

# Security as a System Property: Modeling trust and security in Rosetta

Perry Alexander and Garrin Kimmell  
The University of Kansas, ITTC  
(`{alex,kimmell}@ittc.ku.edu`)

David Burke  
Galios Connections, Inc.  
(`davidb@galois.com`)

January 4, 2007

## Abstract

In this white paper we assert that security and trust are system-level specification properties. As such, using a system-level specification language like Rosetta should be a solution to specification and verification of security and trust properties. We define a vocabulary for trust in Rosetta, define a domain for trusted systems, and use this infrastructure to define a simple trusted system. We close by briefly outlining verification techniques applicable to Rosetta systems of this type.

**This is a draft technical report and will be undergoing frequent modifications. Please see <http://www.ittc.ku.edu/Projects/SLDG/> for updates.**

## 1 Security and Trust as System Properties

The objective of this white paper is to demonstrate the effectiveness of modeling system security and trust as a system-level property. To achieve this, we will outline the specification of a simple, secure system using the Rosetta [1] system-level specification language. We start by defining Rosetta packages and a domain for specifying security. We will then outline mechanisms for specifying various security and trust relationships. With the Rosetta subsystem in place, we will then define a specification for a simple telecommunication system. We close with a brief discussion of how verification of this specification would proceed.

## 2 Vocabulary of Trust

Our first task is to define a *vocabulary of trust* that defines the units of trust semantics. Throughout this paper when we refer to 'trust' we are referring to 'trust' in the security domain with respect to a collection of security properties. Ultimately, every system component is either deemed trustworthy, or the risk in trusting them is deemed acceptable.

We start by defining trust relationships that may hold between different agents. One agent may trust another, be trusted, or ultimately be demonstrated to be trustworthy. We then define what properties of an agent can be trusted. Simply trusting an agent is too general to be useful. Thus we define quantities of trust for the security domain including integrity, confidentiality, availability, and accountability.

### 2.1 Trust Relationships

Three basic relationships describe trust between two agents,  $a$  and  $b$ : (i)  $a$  trusts  $b$  with respect to  $p$ ; (ii)  $b$  is trusted by  $a$  with respect to  $p$ ; and (iii)  $b$  is trustworthy with respect to  $p$ . We use the term agent in the most general sense – people, software components, hardware components, and processes are all encompassed by the term agent. When  $a$  trusts  $b$  with respect to  $p$ ,  $a$  depends on  $b$  to perform its task in a trustworthy fashion with respect to property  $p$ . When  $b$  is trusted by  $a$  with respect to  $p$ , some  $a$  trusts that  $p$  holds for  $b$ . Finally, when  $b$  is trustworthy with respect to  $p$ , it can be determined formally that  $p$  truly does hold for  $b$ .

$trusts(a, b, p(i))$  – agent  $a$  trusts agent  $b$  with  $i$  with respect to  $p$  when it depends  $b$  to exhibit  $p$  when dealing with  $i$  for trustworthy operation. For example, if one agent trusts the another agent with the confidential handling of a key,  $k$ , the agent is relying on the fact that its information will not be transmitted to another, unauthorized agent.

$$trusts(a, b, confidentiality(k))$$

expresses this relationship.

$trusted(b, a, p(i))$  – agent  $b$  is trusted by agent  $a$  with  $i$  with respect to some property  $p$  for trustworthy operation. For example, if one agent's confidential handling of some key,  $k$ , is trusted by another, the other agent is relying on the fact that its information will not be transmitted to another, unauthorized agent.

$$trusted(b, a, confidentiality(k))$$

expresses this relationship. Note that *trusted* is the inverse of *trusts*:

$$\text{trusts}(a, b, p(k)) \Leftrightarrow \text{trusted}(b, a, p(k))$$

*trustworthy*( $b, p, i$ ) – an agent  $b$  in a system is trustworthy with respect to property  $p$  when it can be verified it is safe to trust that  $p$  holds for it. For example, if the confidentiality of an agent is trustworthy in its handling of some key,  $k$ , with respect to  $p$ , then all of the agents that trust its confidentiality are safe in doing so. This relationship is expressed as:

$$\text{trustworthy}(b, \text{confidentiality}(k))$$

We can infer from this that  $a$ 's trust in  $b$ 's confidential handling of  $k$  is justified.

The general goal of verifying a trusted system is assuring that *all trusted agents are trustworthy*. Trust relationships form a chain that must be rooted on trustworthy components. If a trust chain terminates in a trusted component that is not trustworthy, the security of the components involved in that trust chain cannot be assured.

## 2.2 Qualities of Trust

One agent simply trusting another is too abstract and not descriptive enough to be useful in specification or verification. When an agent trusts another, it virtually always refers to some property that is trusted. It is frequently not necessary for all properties to be trusted. For example, it may be that you trust your cell phone's integrity (it will repeat to you what it receives), but not its availability (you don't get '4 bars' everywhere you go) or confidentiality (others may observe information sent and received). Thus, when you say you trust your cell phone, you are really saying that you trust its integrity.

We define four standard properties that describe what is trusted about an agent:

**Confidentiality** – an agent displays confidentiality when it does not reveal information to other unauthorized agents. If one agent trusts the confidentiality of another, then it is relying on the fact that its information will not be transmitted to another, unauthorized agent.

**Integrity** – an agent displays integrity when it does not alter information it receives and only transmit data it believes is true. If one agent trusts the integrity of another, then it is relying on the fact that the agent only communicates information that it believes to be correct and there is no intent to deceive.

**Availability** – an agent displays availability when the services it provides are available when requested. If one agent trusts the availability of another, then it is relying on the fact that the agent will be there when needed.

**Accountability** – an agent displays accountability when it does not deny what it has done (also called non-repudiation). If one agent trusts the accountability of another, then it is relying on the fact that the agent will not deny actions it has already taken. Accountability can also be used to express privacy when an agent is not help accountable for its actions.

The confidentiality, integrity, availability (CIA) Triad model is widely accepted in the trust and security community. We have added accountability, a common extension to the CIA Triad, to support modeling audit. Although the literature proposes many different trust calculi for describing and evaluating trust, we concentrate on this initial property set. Other properties may be added or defined in lieu of these properties.

## 2.3 Trust Properties

With trust relationships and trusted properties defined, we can now define some example trust relationships between components. The following represent a collection of simple trust relationships that might be defined:

$trusts(a, b, confidentiality(i))$  –  $a$  trusts the confidentiality of  $b$  when handling  $i$ .

$trusted(b, a, confidentiality(i))$  – the confidentiality of  $b$  when handling  $i$  is trusted by  $a$ .

$trustworthy(b, confidentiality(i))$  –  $b$  is trustworthy with respect to confidentiality of  $i$ . Any agent that trusts  $b$ 's confidentiality with respect to  $i$  is safe in doing so.

$trusts(a, b, integrity(i)) \wedge trusts(a, b, confidentiality(i))$  –  $a$  trusts both the confidentiality and integrity of  $b$ 's handling of  $i$

In each instance of  $trusts$  above, one agent trusts that a property holds over another. If  $a$  and  $b$  are elements of a telecommunications system:

$$trusts(a, b, confidentiality(i))$$

might state that the confidentiality of a handset depends on the confidentiality of the underlying network's handling of information. If that network is trustworthy:

```

package sysTypes():: static is
  ChanType :: type is constant;
  HashType :: type is constant;
  KeyType :: type is constant;

  InfoType(T::contentType) :: type is data
    Chan(c::ChanType)::chan? | // Represents a channel in an interface
    Hash(h::HashType)::hash? | // Represents a hash used for measurement
    Key(k::KeyType)::key? | // Represents a cryptographic key
    ID(i::IDType)::id? | // Represents the identifier for a resource or
                        // domain resolved by the name server.
    Info(i :: T)::info?; // Information transmitted

  red[T::type](i :: InfoType(T))::boolean // Restricted information
  is not(black(i));
  black[T::type](i :: InfoType(T))::boolean // Unrestricted information
  is not(red(i));
end package sysTypes;

```

**Figure 1:** Rosetta package representing primitives.

*trustworthy(b, confidentiality(i))*

then we are assured that  $a$ 's trust in  $b$ 's confidential handling of  $i$  is safe. If trustworthiness is not established or derivable, that trust may not be safe making the system potentially insecure.

## 2.4 Rosetta Data Models

The Rosetta types used to represent information in a system along with how information is classified are defined in figure 1.

**InfoType** represents the most common data type involved in this model. Any data flowing from one component to another is of type **InfoType**. The **InfoType** data declaration effectively composes the various types of information into a single type. Specifically, **InfoType** takes keys, hashes, IDs, and information and composes these into a single type for processing.

Elements of **InfoType** include **Hash(h)**, **Key(k)**, **ID(i)** all of which represent cryptographic information transmitted within the system. Specifically, **Hash** and **Key** fields represent hashes and keys while **ID** items represent identifiers for agents. Items constructed using **Chan** represent communication channels.

The **red** and **black** predicates represent classification levels for data. Red data is considered restricted and should never be mixed with black data. Black data is

unclassified and access to it is unrestricted. The predicate definitions for `red` and `black` define them to be opposites – if data is `red` it is not `black` and visa-versa.

Not all of this package is used in the following specifications as the architectures represented are relatively static. When bringing up or modifying a system, being able to transmit cryptographic data, set up channels and provide name services becomes critical. For this system, with its static architecture, only the `InfoType` and `KeyType` data is used.

## 2.5 Rosetta Trust Models

The Rosetta models used to specify trust relationships follow directly from the previous descriptions and are defined in figure 2. The `trustTypes` package formally defines predicates for each of the properties discussed previously. These predicates can then be used to specify system properties by including the `trustTypes` package in a specification.

Because the definitions in `trustTypes` follow directly from the informal descriptions defined previously. We are literally defining a vocabulary for representing trust relationships in Rosetta models. The idea is that the new Rosetta specification support packages will create a domain-specific language comfortable for use by domain specialists.

Understanding the relationships between the trust models defined earlier and the definitions provided in the Rosetta specification files involves taking the trusted data element out of the trust quality predicate:

$$\text{trusts}(a, b, p(i)) == \text{trusts}(a, b, p, i)$$

## 3 The Trusted System Domain

With a vocabulary for trust defined, we can now define a domain for modeling trusted systems. The `trusted_state_based` domain defined in figure 3 is a trivial extension of the `state_based` domain that adds a trust observation for each way a component can be trusted. The `integrity`, `confidentiality`, `availability` and `accountability` variables indicate whether the defined system exhibits integrity, confidentiality, availability, and accountability respectively. They are observations made internally about the system independently of the system's operational environment.

Although the trust observations are defined, the definition of how they are calculated is not. As we shall see, this is appropriate as different models will have their own definitions of these properties. However, the trustworthiness of all models with respect to these properties must be observable.

```

use sysTypes();
package trustTypes(dom::static; T::type) is

  // Define a shorthand for the instantiated info type.
  I :: type is InfoType(T);

  // Encoding of trust relationship representations . Semantics
  // are or will be defined with BAN logic.

  // p trusts q with i with respect to one of the CIAN properties.
  trusts(p,q::dom; prop::<*(a::dom; i::I)::boolean*>; i::I)::boolean is constant;

  // p is trustworthy with respect to one of the C,I,A,N properties and i
  trustworthy(p::dom; prop::<*(a::dom; i::I)::boolean*>; i::I)::boolean is constant;

  // p is trusted by q with i with respect to one of the CIAN properties
  trusted(p,q::dom; prop::<*(a::dom; i::I)::boolean*>; i::I)::boolean is constant
    where trusted(p,q,prop,i) == trusts(q,p,prop,i);

  // Encoding of qualities that are trusted . Semantics are or will be
  // defined with BAN logic.

  // p exhibits confidentiality with respect to i
  confidentiality(p::dom; i::I)::boolean is constant;

  // p exhibits integrity with respect to i
  integrity(p::dom; i::I)::boolean is constant;

  // p exhibits availability with respect to i
  availability(p::dom; i::I)::boolean is constant;

  // p exhibits accountability with respect to i
  accountability(p::dom; i::I)::boolean is constant;

end package trustTypes;

```

**Figure 2:** Rosetta package representing basic qualities of trust and security.

```

domain trusted_state_based(state_type :: type):: state_based(state_type :: type) is
  export integrity , confidentiality , availability , accountability ;
  integrity , confidentiality , availability , accountability :: boolean;
begin
end domain trusted_state_based;

```

**Figure 3:** The trusted\_state\_based domain definition.

The four trust properties are an excellent example of heterogeneity in systems design. We know that all components may exhibit these properties, but are not required to specify their calculation. A component may `integrity` when its components exhibit integrity; because it is assumed to have integrity; or as a result of its definition. The specifics of the observation depend on the component being observed. It may also be true that these properties are not known for a given component. In this case, it is still perfectly acceptable to observe integrity even if the observation is inconclusive.

The `assumedTrustworthy` example component defines a system that is assumed to be trustworthy. The terms `integrity = true` and `confidentiality = true` in the assumptions section assert that this component is assumed to be trustworthy with respect to integrity and confidentiality. Such definitions are used to define the “place to stand” where trust relationships must be rooted. By making trustworthiness an assumption, it is easy to identify `assumedTrustworthy` as a component that must be trusted before the analysis begins.

```

component assumedTrustworthy():trusted_state_based is
  // Assumed trustworthy example
  begin
    assumptions
      integrity = true;
      confidentiality = true;
    end assumptions
    ...
  end component assumedTrustworthy;

```

**Figure 4:** Assuming a component is trustworthy.

The `componentsTrustworthy` example defines a system that is trustworthy if its components are trustworthy. If the conjunction of the trustworthiness properties of each of the included components is true, then the component is trustworthy. Such definitions are useful for defining trusted systems that are composed of other subsystems.

The `componentsTrustworthy` model is useful, but naive. It is well known that an untrusted system can be constructed from trustworthy components. Thus, we must be able to define relationships between components in addition to defining the trust of individual components. The `relationshipsTrustworthy` component does precisely this by defining several relationships between components and adding another trustworthiness condition to the implications.

`relationshipsTrustworthy` defines trust relationships between two pairs of components in the terms `t1c2` and `t2c3`. Specifically, `c1` trusts the integrity of `c2` and `c2` trusts the integrity of `c3`.

The implications section extends the previous definition of trustworthy to include an integrity relationship between `c1` and `c3`. Specifically, if all components

```

component componentsTrustworthy():trusted_state_based is
// Component trust example
begin
assumptions
...
end assumptions
definitions
  c1: comp1();
  c2: comp2();
  c3: comp3();
end definitions;
implications
  integrity = c1. integrity and c2. integrity and c3. integrity ;
end implications;
end component componentsTrustworthy;

```

**Figure 5:** A system that is trustworthy when its components are trustworthy.

```

component [T::type](relationshipsTrustworthy (): trusted_state_based is
begin
assumptions
...
end assumptions
definitions
  c1: comp1();
  c2: comp2();
  c3: comp3();
  tic1c2: forall (i :: InfoType(T) | trusts (c1,c2, integrity , i ));
  tic2c3: forall (i :: InfoType(T) | trusts (c2,c3, integrity , i ));
end definitions;
implications
  integrity = forall (i :: InfoType(T) | trusts (c1,c3, integrity , i))
    and c1. integrity and c2. integrity and c3. integrity ;
end implications;
end component componentsTrustworthy;

```

**Figure 6:** A system that is trustworthy when trust relationships between its components are valid.

display integrity and it can be inferred that `c1` trusts the integrity of `c3`, then this component is trustworthy.

Note that the trust relationships are defined in the definitions section rather than the assumptions section. This means that these statements are axioms and true by definition. If they were moved to the implications section they would have to be proved. If they were moved to the assumptions section, then would have to be assured before the component is used. Thus, the component definition structure supports different levels of assurance for different circumstances.

In all of the relationships thus far we have defined concepts of *static* trust. Once proved, the trust concepts thus far will not change during operation. Another class of trust, *dynamic* trust, changes as the system changes state during operation. Modeling dynamic trust will allow us to consider operational situations where changing components and environment can change the trustworthiness of a system.

```

component dynamicTrustworthy[T::type]()::trusted_state_based is
// Transitive trust example where trust can change from state to state
begin
assumptions
...
end assumptions
definitions
  c1: comp1();
  c2: comp2();
  c3: comp3();
  forall (i :: InfoType(T) | trusts (c1,c2, integrity , i ));
  forall (i :: InfoType(T) | trusts (c2,c3, integrity , i ));
  integrity ' = forall (i :: InfoType(T) | trusts (c1,c3, integrity , i ))
                and c1. integrity and c2. integrity and c3. integrity ;
end definitions ;
implications
end implications ;
end component componentsTrustworthy;

```

**Figure 7:** A system whose trust properties may change dynamically.

The `dynamicTrustworthy` component moves the integrity calculation to the definitions section and adds a quite innocent 'tick' to the `integrity` definition. This notation, `integrity '` refers to the value of `integrity` in the next state. This component has integrity in the next state if `c1` trusts the integrity of `c3` and all components display integrity in the current state. If the integrity of any component or the integrity condition changes in any state, then the integrity of this component changes.

We can change the integrity definition in several ways. First, we can assure that once a system becomes untrustworthy it is always considered untrustworthy using the following definition:

```

integrity' = integrity
  and forall (i :: InfoType(T) | trusts (c1,c3, integrity , i))
    and c1. integrity and c2. integrity and c3. integrity ;

```

Now not only must the original condition hold, but the system must have integrity in the current state as well. If `integrity` ever becomes `false`, there is no way that `integrity'` can every be true again. Alternatively, we can use disjunction:

```

integrity' = integrity
  or forall (i :: InfoType(T) | trusts (c1,c3, integrity , i))
    and c1. integrity and c2. integrity and c3. integrity ;

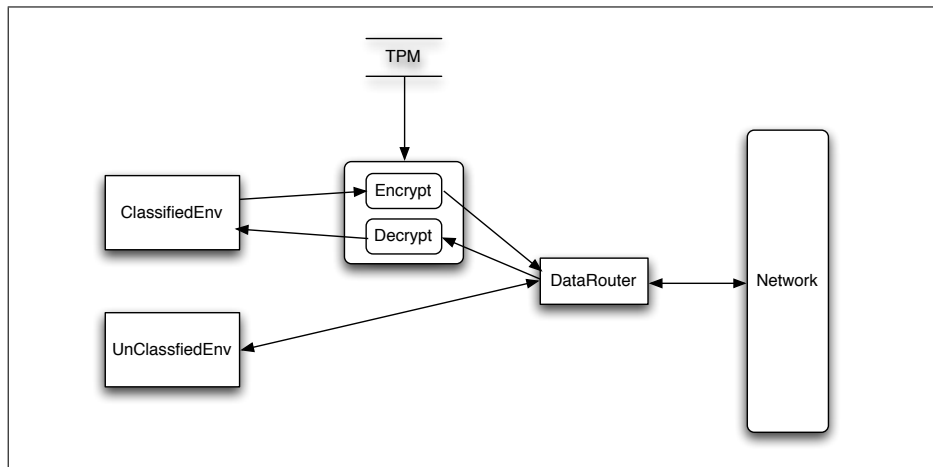
```

This is a dangerous specification because if a system ever displays integrity, it will always display integrity.

The list of trust definitions here is not intended to be exhaustive, but only serves as a collection of examples. The representation is flexible enough to specify a wide variety of integrity definitions.

## 4 Building a Trusted System Specification

With the tools described in the previous section, we can now show the specification of a simple trusted system as described by the data flow diagram in figure 8. The primary task of this system is to allow a classified environment to co-exist in a communications environment with unclassified environments. To certify this system, one must show that red data is never accessible to the unclassified environment and is never transmitted to the network without encryption.



**Figure 8:** High-level architecture of a simple trusted system that routes data to and from an unclassified and a classified environment.

In figure 8 the data router exchanges data with an unclassified network much like an Ethernet card. Data retrieved is assumed to be black because it is either unclassified or encrypted and is routed to both the classified and unclassified environments. The unclassified environment simply processes the data and may route data back to the data router. The cryptographic subsystem attempts to decrypt data it receives from the router, converting it from black to red. Any data it receives from the classified environment is likewise encrypted and thus converted from red to black. By putting the classified system behind a cryptographic system, the hope is that it can use an unsecured data network for communication without sacrificing security.

#### 4.1 The Data Router

The heart of this system is the data router, modeled in figure 9 by the `DataRouter` component. The data router moves data between system components and the network. It makes no attempt to secure the data or transform it in any way, but simply provides transport services.

```

use trustTypes ();
component DataRouter[T::type](
    fromNet,fromCrypto,fromUC:input InfoType(T);
    toNet,toCrypto,toUC:output InfoType(T)
):: trusted_state_based ;
begin
  assumptions
    // No incoming red data
    noRedIn: black(fromCrypto) and black(fromUC);
  end assumptions;
  definitions
    // If information arrives, move it across the router
    netDriver: toNet' = if event(fromCrypto) or event(fromUC) then fromCrypto
                      elseif event(fromUC) then fromUC
                      else toNet
                      end if;
    cryptoDriver: toCrypto' = if event(fromNet) then fromNet
                            else toCrypto end if;
    UCDriver: toUC' = if event(fromNet) then fromNet
                    else toUC end if;
    noRedOut: confidentiality ' = black(toNet') and black(toUC');
  end definitions ;
  implications
    confidential : confidentiality ' == true;
  end implications;
end facet DataRouter;

```

**Figure 9:** Router specification skeleton.

These basic services are described by assertions in the **definitions** section where the value of each input or output signal is defined for the next state. Roughly, the data router monitors each input for events. When an event occurs, data is taken from the input and routed to the appropriate output. If no event occurs, outputs are not changed.

The data router component defines one assumption, one implication and defines confidentiality in terms of data classification. The **noRedIn** term in the **assumptions** section states that all inputs from the cryptography block and the unclassified environment are black. The component is not asserting the inputs are black, but stating that it assumes the inputs are black. When the component is used, these assumptions must be discharged.

The **noRedOut** term in the **definitions** section states that the component displays confidentiality if all outputs from it are black. Unlike the **noRedIn** term, the **noRedOut** term is an assertion of the definition of confidentiality.

Finally, the **confidential** term in the **implications** section defines a correctness condition asserting that **confidentiality** should be true in the next state. If this component is correctly defined, the assumptions and definitions should together prove the implications. The engineering result is that if the component is used in an environment where its assumptions are assured, the verified component will perform as defined.

It should be a simple proof-by-cases to show that by assuming all current inputs from the classified and unclassified are black, all outputs are black in the next state. Output is placed on the **toNet** signal whenever the cryptographic or unclassified subsystem provide data. Looking at the arms of the **if** expression, **toNet** will either get data from **fromUC**, **fromCrypto**, or the previous value of **toNet**.

The first two cases are discharged immediately because data on **fromUC** and **fromCrypto** is assumed black. The third case is more problematic. Nothing is assumed about **toNet** in the initial state. Remember, **toNet'** is the output value in the *next* state. Specification of the initial state has been omitted. This problem can be eliminated by adding the assumption:

**blackOutput**: **black(toNet)**

asserting that the data router is assumed to output black data to the network in the current state. Alternatively, the assumption could be added to the correctness condition:

**confidential** : **black(toNet) => confidentiality** '

again assuming that black data is being output to the net in the current state.

Another question arises with respect to lack of any assumptions about **fromNet**. Nothing is assumed about information arriving from the network implying that it could be red. If that were the case, the data router would send red data to the unclassified environment. However, the data router cannot retransmit

that input data back to the network – network inputs are sent only to the cryptographic and unclassified environments. Should either environment send the red data back to the data router, the data router’s assumptions are violated.

Should the specification be updated to require that no red data appear in the unclassified environment? Should this new requirement be reflected as an assumption on the input to the unclassified environment? Should it be a provable requirement? From a system-level design perspective, discovering these questions is more important than any correctness proof.

## 4.2 The Cryptographic Subsystem

The cryptography subsystem that provides encryption and decryption services for the classified environment is shown in figure 10. This subsystem consists of two components for encryption and decryption respectively that are assembled into a single subsystem. The entire subsystem is defined as displaying confidentiality and integrity if its components display confidentiality and integrity.

The **Encrypt** component accepts an input, encrypts it and marks it as black. By encrypting data, the **Encrypt** component turns red data black. Conversely, the **Decrypt** accepts an input, decrypts it and marks it as red. By decrypting data, the **Decrypt** component turns black data red. Both operations require key information from the TPM described later.

The **Crypto** component assembles the **Encrypt** and **Decrypt** components into a single subsystem. It accepts inputs for encryption and decryption and produces appropriate output for both. The reason for this assembly is simply one of abstraction.

Both **Decrypt** and **Encrypt** are assumed to display confidentiality and integrity. What this says is that we are assuming that both exhibit confidentiality and integrity in isolation. As we shall see, their trustworthiness within a system will depend on their relationships with the components they interact with. This reflects the system-level nature of trust and security requirements.

## 4.3 Components Held Abstract

The remaining specification components are held abstract with little detail of their actual function specified. The **Network** component (figure 12), **ClassifiedEnv**, and **UnClassifiedEnv** (figure 11) represent the data network and data processing environments as data sources and sinks. The **TPM** component (figure 13) represents a source of keys for encryption and decryption.

Each of these components is assumed to be trustworthy with respect to confidentiality and integrity as nothing is known of their actual implementation. The one exception is the **Network** component. As the goal of this system is to provide trusted communications over an untrusted network, making a trustworthiness

```

component Crypto(
    fromRouter,fromC::input InfoType;
    toRouter,toC::output InfoType;
    fromTPM::input KeyType
):: trusted_state_based ;

begin
    definitions
        encrypter : Encrypt(fromRouter,toC,fromTPM);
        decrypter : Decrypt(fromC,toRouter,fromTPM);
        integrity = encrypter. integrity and decrypter. integrity ;
        confidentiality = encrypter. confidentiality and decrypter. confidentiality ;
    end definitions ;
end component Crypto;

component Decrypt(
    fromRouter,fromC::input InfoType;
    toRouter,toC::output InfoType;
    fromTPM::input KeyType
):: trusted_state_based ;

begin
    assumptions
        integrity =true;
        confidentiality =true;
    end assumptions;
    definitions
        classify : red(toC');
        driver : toC' = decrypt(fromTPM,fromRouter);
    end definitions ;
end component Decrypt;

component Encrypt(
    fromC::input InfoType;
    toRouter::output InfoType;
    fromTPM::input KeyType
):: trusted_state_based ;

begin
    assumptions
        integrity =true;
        confidentiality =true;
    end assumptions;
    definitions
        unclassify : black(toRouter');
        driver : toRouter' = encrypt(fromTPM,fromC);
    end definitions ;
end component Encrypt;

```

Figure 10: Cryptographic support subsystem specification skeleton.

```

component ClassifiedEnv(
    fromCrypto::input InfoType;
    toCrypto::output InfoType;
):: trusted_state_based ;

begin
    assumptions
        confidentiality = true;
        integrity = true;
    end assumptions;
    definitions
        hot: red(toCrypto);
    end definitions ;
end component ClassifiedEnv;

use trustTypes ();
component UnClassifiedEnv[T::type](
    fromRouter::input InfoType(T);
    toRouter::output InfoType(T);
):: trusted_state_based ;

begin
    assumptions
        confidentiality = true;
        integrity = true;
    end assumptions;
    definitions
        notHot: black(toRouter);
    end definitions ;
end component UnClassifiedEnv;

```

**Figure 11:** Classified and Unclassified Environment specification skeletons.

assumption – requiring a trusted network – would defeat the purpose of the entire design.

```

use trustTypes ();
component Network[T::type](
    i::input InfoType(T);
    o::output InfoType(T)
):: trusted_state_based ;

begin
    assumptions
    end assumptions;
    definitions
    end definitions ;
end facet Network;

```

**Figure 12:** Network specification skeleton.

## 5 Trust as a System Property

The functional architecture for this system is shown in figure 14 with each component instantiated and interconnected in a Rosetta component. This specification should not be surprising to anyone working with VHDL, Rosetta or any other system supporting structural specification.

The only correctness conditions specified are that each of the components must be individually trustworthy with respect to confidentiality and integrity except the network. Each component has a trustworthiness definition guaranteed by

```

use trustTypes ();
component TPM[T::type](
    fromTPM::input InfoType(T)
):: trusted_state_based ;
begin
    assumptions
        confidentiality = true;
        integrity = true;
    end assumptions;
    assumptions
        trustTPM1: confidentiality =true;
        trustTPM2: integrity=true;
    end assumptions;
end component TPM;

```

Figure 13: TPM specification skeleton.

```

use trustTypes ();
component TrustedSystemArch[T::type]()::trusted_state_based;
    netToRouter, cryptoToRouter, UCtoRouter,
    routerToNet, routerToCrypto, routerToUC,
    cryptoToC, CtoCrypto::InfoType(T);
    TPMout::KeyType;
begin
    assumptions
    end assumptions;
    definitions
        CE: ClassifiedEnv (cryptoToC, CtoCrypto);
        UCE: UnClassifiedEnv(routerToUC, UCtoRouter);
        CRYPT: Crypto(routerToCrypto, CtoCrypto, cryptoToRouter, cryptoToC, TPMout);
        TPM: TPMod(TPMout);
        R: DataRouter(netToRouter, cryptoToRouter, UCtoRouter,
            routerToNet, routerToCrypto, routerToUC);
        N: Network(netToRouter);
    end definitions ;
    implications
        allIntegrity : CE.integrity and UCE.integrity and CRYPT.integrity
            and TPM.integrity and R.integrity ;
        allConfidentiality : CE.confidentiality and UCE.confidentiality
            and CRYPT.confidentiality and TPM.confidentiality and R.confidentiality ;
    end implications;
end component TrustedSystemArch;

```

Figure 14: Architectural specification of the trusted system.

the domain used. Satisfying the correctness condition reduces to simply assuring that each component's definition of `confidentiality` and `integrity` are true.

There are no trust or security relationships specified *between* components as a part of the functional architecture in figure 14. Only references to a component's isolated trustworthiness is referenced. From a system-level design perspective, this is exactly how it should be. A separate specification, `TrustedSystemTrust`, is shown in figure 15 that describes trust relationships between components separating this concern from that of specifying structure or connectivity.

```

use trustTypes ();
component TrustedSystemTrust[T::type]()::trusted_state_based ;
begin
  assumptions
    forall ( i :: InfoType(T) |
      (key?(i) implies trusts (CRYPT,TPM,confidentiality,i))
      and trusts (CRYPT,TPM,integrity,i)
      and trusts (R,N, integrity , i)
      and trusts (CRYPT,R,integrity,i)
      and trusts (CE,CRYPT,confidentiality,i) and trusts (CE,CRYPT,integrity,i)
      and trusts (UCE,R, integrity , i ));
  end assumptions;
  definitions
    CE: ClassifiedEnv ();
    UCE: UnClassifiedEnv();
    CRYPT: Crypto();
    TPM: TPMod();
    R: DataRouter();
    N: Network();
  end definitions ;
  implications
    CEOK: forall (T::type; i :: InfoType(T) |
      trustworthy(CE, integrity , i) and trustworthy(CE, confidentiality , i ));
    SystemOK: integrity and confidentiality ;
  end implications;
end component TrustedSystemTrust;

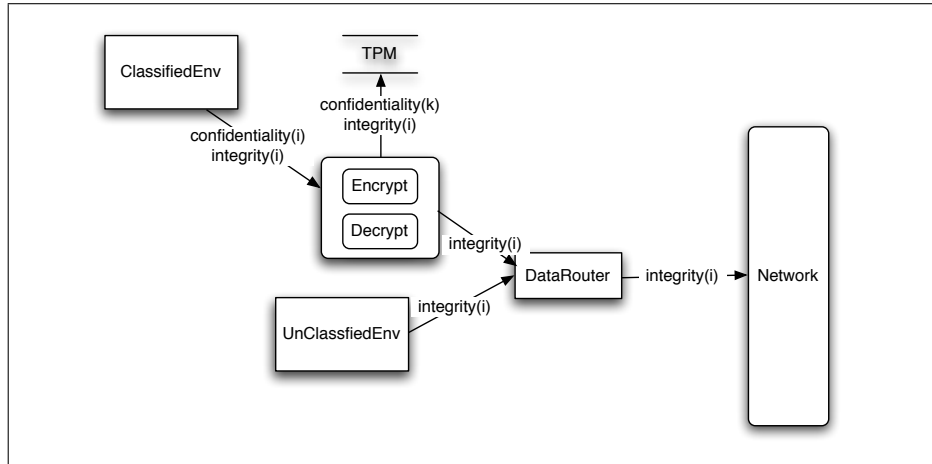
```

**Figure 15:** Trust relationships between components in the trusted system.

Unlike the `TrustedSystemArch` specification, the `TrustedSystemTrust` specification does not concern itself at all with interconnections between components. It simply asserts that each component exists and defines trust relationships between the components.

The assumptions define a collection of trust relationships between components in the system. For example, the `CRYPT` component trusts the confidentiality and integrity of keys from the `TPM` module while the data router (`R`) trusts the integrity of all information from the network, but says nothing of confidentiality. The classified environment (`CE`) must trust the integrity and confidentiality of

how the cryptographic system handles all data. Together, these assumptions define a network of trust relationships that are assumed to hold between system components and are shown as a trust relationship diagram in figure 16.



**Figure 16:** Trust relationships specified by `TrustedSystemTrust` in graphical form.

The implications section defines what we are ultimately after – a correctness condition that will tell us if the system is trustworthy and secure. The implications section includes two terms. The `CEOK` term asserts that the classified environment must be trustworthy with respect to integrity and confidentiality. Simply stated, we must prove or otherwise verify that the classified environment will not retransmit classified data to unauthorized third parties and that information it provides is believed to be true.

The `SystemOK` term may seem a bit odd as it simply asserts the `integrity` and `confidentiality` observations for the system. In addition, there is no definition for either in the entire `TrustedSystemTrust` specification. How can this be verified? The answer lies in the essence of system-level design – understanding the global impacts of local design decisions. We must now compose `TrustedSystemArch` with `TrustedSystemTrust` to understand how they interact.

```
TrustedSystem(): trusted_state_based is
    TrustedSystemArch() * TrustedSystemTrust();
```

**Figure 17:** The final trusted system specification composing the architectural specification with the trust relationship specification.

The final specification, `TrustedSystem`, is shown in figure 17 and may seem at bit anti-climactic. The `TrustedSystem` component is simply the product of the `TrustedSystemArch` and `TrustedSystemTrust` components. It asserts that both models *simultaneously* describe the trusted system.

The ramifications of `TrustedSystem` are that changes in the architecture can impact trust. Recall that we had no mechanism for determining a value for `integrity` and `confidentiality` used in the implications of `TrustedSystemTrust`. Now we do – the value of `integrity` and `confidentiality` in both specifications must be consistent due to the product operation. Therefore, information transfers between the two models and changes in one model may impact others. *Rosetta's contribution is this ability to detect inconsistencies and changes in correctness.*

This is an example of Rosetta's facet algebra and interaction system at work. In actuality, this simple example uses these Rosetta subsystems in rather trivial ways. However, in a more realistic specification situation one can imagine the utility of moving information between specifications when determining trust.

## 6 Verifying The Trusted System Specification

With many systems, the act of writing a specification is much more valuable than the verification process. Dealing with software that that requires certification or operates with high-assurance constraints, verification becomes more critical.

The Raskell verification environment is a collection of tools specifically designed to address analysis of Rosetta specifications. It is much less a tool than a confederation of tools that make writing custom analysis capabilities much easier. Because Rosetta supports multiple semantics, writing a single Rosetta simulator or analysis engine is intractable. Instead, Raskell enables analysis by supporting *abstract interpretation* using *functors* to extract simpler models that are semantically sound.

### 6.1 Abstract Interpretation

The semantics of abstract interpretation is well beyond the scope of this paper. It's intent is to extract simple, sound models from complex models in order to perform analysis. Rosetta functors are functions that move a specification from one domain into another. Thus, a functor can be used to define a transformation from a concrete domain into a more abstract domain.

Rosetta domains are arranged semantically in a lattice over which we have verified the existence a *Galois connection*. The net result is that abstractions performed by functors in the lattice are safe by definition or can be verified as safe within the lattice. By traveling up the lattice, functors can ensure an increase in abstraction resulting in less precise, yet equally valid models. Here the term *precise* simply means that you can learn less from a model, not that what you learn is less accurate. By making a model less precise, we make that model easier to analyze.

Model safety is critical as it assures that the results of analyzing the abstract model are applicable to the concrete model. Safety ensures that loss of precision

does not result in loss of accuracy. As an illustrative example, Ohm's Law is a safe abstraction of a circuit. Although it does not consider all the details of the circuit information derived by analysing a circuit using Ohm's law still applies.

Although precision is good, having a model that is too precise prevents efficient analysis. By removing detail, an abstraction always results in a less precise model that may be more easily analyzed. Ohm's law is less precise than an full-blown, atomic-level circuit model. However, it is also solvable in reasonable time without the use of high-performance computing systems.

## 6.2 Symbolic Evaluation

Among the most popular analysis techniques for complex systems is simulation. Raskell provides a built-in functor for transforming state-based specifications into simulation models for our comonadic, symbolic simulation environment. The simulator is symbolic because it does not need to fully evaluate a facet state to move forward in time. An incomplete solution can be used as a kind of state descriptor representing a collection of states with common characteristics.

Any Rosetta specification written in the `state_based` domain or one of its subtypes can be projected into the symbolic evaluation system. As `trusted_state_based` is an immediate subtype of `state_based`, the specifications written in this document have safe abstractions in the comonadic simulation domain. The functor for synthesizing these simulations is available in the comonadic simulation package.

While any `state_based` specification can be mapped to the comonadic simulator, not all specifications can be simulated in the traditional sense. If the facet or component definition does not have enough information to be solved for a next state, or exists in a form for which no solver exists, the simulator will not generate suitable results. Of the specifications in this document, all are simulatable although some (`UnClassifiedEnv` and `ClassifiedEnv`) must be stubbed out for meaningful simulation to occur.

Even when simulation is not directly available, the comonadic model can be used for formal verification using any of a number of theorem provers. We know of monadic specification translations to the Isabelle [2] proving environment although the comonadic specification translation has not been completed at this time. There are no technical issues to overcome, we simply have not had demand for this capability sufficient to justify resources. Having said that, a simulator that has realizations in both executable code and in theorem proving environments would prove exceptionally useful.

## 6.3 Model Checking

To fully ensure correctness of the `TrustedSystem` specification and its components, the entire state space must be explored. There is no way to guarantee coverage

in simulation for reasonably sized systems, yet even low probability events that cause errors must be detected. Particularly given the nature of attacks where an adversary does not rely on random occurrence. While simulation cannot achieve this, model checking and theorem proving can.

When Rosetta specifications are model checked, we again use a functor to perform an abstraction on a specification. In this case, instead of targeting a native Raskell simulator, we target a third party model checker, SAT solver or other decision procedure. Within Raskell, a semantic algebra is written that transforms the Rosetta model into a model appropriate for input to the selected third party tool. This semantic algebra corresponds to a functor and accomplishes the same thing in the operational Raskell environment. Furthermore, the Raskell environment provides extensive support for writing and verifying these semantic algebras.

We have targeted the zChaff [3] SAT solver and the SPIN[4] model checker with successful results. The zChaff solver was used to automatically check specifications as required in this experiment and to automatically generate text vectors for external simulation tools. The SPIN model checker has been targeted with various models along with temporal logic specification of correctness conditions.

## 6.4 Theorem Proving

Like model checking, theorem proving provides a capability that can completely explore the state-space of a system. Instead of enumerating the entire state space, theorem provers explore the state space symbolically by transforming mathematical expressions. Although far more difficult to use, theorem provers can provide quite concise results much faster than model checkers.

Any Rosetta specification can be transformed into a single, monolithic model for an automatic or semi-automatic theorem proving environment. One simply finds a denotation from Rosetta to the appropriate theorem proving language, applies the denotation function, and uses the theorem prover for analysis. However, our experience with earlier languages such as VSPEC and earlier Rosetta instantiations suggests that resulting models are unwieldy and difficult to use.

A more effective approach for analysis is generating abstract models for specific problems. Instead of generating a single, monolithic model for all proofs, use an abstraction function generate a smaller model specific to the proof being performed. The abstraction function must be verified, however it is necessary to do this only once for each abstraction function.

Theorem provers are targeted in exactly the same manner as model checkers. A semantic algebra is constructed that generates output for the theorem prover from a Rosetta input. The same Raskell support systems for generating inputs to model checkers is used to generate inputs to theorem provers (and the comonadic stimulator).

We have targeted the PVS [5] theorem proving environment from Rosetta specifications to support component comparison and verification. Our current work looks at the Isabelle prover due to its availability in the public domain and multiple logic packages. Our work using Isabelle is in early stages, but is quite promising.

## 6.5 Assertion Generation

A final technique akin to verification is synthesis of assertions for models. An assertion is simply a Boolean predicate that is checked when encountered in an executing program or simulation. If the assertion is true, then it has no effect. If it is false, it causes the program or simulation to react and report an error. Such reporting may range from completely halting the system to simply reporting the failed assertion.

We choose to examine assertions due to their effectiveness in supporting testing and run-time error detection. The PSL standard provides a property specification language independent of any particular application language and has hooks into SystemVerilog, SystemC, and VHDL. Rosetta generated assertions may target PSL or any other assertion language.

The Raskell assertion generation system has only been prototyped and never fielded in a production system or experiment. Assertions can be generated using the same semantic algebra techniques used for simulation, model checker or theorem prover model generation.

## 6.6 Raskell Thoughts

The Raskell analysis system is more an analysis toolkit than a single analysis engine. In each of the previous examples, Raskell is used to generate analysis models rather than perform analysis directly. The single exception being the simulation tool that shares the same underlying structure with Raskell.

We do not pretend that we can predict every way models in a heterogeneous specification language like Rosetta will be analyzed. So much so that we are frequently confronted with new analysis requirements that we could not anticipate. Thus, Raskell provides the user with tools for writing their own analysis tools or targeting existing analysis tools. If one understands the transformation that must be made, writing a semantic algebra using Raskell can be a simple task.

By defining functors in the domain lattice and using them in conjunction with Raskell, the transformations performed on Rosetta specifications can be proved to be sound. For certification purposes, this is critical as evaluators must be certain of the safety of any information transformation involved in the verification process. Verifying functors and transformations is not always a trivial

task. However, the formal semantics of Rosetta makes the task tractable when it could not even be attempted in other languages.

Finally, the same Raskell infrastructure used to analyze systems has also been used to synthesize hardware from Rosetta models. Synthesizable VHDL is generated from Rosetta specifications and fed to commercial synthesis tools that generate FPGA implementations. Although synthesis is less mature than analysis, we find the same underlying infrastructure useful for both.

## 7 Discussion

In this white paper we have provided an example using Rosetta to model trust and security. We defined a vocabulary for trust and a domain model for trusted components. We then used these Rosetta constructs to define a simple system and explore various trust and security properties. Although specifics of model verification are omitted, we presented several mechanisms for performing verification enabled by the Raskell analysis subsystem.

The models you see were written over the period of about 4 hours by an expert in Rosetta specification. They can be dramatically improved by using more sophisticated Rosetta modeling constructs and by adopting a more realistic model of trust and security. However, the specifications are quite similar other ongoing sponsored work in the secure systems area.

## References

- [1] Perry Alexander. *System-Level Design with Rosetta*. Morgan Kaufmann Publishers, Inc., 2006.
- [2] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [3] S. Malik, Y. Mahajan, and Z. Fu. Zchaff2004: An efficient sat solver. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004*, 2004.
- [4] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [5] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *Proc. of 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.