A Software Architectural Approach to Security By Design

Arnab Ray Rance Cleaveland
Fraunhofer USA Center for Experimental Software Engineering
University of Maryland at College Park
{arnabray,rcleaveland}@fc-md.umd.edu

Abstract

This paper shows how an architecture description notation that has support for timed events can be used to provide a meta-language for specifying exact communication semantics. The advantages of such an approach is that a designer is made fully aware of the ramifications of her design choices so that an attacker can no longer take advantage of hidden assumptions.

1 Introduction

The recent National Cyber Security software task force [2] reports defects in software as being one of the major causes of software vulnerability in computer-based systems. Defects can be introduced at all stages of the life-cycle – however, the earlier they are detected and prevented from propagating downstream, the greater are the savings in terms of time and cost.

Design vulnerabilities result from design choices during system construction. An attacker can compromise the system if he is able to interact with the system in ways that were not anticipated by its designer. The problem is made even more complicated when real-time considerations need to be factored in. For example, in denial of service (DOS) attacks, attackers masquerading as genuine users can consumer resources by initiating transactions they do not intend to complete, thereby degrading real-time responsiveness to legitimate users. The principle behind successful DOS attacks is the system assumption that all users are genuine and the consequent failure of the system to make a distinction between legitimate and illegitimate users.

As a solution to the problem of security threats due to imprecise expression of system interaction dynamics, this paper advocates the use of a timed architectural design notation called Timed Architectural Interaction Diagrams (TAID) for formally specifying system interaction. In addition to being mathematically precise, TAID is executable, thereby enabling rigorous analysis at design time.

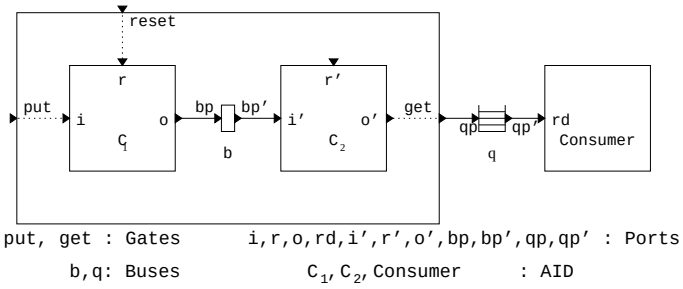

Figure 1. A TAID Architecture

2 A Short Introduction To TAID

From Figure 1, one may identify the following components of a TAID network: 1) TAIDs describing subsystems/components; 2) interfaces, containing read and write ports – each port is a conduit point for data for the subsystem surrounding by the interface; 3) connections between ports in a subsystem and ports in an interface (cf. the dotted lines in Fig. 1) called gates which enable one to selectively expose ports in an inner TAID; 4) buses, the "connectors" through which subsystems communicate with each other; 5) links from interface ports to buses.

The execution semantics of TAIDs are formalized in terms of a transition relation describing system-level execution steps. At the lowest level, a TAID component is a state-machine that comprises states and transitions. It can perform write transitions (i.e. output a value to a "write" port), read transitions (i.e. read in a value from a "read" port), or an internal transition. It may also engage in a clock tick transition (denoted by a label of 1 on the transition). Intuitively, the semantics of time is that it cannot advance as long as there are internal computation steps still left to be performed. All the actions that take place between two consecutive ticks are assumed to be instantaneous.

Buses are the most critical elements of the TAID paradigm: they can be looked upon as transducers that convert transitions of incident components (i.e. components who are connected to them) to system-level transitions. As

such, they have two responsibilities: the transfer of data between senders and receivers, and the synchronization of sender/receiver transitions, depending on the semantics of the interaction mechanism.

More precisely, a bus in TAID consists of an interface I, a set of bus states B (i.e. valuations of the data structures that represent the snapshot of a bus at any particular instant), an initial bus state (b_{init}), and two kinds of transitions – communication and timed.

A communication transition (which encapsulates an instantaneous transfer of data) is of the form $\langle b, WV, R, W, RV, b' \rangle$ written as: $b \xrightarrow[WV\ R]{W\ RV}_M b'$.
The way to read this transition is as follows: "if the bus is in state b, and subsystems connected to the bus enable write transitions as indicated in WV and read transitions as enabled in R, then the bus fires read transitions as indicated in RV and write transitions as indicated in W and goes to state b'." This firing of selected read and write transitions in systems connected to the bus is also done atomically: thus one bus transition may "consume" several transitions from the components connected to it. Also, "writing" to a bus is interpreted with respect to components connected to a bus: so write ports on a subsystem are connected to write ports on a bus, and similarly for read ports.

The sets WV, R, W and RV deserve further comment. WV contains pairs of the form $\langle w, v \rangle$ where w is a write port on the bus and v is a value. Intuitively $\langle w, v \rangle \in WV$ if there is a writer connected to the bus on its port w that wishes to write value v to it. Similarly, $r \in R$ means that there is a reader connected to the bus on its port r that is interested in reading. The bus then, out of these enabled transitions, chooses writers whose port values are stored in W and readers as indicated by RV where $\langle r, v \rangle \in RV$ if reader connected to port r gets value v.

As mentioned before buses in TAID also contain timed transitions of the form $\langle b, b' \rangle$ where b and b' are bus states.We write timed transitions as: $b \xrightarrow{1}_M b'$.

Mathematically, we define the semantics of TAIDs in the Structural Operational Semantic (SOS) style where the definition of the transition relation of a TAID is given inductively using inference rules that explain how transitions of subsystems are combined to form transitions of systems. Each system state records current state information for each component and bus, and the initial state should contain the initial states of each component and bus.

In this paper we do not formally define the semantics of Timed Architecture Interaction Diagrams (TAID)—the interested reader is asked to refer to [1] for the operational semantics of the non-timed version of TAID. The paper that elaborates on the semantics of the timed part is currently under review.

3 Security Modeling by TAID

In this section, we demonstrate the use of TAID for precisely encoding assumptions of system communication and show how that exercise leads to the uncovering of subtle security bugs.

One of the simplest modes of communication is *biparty synchronous handshake*. Its semantics is that if a writer wishes to perform a write action on a name and some reader wants to do a corresponding read action, then the writer and reader coordinate i.e. the two actions handshake and a communication is said to take place.

Though the classical semantics of biparty synchronous handshake is that it is instantaneous, in the real world, communication usually takes a non-zero amount of time. Hence we have what is known as the *timed* biparty synchronous handshake.

In order to provide an unambiguous semantics to this kind of communication, the question we need to ask is: "What exactly is the order of events that lead to a successful timed handshake?" Two alternative scenarios may be considered. In the first: 1) δ time units are allowed to elapse. 2) The writer does the $write$ action and is unblocked. 3) The reader, at the same instant, does the $read$ action and is unblocked. In the second: 1) The writer does a write action and is then blocked. 2) δ time units is allowed to elapse. 3) The reader does the $read$ action and both writer and reader are unblocked.

Thus we see two different ways of performing a handshake. This can lead to different attack scenarios as will be seen later.

Let us see how TAID provides a metalanguage for representing these two flavors of biparty synchronous handshake by encapsulating each of their semantics as a bus, thus enabling us to distinguish between them. Looking at the obligations we need to discharge in order to provide a definition for a bus, we have to define what the bus states are (i.e. the snapshots of the internal data structures), the initial data structure, the set of read and write ports for the bus and the bus transitions — both timed as well as untimed.

The first semantics is given in Table 1. The first line in the table says that the bus's interface has a set $\mathbb{W}$ of write and a set $\mathbb{R}$ of read ports. The second line in the table defines the bus states. There is a distinguished bus state: b_{init} and also a set of bus states, each of which maintains in the variable i the number of clock ticks already elapsed at that state.

Rule 1 states that if there exists at least one writer and a reader, then the counter (representing the clock) is activated, and vide Rule 2 the counter is incremented until the counter value reaches the value δ at which the delay has elapsed, and the handshake is allowed to proceed.(Rule 3).

For the second semantics (Table 2), who gets to write and

Table 1. Synchronous Handshake with δ delay–Version 1

I=$\langle \mathbb{W}, \mathbb{R} \rangle$
$B_\delta = \{b_{init}\} \bigcup \{i \mid i \in \{0..\delta\}\}$
1. $b_{init} \xrightarrow[WV\ R]{W\ RV} 0$ iff
$\exists \langle w, v \rangle \in WV. \exists r \in R \wedge W = \emptyset \wedge RV = \emptyset$
2. $i \xrightarrow{1} i+1$ iff $i < \delta$
3. $\delta \xrightarrow[WV\ R]{W\ RV} b_{init}$
iff $\exists \langle w, v \rangle \in WV. \exists r \in R \wedge W = \{w\} \wedge RV = \{\langle r, v \rangle\}$

Table 2. Synchronous Handshake with δ delay–Version 2

I=$\langle \mathbb{W}, \mathbb{R} \rangle$
$B_\delta = \{b_{init}\} \bigcup$
$\{\langle i, w, r, v \rangle \mid i \in \{0..\delta\}, v \in \mathbb{V}, w \in \mathbb{W}, r \in \mathbb{R}\}$
1. $b_{init} \xrightarrow[WV\ R]{W\ RV} \langle 0, w, r, v \rangle$
iff $\exists \langle w, v \rangle \in WV. r \in R \wedge W = \emptyset \wedge RV = \emptyset$
2. $\langle i, w, r, v \rangle \xrightarrow{1} \langle i+1, w, r, v \rangle$ iff $i < \delta$
3. $\langle \delta, w, r, v \rangle \xrightarrow[WV\ R]{W\ RV} b_{init}$ iff $W = \{w\} \wedge RV = \{\langle r, v \rangle\}$

who gets to read is decided at the beginning (and not at the end of the δ time units). Hence here we need to keep track of the writer and its value along with the reader. This leads us to define a set of bus states, of form $\langle i, w, r, v \rangle$ where i denotes the number of clock ticks already elapsed at that state, w denotes the write port that was selected for writing among the multiple enabled writers, r denotes the read port that was selected among the multiple read ports and v stores the value being written. At the end of δ time units, the bus looks at the value of w and v it has stored, enables a writer and reader on the basis of that and then transfers its stored value v to the selected reader.

Let us now consider the security vulnerabilities inherent in these two definitions. In the communication defined in Table 1, the communication is initiated once the WV set and the R sets are non-empty, i.e., there is at least one writer and one reader that want to communicate. Let δ time units pass. During this period, the original writer or reader or both may have withdrawn (i.e. no longer be present in the WV and R sets respectively) and been replaced by some other writer and reader. The communication will still complete because the bus only checks to see if there is *some* writer and reader present at the end of the delay, not whether the writer and reader present are the same writer and reader who initiated the communication δ time units ago. Hence this communication is open to a kind of *spoofing* attack.

Now let us look at the second bus implementation of timed biparty handshake as given in Table 2. Here the original writer who initiated communication and its data value is recorded in the bus state along with the selected reader and so at the end of δ time units, the identity of the writer/reader can no longer be substituted as in the first example i.e. the writer/reader who initiated the communication will be the one that writes/reads at the end. Hence it is not susceptible to the spoofing attack the previous example was, but it still has a vulnerability.

The attack scenario is this: Suppose component A is chosen by the connector to write/read among all the enabled write/read transitions. A is a malicious entity. It then intentionally removes itself from the communication but the connector is not even cognizant of the fact and completes, what is thinks, is a successful interaction. A has thus been successful in blocking off other benign writers/readers (a DOS attack) for at least δ time units, wasted δ units of communication resources, and the system, as a whole, is not even aware of this malicious activity.

The problem here stems from the fact that an attacker can take advantage of an environmental assumption made by the connector, the assumption being that "writers who wanted to write when the interaction began shall be there at the end when the communication terminates".

As a result, we need to refine our definition of biparty synchronous handshake so that the parties who initiate the communication are present when it ends. Our semantics thus becomes: 1) The writer does a write action but remains blocked. 2) δ time units is allowed to elapse. 3) The reader does the $read$ action and if the original writer and reader who were chosen initially are still present in the interaction, only then are both writer and reader unblocked. Otherwise, if the original writer and reader are not present, we flag a *communication failure*.

The bus definition for this is given in Table 3. Here, the set of bus states is augmented with a special error state called b_{err}. In Line 3, we perform a check to see if the (write port, value) tuple that was selected initially is still present in WV i.e. in the set of enabled (write port, value) tuples. We also check to see if the read port that was selected originally is present in R. Only if both the checks succeed do we then proceed with the handshake. If any one of them fails, then the bus transitions to an error state (Line 4); signifying that the handshake failed. The designer then has to check if it is possible, and if it is under what conditions, for the system to reach b_{err}.

However, even now we have a vulnerability. The assumption made during timed biparty synchronous handshake is that the writers/readers remain blocked for the duration of the communication. However the semantics of the bus given in Table 3 only ensures that the same writer and reader is present at the beginning and the end—it can-

Table 3. Synchronous Handshake with δ delay–Version 3

I=$\langle \mathbb{W}, \mathbb{R} \rangle$
$B_\delta = \{b_{init}, b_{err}\} \bigcup$
$\{\langle i, w, r, v \rangle \mid i \in \{0..\delta\}, v \in \mathbb{V}, w \in \mathbb{W}, r \in \mathbb{R}\}$
1. $b_{init} \xrightarrow[WV\ R]{W\ RV} \langle 0, w, r, v \rangle$ iff
$\exists \langle w, v \rangle \in WV. \exists r \in R \wedge W = \emptyset \wedge RV = \emptyset$
2. $\langle i, w, r, v \rangle \xrightarrow{1} \langle i+1, w, r, v \rangle$ iff $i < \delta$
3. $\langle \delta, w, r, v \rangle \xrightarrow[WV\ R]{W\ RV} b_{init}$ iff
$\langle w, v \rangle \in WV, r \in R \wedge W = \{w\} \wedge RV = \{\langle r, v \rangle\}$
4. $\langle \delta, w, r, v \rangle \xrightarrow[WV\ R]{W\ RV} b_{err}$ iff $\langle w, v \rangle \notin WV \vee r \notin R$

not guarantee that the writer and reader selected remained blocked throughout δ time units.

As an example, let w be selected by the connector to do the write operation, its value v be taken in by the connector and the writer kept blocked till the delay finishes. However let us assume that, as before, w times out (i.e. removes itself from the interaction) and then just before the delay finishes, it re-enters the interaction with a different value. Now when the reader gets the value v at time t, it may presume that the value available at time t from the writer is v when in reality it is not (since the writer has now re-entered the interaction with a value v'). Hence the reader gets an inconsistent view of the system and the semantics of the biparty handshake is violated without the connector being cognizant of the fact. (The violation occurs because the assumption that a writer remains blocked throughout a handshake is not enforced by the communication semantics)

While this condition itself may not be an "attack" in the classical sense of the term (i.e., no privilege is being acquired or services blocked) it enables a component to send the system into an *inconsistent* state in a silent, undetectable fashion. Conventional attacks may then be crafted at the component level that may take advantage of this inconsistency.

To take care of this, we need to craft a more refined handshake semantics with the assumption that a writer/reader cannot withdraw from the communication and come back before the delay expires. In other words, once a writer and reader are selected for communication, they have to be blocked for the duration of δ time units in order for the communication to be properly completed. Here: 1) The writer does a write action but remains blocked. 2) δ time units is allowed to elapse. After each clock tick, the connector checks to see if the original writer and the reader who initiated the communication is present. If it is not then a *communication failure* is flagged. 3) The reader does the $read$ action and if the original writer is still present in the interaction, only then are both writer and reader unblocked. Otherwise, if the original writer is not present, we flag a *communication failure*.

We do not write down the formal bus definition in this paper for lack of space, but the intuition behind the modification that needs to be made with respect to the rules in Table 3 is that at the end of every time transition, we transition to a special "hold" state where we check to see if the originators of the interaction are still present. If they are, then the bus is ready for the next tick and if not, a transition to b_{err} takes place.

4 Conclusion

This paper thus shows how a timed, architecture description language (TAID) may be used to provide mathematically precise formalisms for representing security-critical communication assumptions. Being an executable notation, TAID allows us to use these definitions as basis for simulation-based security analysis and model-checking at design time.

Bus specifications precisely encapsulate assumptions about the behavior of inter-component interactions, and our work suggests that this level of detail is necessary in order to consider security issues at design time. These specifications in fact serve two purposes. For component designers, having such assumptions made explicit enables them to determine what security countermeasures must be included within their components, and how much the medium can be relied on. For architecture designers working with traditional platform specifications, TAID-style bus definitions are unlikely to exist. Such designers can, however, explicate their assumptions about bus behavior using TAID specifications and use these as a basis for assessing whether their understanding of the platforms to be used is correct or not.

Future work lies in creating a library of application-specific policy specifications using TAID and providing tool support for TAID architectural descriptions.

References

[1] A. Ray and R. Cleaveland. Architectural interaction diagrams: Aids for system modeling. *Proceedings of the International Conference on Software Engineering,(ICSE)*, pages 396–406, 2003.

[2] S. Redwine and N. Davis. Software process subgroup of the task force on security across software development lifecycle. *National Cyber Security Summit*, 2004.