components are responsible for ensuring dependability, and structuring the system so that there are fewer such components.

Feather discusses a modeling approach in which a system is described as being composed of interacting *agents*, and introduces the notion of a *responsibility* as a task that

each agent must perform in order to satisfy an overall system requirement [25]. A group of agents that are responsible for a requirement corresponds to our trusted base for the requirement. However, our approach puts greater emphasis on the notion of "trust"; a component can be assigned a responsibility but still be considered unreliable, and consequently, excluded from a trusted base for a critical requirement.

VII. Conclusion

We have proposed an approach in which a system is designed with small, reliable trusted bases that establish its most critical requirements. Much work remains to be done to systematize the task of designing a system to reduce its trusted base. Our approach advocates redesign of a system with an unreliable trusted base, but there is no systematic way to go about such a redesign. We are currently investigating ways to capture existing and often informal design knowledge in a form that is suitable for reuse across applications.

Acknowledgments

Thank you to Emily Shen for explaining the details of Scantegrity; to Aleksandar Milicevic, Joseph Near, Rishabh Singh, and Kuat Yessenov for helpful comments; to the anonymous reviewers for help with the presentation; and to our sponsors for supporting our research: 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).

References

[1] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?" *IEEE Computer*, vol. 39, no. 5, pp. 44–51, 2006.

[2] N. Leveson, *Safeware: System Safety and Cmputers*. ACM New York, NY, USA, 1995.

[3] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, 1984.

[4] B. W. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, pp. 265–310, 1992.

[5] "Trusted computer system evaluation criteria," 1983, department of Defense, CSC-STD-001-83.

[6] G. Popek and D. Farber, "A model for verification of data security in operating systems," *Communications of the ACM*, vol. 21, no. 9, pp. 737–749, 1978.

[7] J. Rushby, "Kernels for safety," *Safe and Secure Computing Systems*, pp. 210–220, 1989.

[8] B. W. Lampson, "Hints for computer system design," *IEEE Software*, vol. 1, no. 1, pp. 11–28, 1984.

[9] D. Chaum, R. Carback, J. Clark, A. Essex, S. Popoveniuc, R. L. Rivest, P. Y. A. Ryan, E. Shen, and A. T. Sherman, "Scantegrity ii: End-to-end verifiability for optical scan election systems using invisible ink confirmation codes," in *EVT*, 2008.

[10] M. Jackson, *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2000.

[11] R. Seater and D. Jackson, "Requirement progression in problem frames applied to a proton therapy system," in *RE*, 2006, pp. 166–175.

[12] M. Jackson and P. Zave, "Deriving specifications from requirements: An example," in *ICSE*, 1995, pp. 15–24.

[13] D. Jackson, *Software Abstractions: Logic, language, and analysis*. MIT Press, 2006.

[14] E. Kang, "A framework for dependability analysis of software systems with trusted bases," Master's thesis, Massachusetts Institute of Technology, 2010.

[15] A. Kiayias, L. D. Michel, A. Russell, and A. A. Shvartsman, "Security assessment of the diebold optical scan voting terminal," University of Connecticut Voting Technology Research Center, Tech. Rep., October 2006.

[16] H. Hursti, "Critical security issues with Diebold optical scan design," *Black Box Voting Project, July*, vol. 4, 2005.

[17] C. B. Haley, R. C. Laney, J. D. Moffett, and B. Nuseibeh, "The effect of trust assumptions on the elaboration of security requirements," in *RE*, 2004, pp. 102–111.

[18] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," *Sci. Comput. Program.*, vol. 20, no. 1-2, pp. 3–50, 1993.

[19] T. Kelly and R. Weaver, "The Goal Structuring Notation–A Safety Argument Notation," in *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[20] E. S. K. Yu, "Towards modeling and reasoning support for early-phase requirements engineering," in *RE*, 1997, pp. 226–235.

[21] D. Jackson and E. Kang, "Property-Part Diagrams: A Dependence Notation for Software Systems," in *ICSE '09 Workshop: A Tribute to Michael Jackson*, 2009.

[22] A. van Lamsweerde and E. Letier, "Handling obstacles in goal-oriented requirements engineering," *IEEE Trans. Software Eng.*, vol. 26, no. 10, pp. 978–1005, 2000.

[23] P. J. Graydon, J. C. Knight, and E. A. Strunk, "Assurance based development of critical systems," in *DSN*, 2007, pp. 347–357.

[24] R. R. Lutz and A. Patterson-Hine, "Using fault modeling in safety cases," in *ISSRE*, 2008, pp. 271–276.

[25] M. S. Feather, "Language support for the specification and development of composite systems," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 2, pp. 198–234, 1987.