CoBFIT: A *Co*mponent-*B*ased *F*ramework for *I*ntrusion *T*olerance *

HariGovind V. Ramasamy Adnan Agbaria William H. Sanders
Coordinated Science Laboratory, University of Illinois
1308 W. Main Street, Urbana IL 61801, USA
{ramasamy, adnan, whs}@crhc.uiuc.edu

Abstract

In this paper, we present the architecture of CoBFIT, a component-based framework for building intrusion-tolerant distributed systems. The CoBFIT framework, by virtue of its design and implementation principles, can serve as a convenient base for building components that implement intrusion-tolerant protocols and for combining these components in an efficient manner to provide a number of services for dependability. For example, in this paper, we describe the CoBFIT implementation of a prototype intrusion-tolerant group communication system that includes services such as reliable multicast, group membership management, and total ordering of multicast messages.

1. Introduction

Traditional security aims to build systems equipped with defense mechanisms that safeguard the systems against attacks. It also tries to identify vulnerabilities in components either by rigorous testing before deployment or from successful attacks after deployment, and patches them. Although this approach has been effective in handling many attacks, practical experience shows that most systems remain vulnerable, at least to some extent. This is particularly true for distributed systems whose correct functioning can depend on the possibly complex interactions of software running on many nodes. The concept of "intrusion tolerance" (e.g., [8]) acknowledges the existence of such vulnerabilities and assumes that over the course of time, a subset of them will be successfully exploited by intruders. The focus of intrusion tolerance is to ensure that systems will remain operational (possibly in a degraded mode) and continue to provide core services despite faults due to intrusions. Traditional security and intrusion tolerance can be combined to provide an effective "defense-in-depth" strategy for achieving dependability in the face of attacks, failures, or accidents.

In this paper we describe *CoBFIT*, a *Co*mponent-*B*ased *F*ramework for *I*ntrusion *T*olerance. The goal of CoBFIT is to provide a robust, flexible, reusable, reconfigurable, and portable software framework that could serve as a platform for building and testing a variety of intrusion-tolerant distributed systems without having to re-implement the common support for each of those systems. A key challenge to achieving this goal is that of finding a way to separate the common support for intrusion tolerance from intrusion-tolerant systems. Examples of commonly needed features for supporting intrusion tolerance include secure buffer management, cryptography, coordination of inputs from multiple intrusion detection mechanisms, replication over multiple nodes, primitives for secure communication, and Byzantine agreement.

To provide a concrete focus and a basis for demonstrating the applicability of the CoBFIT framework to the creation of intrusion-tolerant systems, we consider intrusion-tolerant applications from a specific domain, namely the domain of applications based on group communication. Group communication [18] is a key building block for several dependable systems. Group communication systems (GCSs) can be used to ensure state consistency among processes that constitute a group in the presence of benign and malicious failures. Building intrusion-tolerant group communication protocols that can tolerate the malicious corruption of some group members is an active research area [7][6][13][20][21] and is among the objectives of our research. However, the focus of this paper is to highlight the infrastructure support needed in building intrusion-tolerant group communication protocols, and to describe how the CoBFIT framework provides that support. The framework can serve as an excellent platform on which various design choices may be implemented and evaluated during the building of intrusion-tolerant group communication protocols. In this paper, we describe a prototype intrusion-tolerant GCS that was built using the CoBFIT framework.

* This research has been supported by DARPA contracts F30602-00-C-0172 and F30602-02-C-0134.

2. Related Work

We use group-communication-based intrusion-tolerant systems to demonstrate the validity of our framework. In this section, we compare CoBFIT with other group-communication-oriented frameworks that focus on protocol composition, customization and flexibility. However, we would like to point out that CoBFIT is a more general framework that can potentially facilitate the creation of other kinds of intrusion-tolerant systems. For example, the CoBFIT framework components can also facilitate the building of an intrusion-tolerant database system, such as the one mentioned in [15].

Horus and Ensemble [4] are successive generations of highly configurable GCSs. They offer an environment in which software modules called *microprotocols* communicate using events, and can be combined to provide different QoSs. Micro-protocols are structured as multiple layers using stack-like composition models, and events are FIFO streams that are used for communication between these layers. In contrast, protocols in CoBFIT (implemented as service components) can be organized and made to interact with each other in arbitrary ways, and handle only the events they are interested in. Ensemble also has security protocols that provide customizable cryptographic techniques. However, its focus is on secure group communication, rather than on intrusion tolerance.

Cactus [12] implements finer-grained (compared to Horus and Ensemble) microprotocols that provide individual properties of the target service. Like CoBFIT, Cactus structures protocol objects using events and event handlers to enhance the configurability by minimizing explicit references between modules and prevents unnecessary event processing by using a publish-subscribe model. Cactus was also a pioneer in applying configurability and customization principles specifically for dependability. The CoBFIT framework follows that tradition and aims to provide more significant infrastructure support for intrusion tolerance by incorporating some abstractions and primitives commonly used in the development of intrusion-tolerant services in its core architecture.

Appia [16] is a protocol kernel whose main focus is on facilitating the specification and implementation of inter-channel constraints while retaining the flexibility of Ensemble. It uses an open event model and allows protocols to subscribe only to events they are interested in. While CoBFIT shares Appia's ability to provide flexibility in dynamic composition of the protocol combinations, an important difference between Appia and CoBFIT is that CoBFIT provides explicit infrastructure support aimed at intrusion tolerance.

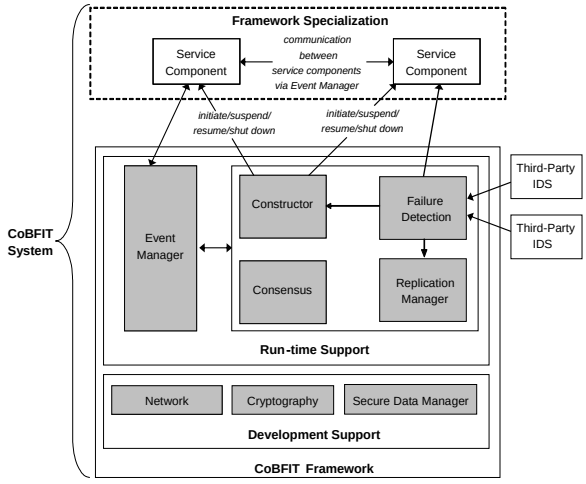

Figure 1. The CoBFIT Architecture

3. The CoBFIT Framework

3.1. Overview of the CoBFIT Architecture

Figure 1 depicts the CoBFIT architecture, in which intrusion-tolerant services are built using the support provided by the CoBFIT framework. The set of CoBFIT services along with the CoBFIT framework constitute a *CoBFIT system*. Both the framework and the services are realized by a collection of components. A component is a coherent, encapsulated part of the CoBFIT architecture. Framework components implement the structure of intrusion tolerance in the form of abstractions, primitives, and supporting software mechanisms that are commonly needed for the creation of intrusion-tolerant services. Some framework components provide development support (i.e., they are useful in the development of service components), while others provide run-time support. Service components implement the functionality of intrusion tolerance that is specific to a particular domain of applications. At a high level, a service component will be the implementation of an intrusion-tolerant protocol or algorithm that provides a set of desired intrusion tolerance properties. The particular protocol or algorithm used for implementing a service component could vary depending on the service specifications. Our work was inspired by [5], which describes a framework for motion control that enforces a clean separation between the structure and functionality of motion control.

It is important to note that implementing a service in the CoBFIT framework does not necessarily make the service intrusion-tolerant. The service will be intrusion-tolerant only if the protocol or algorithm upon which the service is based is intrusion-tolerant by design. The CoBFIT framework merely facilitates the development of the service

components (development support) and the run-time adaptation of the services based on perceived faults (run-time support).

Using the CoBFIT framework, a user will be able to choose the implementation of a service component from a set of multiple available implementations (all of which expose the same external interfaces), and can switch the choice during run-time. This flexibility could be exploited for performance and/or intrusion tolerance benefits. The key idea that allows us to achieve such flexibility is that of enforcing the interaction between various service components strictly through events. The specification of a service component consists of the set of interfaces it provides for other components and the set of interfaces that it requires from other components. The service component implements the set of interfaces it provides for other components via a set of event handlers. Events and event handlers are both implemented as objects. The CoBFIT framework follows an open event model (like Appia [16]), which makes it possible to define and use new events in new service components, while allowing the older service components to continue executing correctly, unaware of the newly defined events.

In this section we describe the completed and planned design of the framework components through which the service components operate.

3.2. Framework Components

Event Manager Event processing in the CoBFIT architecture is based on the publish-subscribe model. At the time of their instantiation, the service components publish the events they generate to the Event Manager. They also subscribe to the events that they are interested in receiving (handling) from the Event Manager. Thus, at initialization, the Event Manager knows the set of all events, and the handlers bound to them. Note that these bindings can change dynamically when one or more of the service components are replaced or reconfigured during run-time.

The Event Manager component is based on the Reactor Software pattern [24] for event handling. The Event Manager detects and demultiplexes events (such as input-output-based port-monitoring events, timer-based events, signal events, exception events, and notification events that are used for component-to-component communication), and dispatches them to the service components that have subscribed to those events. When multiple service components have subscribed to the same event (for example, many service components may be interested in an intrusion-detection event) then the order in which the event handlers in various service components are invoked may be important. To determine the order, the Event Manager takes into account the dependencies between the various service components. It should be noted here that there will not be any dependencies among the implementations of various service components. The event-based interactions are meant to prevent any explicit references between service components and hence any dependencies in their implementations. The Event Manager, when it is instantiated (by the Constructor component, as we will explain below), is handed a directed graph giving the dependencies between various service components. The directed graph, called the *dependency graph*[1], has the service components as its nodes. An edge from one node to another will indicate that the correct operation of the service component corresponding to the first node is dependent on the correctness of the properties provided by the service component corresponding to the second node. We give an example dependency graph in Section 4.2. The Event Manager uses the knowledge about the dependencies among the various components and the event-handler-to-event bindings to determine, for each published event, the order in which the different event handlers that have subscribed to that event must be invoked. The order of invocation is determined as follows: for any two components c_1 and c_2 that subscribe to an event, with c_2 dependent on c_1, the event handler in c_1 is invoked first followed by the invocation of the event handler in c_2. Thus, the Event Manager implements an automated way to translate the dependencies among the high-level properties that various components provide into low-level event-based interactions.

A disadvantage of Reactor-based event management is that it is synchronous, and hence could block when performing I/O operations. However, that problem can be circumvented by programming all event handlers as non-blocking I/O objects [1]. For this purpose, an event handler stores its own state and any relevant portions of the parent component's[2] state in memory using the Memento pattern [9] whenever it blocks on I/O, and returns control to the Event Manager's main event loop. The Event Manager can then dispatch other event handlers. When the I/O is ready, the Event Manager calls back to the appropriate event handler, which can then retrieve its stored state and continue.

Constructor The Constructor component is responsible for the reconfigurability of a CoBFIT system. Reconfigurability is one of the design principles of CoBFIT that resulted from the realization that different intrusion-tolerant systems require different sets of protocols that may have to be reconfigured (without requiring modification, recompilation, or re-linking of the program itself, or shutting down

1 The idea of dependency graphs is an outgrowth of the work in [11], which defines relations between properties as well as between components that implement those properties.

2 The *parent* component of a given event handler object is the component that instantiated the object.

and restarting of the CoBFIT system) when responding to attacks. It is a key capability that is required by intrusion-tolerant systems that adapt to attacks by changing the security posture and switching to increased levels of alertness (e.g., AITDB [15], ITUA [3]).

The Constructor is the only component that is directly instantiated by the user. The user will give the Constructor a list of service components that need to be included in a particular system configuration and the dependency graph for that configuration. Then, the Constructor creates the other framework components, followed by the service components. When the Event Manager is instantiated, the Constructor hands the dependency graph to the Event Manager component. The design of the Constructor is based on the Component Configurator design pattern [24]. All CoBFIT service components implement a uniform component management interface through which they can be configured and controlled by the Constructor. The interface defines operations for (re)initializing, shutting down, suspending the execution, and resuming the execution of a component. The Constructor maintains a component repository; it implements a mechanism that interprets and executes a script specifying which of the available components to link into and unlink out of the CoBFIT system dynamically. There can be multiple scripts specifying different adaptation strategies. In that case, the Constructor will implement the rules for choosing and executing the appropriate script based on the current security posture, attack type and attack severity.

The Constructor component is useful because it has the ability to alter the configuration of a CoBFIT system. It is important to ensure that this capability is used only when necessary. For example, if the trigger for reconfiguration is an intrusion alert, then an alert that is only a false alarm will cause the system to be reconfigured and perhaps switch to expensive protocols, thereby affecting performance. On the other hand, the reconfiguration trigger should not always depend on directives from the system administrator, since automatic adaptation is necessary to ensure dependability in the face of attacks. We are currently investigating decision procedures that strike a balance between automated reconfiguration and unnecessary reconfiguration. These decision procedures, when incorporated into the Constructor component, will facilitate graceful adaptation for switching service components at runtime in a coordinated manner.

Network All CoBFIT service components send messages to and receive messages from the network through the Network component. Messages are considered as a special type of CoBFIT events. While normal (local) events are used for communication between components belonging to the same process (CoBFIT system), messages are events that are used for inter-process communication. Inter-process communication includes communication between a CoBFIT system and another CoBFIT system, and more generally, the communication between a CoBFIT system and the outside world. A service component at a process that wants to send a message to its peer service component at another process does so through the Network component. The service component conveys the payload to be transmitted through a local event to the Network component. The Network component generates a message with the payload and appropriate header (indicating the service component to which the peer Network component at the recipient process should convey the payload), and transmits the message on the network. At a peer process, the receipt of any message will be an I/O event to which only the process's Network component subscribes; hence, the Event Manager at the process will invoke the event handler in the Network component that handles the event. The event handler will examine the header and convey the payload to the corresponding service component by generating a local event.

The Network component is designed in accordance with the Wrapper Facade design pattern [24] and consists of classes that encapsulate platform-specific low-level network functions and data within a type-safe, portable, object-oriented interface. The component also serves as an adapter and hides the underlying transport mechanism from the service components, thereby providing the flexibility to change the transport mechanism without having to modify the service components. It adapts the uniform networking interface that all service components expect to the actual interface provided by the underlying transport protocols, and thus will enhance the reusability of the service components.

Secure Data Manager Complex, even verifiably correct intrusion-tolerant protocols do not ensure security unless they are correctly implemented, but it is easy, even for experienced developers, to code implementation flaws into a verifiably correct protocol design. To help alleviate this problem for the developers of service components, the Secure Data Manager component provides primitives and data structures that facilitate secure and efficient data management. More precisely, it provides "safe classes" whose methods are essentially wrappers around C/C++ standard library functions (a representative list of such calls is given in [26]) that are generally perceived to be unsafe, because of their high susceptibility to buffer overflows and format string problems. It also provides classes that manage messages efficiently with operations for buffering messages (for transmission/retransmission or upon reception), marshalling and demarshalling, fragmenting and reassembling, and reordering messages received out of sequence. For efficient message manipulation when transferring data between components, the classes will employ "shallow" copy instead of "deep" copy, i.e., pointers to a single reference-counted message copy (instead of multiple copies) are used. The message operations are made safe through sanity checks

made wherever necessary (e.g., bounds checks on array and pointer references, or a message format check on received messages).

Cryptography Several intrusion-tolerant architectures (e.g., those that rely on Byzantine agreement [3]) use cryptography for secure communication not only with the outside world but also within the system, so that insider attacks and compromised subsystems can be tolerated. The Cryptography component facilitates the deployment of cryptography in service components. However, the focus of this component is not to invent and implement new cryptographic algorithms or new cryptographic protocols. Well-scrutinized implementations of those protocols already exist in many widely used cryptographic libraries. Each of the libraries has its own merits and demerits (a comparative study is given in [26]). We thus believe it is best to leave the choice to the discretion of the intrusion-tolerant system builder. However, the service components that utilize cryptography must be reusable even if the choice of cryptographic library changes. To meet those objectives, the Cryptography component provides a uniform way to access different cryptographic libraries. In essence, it defines interfaces for common cryptographic operations (e.g., signing/verifying a message, encrypting/decrypting a message, or computing the digest for a message) that could be invoked by the service components and adapts the interface of the chosen cryptographic library to the defined interface. Since service components do not make direct calls to the third-party cryptographic library, but instead invoke interfaces in the Cryptography component, the service components are reusable even if the choice of cryptographic library is changed later.

Failure Detection Intrusion-tolerant systems are often designed to withstand attacks and mask failures without having to detect the failure of a subsystem. However, that strategy is useful only up to a point, after which either enough subsystems have been corrupted or a critical subsystem has been compromised, rendering the whole system corrupt and unable to provide the required services. Hence, it is important to detect successful attacks so that the compromised subsystems can be replaced, repaired, or removed. Intrusion detection in the CoBFIT architecture can be internal, external, or both. Internal intrusion detection is done by any service component based on observed anomalies or deviations from the specifications. External intrusion detection employs third-party intrusion detection systems (IDSs). Many of these intrusion detection mechanisms (especially those in IDSs) are constantly updated based on new attacks. We would like for the implementation of the CoBFIT service components to be independent of the specific intrusion detection tools or mechanisms used. One of the goals of the Failure Detection component is to enforce a clean separation between intrusion detection and intrusion response mechanisms. Our design of the component is based on the Mediator software pattern [9], by which the Failure Detection component acts as the hub of communication for intrusion detection.

The Failure Detection component serves as the central sink for intrusion detection reports or suspect reports from internal or external intrusion detection sources. It processes reports from diverse sources and implements policies to determine which reports should actually lead to system adaptation. For such reports, it generates a failure detection event to which other components can subscribe; the components can then respond to the event based on the particular protocol or strategy that they implement. For example, the Constructor component can subscribe to the event and implement the logic to appropriately reconfigure the system based on the severity of the failure given by the failure detection event type. In response to detection of a malicious attack, the Constructor could dynamically unlink service components that can tolerate only crash faults, and replace those components with Byzantine fault-tolerant versions.

By acting as the sole sink for intrusion detection reports or suspect reports from both external and internal detection mechanisms and as the sole intrusion/suspect notifier for other components, the Failure Detection component defines a uniform way in which all components can receive intrusion/suspect alerts. That will make the implementation of service components independent of the specific intrusion detection tools or mechanisms used. However, interfacing to multiple third-party IDSs will be challenging. Some IDSs may be in the user space, while others may be in the kernel space; some may be host-based, and others may be network-based; some may signature-based, and others may be anomaly-based. Another challenge will be that of coming up with appropriate policies for determining which intrusion reports or which combinations of reports should actually trigger system adaptation. A particular concern in this regard is the relatively high frequency of false positives for which IDSs are notorious. We are currently investigating policies for analyzing intrusion reports that minimize the false positives that trigger system adaptation, while ensuring that appropriate adaptation mechanisms are triggered in the face of real attacks.

Replication Manager Replication by redundancy is an important design primitive used in many fault- and intrusion-tolerant systems to improve the resilience to faults [17]. The Replication Manager component in the CoBFIT framework provides operations to manage a replicated application. Each replica will constitute a CoBFIT system that contains a Replication Manager component. The Replication Manager components at the replicas communicate with each other to translate high-level dependability requirements given at run-time to particular replication configurations. Dynamic replication management involves re-

configuring the replicated application in response to faults and changes in desired dependability requirements. Thus, while the Constructor dynamically alters the configuration of the CoBFIT system corresponding to a single process, the Replication Manager components help dynamically reconfigure multiple CoBFIT systems corresponding to a replicated application. The choice of how to alter the replication configuration may depend on the types of faults to tolerate (crash, omission, timing, or Byzantine), the styles of replication to use (active, passive, semi-passive, or semi-active), and the location of the replicas, in addition to factors specific to the particular application being replicated. In Section 4.3, we describe the functionality of the Replication Manager using an example.

Consensus Fault-tolerant consensus is an important building block for many distributed services (such as replication and atomic multicast) [10]. The goal of the Consensus component in the CoBFIT framework is to provide a consensus primitive that can be used to construct such services. The Consensus component would provide multiple consensus protocol implementations, each implementation differing in the type of faults it tolerates and applicable for use in different replication strategies (active, passive, semi-active, or semi-passive). We describe one such consensus protocol called Lazy Byzantine Consensus that can be used for Byzantine fault-tolerant semi-passive replication in [19]. The Consensus component provides an interface that can be used by any service component to reach consensus.

3.3. CoBFIT Framework Implementation

Robustness Our goal is to make the CoBFIT framework robust in design and implementation. While we believe that it is impossible to build the framework to be fully free of vulnerabilities, our goal is to minimize the vulnerabilities. As detailed in Section 3.2, the design of many CoBFIT framework components is based on software patterns [9][24] that have been thoroughly examined by software practitioners. The implementation of the CoBFIT framework uses the Adaptive Computing Environment (ACE) toolkit [22][23]. The toolkit is open-source, widely used, and stable, and has proved to be a robust platform for building high-performance communication services. Of course, one can use a well-established design pattern and a robust programming environment, and still produce a software that crashes! We plan to evaluate the robustness of the design at the model level and assess the robustness of the implementation by thoroughly testing it.

Portability Redundancy by replication is useful in the case of benign crash failures, but needs to be coupled with heterogeneity to be useful in the presence of malicious attacks, because otherwise the attacker could use the same vulnerability to penetrate all replicas. One of the ways to achieve such heterogeneity is to use diverse operating systems[3]. To allow use with diverse operating systems, the CoBFIT framework and the service components should be easily portable. In order to meet that objective, the implementation of the framework should be independent of specific operating system support to the extent possible. The use of the ACE toolkit for implementing CoBFIT framework and service components facilitates that, since ACE provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of operating systems and platforms.

We have built a prototype CoBFIT framework that implements a subset of the CoBFIT framework design described in Section 3.2. In the remainder of this section, we present current and planned implementations for the CoBFIT framework components.

We implemented the Event Manager component by extending the ACE Reactor framework [23]. We defined a *CoBFIT_Event* class from which the class of any event object exchanged between CoBFIT components derives. The ACE Reactor framework implements the functionality to detect the occurrence of events from I/O handles, timers, and signals. We extended the functionality so that the Event Manager can detect and demultiplex any event object instantiated from the *CoBFIT_Event* class or its derived classes.

The reconfiguration capabilities of the Constructor component were built using the ACE Service Configurator framework [23], which is a portable implementation of the Component Configurator pattern. All CoBFIT service components derive from the *ACE_Service_Object* class, which defines a interface through which the Constructor can initialize, suspend, resume, or terminate a service. The Constructor itself derives from the *ACE_Service_Config* class, which provides the capability to parse and execute scripts specifying which services to dynamically reconfigure into an application.

The “safe” data insertion and manipulation operations of the Secure Data Manager component are drawn from the capabilities provided by the *ACE_Message_Block* class [22]. The operations are made safe through use of sanity checks such as bounds check and input validation. Multiple *ACE_Message_Block* objects can flexibly share data among themselves without the overhead of copying the data. The message manipulation mechanisms of the Secure Data Manager are constructed from the *ACE_Input_CDR* and *ACE_Output_CDR* classes. The classes provide methods for message operations, like linearization of data structures to/from raw memory buffers

3 Ideally, heterogeneity at the OS level should be coupled with heterogeneity at the CoBFIT system level (for example, by using N-version programming).

and marshaling/demarshaling of data using the Common Data Representation (CDR) format. The standard CDR format allows for correct interoperation in environments with heterogeneous compiler alignment constraints and hardware instructions with different byte-ordering rules, thus enhancing portability. The Secure Data Manager will be extended in the future to provide a comprehensive library of safe classes.

The Cryptography component in the prototype implementation serves as an adapter to the Cryptlib [2] cryptographic library. Currently, the adapter provides interfaces to RSA public-key operations, such as generation of key pairs and signing and verification of message buffers. The adapter also defines interfaces for creating message digests using the SHA-1 hashing functions, and for verifying whether the hash of a message buffer matches a given hash. Future work includes (1) providing interfaces for other hash algorithms (e.g., MD5) and other public-key algorithms (e.g., DSA and El Gamal) and (2) writing adapters for other common cryptographic libraries, such as OpenSSL, Crypto++, and BSAFE.

The Failure Detection component in our prototype has a fairly simple implementation. It accepts intrusion or suspect reports from internal intrusion detection sources (CoBFIT service components). The service components generate a *Suspect_Report* event, for which the Failure Detection component implements the sole event handler. The event handler analyzes these reports based on rules specific to the particular type of intrusion-tolerant system being implemented. If certain conditions specified by the rules are satisfied, the event handler generates a *Failure_Detect* event to which all CoBFIT components that adapt in response to intrusions subscribe. In Section 4.3, we describe the rules that were used to generate the *Failure_Detect* event in the prototype CoBFIT GCS. The component will be extended in the future to interface with and analyze reports from external intrusion detection sources, such as the Snort open source network intrusion detection system [14].

The use of ACE Socket Wrapper Facades [22] simplified the implementation of the Network component in our prototype. The set of wrapper facades provides classes for passive and active connection establishment, UDP-based connectionless messaging services, TCP-based connection-oriented messaging services, and datagram-based multicast/broadcast services.

Future work includes the implementation of the Replication Manager and the Consensus components that have not been implemented in the prototype.

4. Framework Specialization: Intrusion-Tolerant Group Communication

To demonstrate the utility of the CoBFIT framework for building intrusion-tolerant systems, we have implemented a prototype GCS that can be used to coordinate processes in the state-machine replication model [25]. In this example CoBFIT framework specialization, the CoBFIT services are implementations of intrusion-tolerant group communication protocols, such as reliable multicast, total ordering, and group membership (represented by the service components in Figure 1). Together, the service components and the CoBFIT framework form an intrusion-tolerant group communication system called the *CoBFIT GCS* (represented by the entire CoBFIT system in Figure 1).

One way to build an intrusion-tolerant application is to structure the application as a state machine, replicate the application on multiple nodes, and coordinate the replicas using the CoBFIT GCS. The replicas together form a *replication group* representing the application. Each replica forms an instance of the CoBFIT system (Figure 1), and the replication group is a set of CoBFIT systems communicating with each other. Each replica maintains application state information, and the group communication protocols of the CoBFIT GCS guarantee the consistency of replicated information across all correct members despite malicious corruption of some members. The CoBFIT component configuration is identical across all replicas, i.e., all replicas have an identical collection of components and the same implementation strategy for each component.

4.1. Service Components in the CoBFIT GCS

We present here the service components that make up the prototype CoBFIT GCS, and briefly describe the functionality that each service component provides. The intrusion-tolerant group communication protocols implemented by the service components are described in [20]. The protocols assume that no more than one-third members of the group are faulty. Here, our focus is not the protocols themselves, but to describe their implementation in the CoBFIT framework.

Group Membership The component implements an intrusion-tolerant group membership protocol for removing faulty processes from the replication group, adding new processes to the group, and maintaining consistent group membership information across all correct members of the group. The component subscribes to the *Failure_Detect* event generated by the Failure Detection component. The protocol guarantees that if the *Failure_Detect* event is generated by the Failure Detection component at any correct group member p_i urging the removal of another member p_j, then a new view installation that removes p_j from the

group membership will complete at all correct group members. All group membership protocol messages are sent to the group using the reliable multicast service component.

Reliable Multicast This service component implements a protocol that guarantees that all correct processes deliver the same set of multicast messages. The protocol also ensures that the contents of a particular multicast message (i.e., a particular sequence number) as delivered at all correct processes are the same. It thus prevents situations in which a malicious group member sends mutant messages, i.e., two messages with the same sequence number but different contents to two subsets of the replication group. For this purpose, the protocol uses operations provided by the Cryptography component for generating message digests and digital signatures. The protocol guarantees FIFO delivery for the messages multicast by a given sender, but does not guarantee any delivery order for messages multicast by different senders.

Total Ordering In the state-machine replication model, it is crucial that all replicas receive application-level messages multicast by different senders in the same order. The Total Ordering component ensures that if two correct replicas both deliver multicast messages m_1 and m_2, then they deliver them in the same order. All messages generated by this component are sent to the group using reliable multicast. At the time of a new view installation, the total ordering protocol partitions the set of all possible multicast message sequence numbers among the members of the replication group and assigns one partition to each replica. Each replica generates messages with monotonically increasing sequence numbers from its assigned partition without any gaps. The multicast messages are delivered in sequence number order. A faulty process can stall the progress of the protocol by not sending a message. We avoid that by forcing group members to transmit protocol-level null messages (i.e., no payload) if they do not have any other messages to send. All processes monitor the progress of the protocol. A *Suspect_Report* event is generated by the Total Ordering component at a correct process p_i for a process p_j that attempts to stall the protocol by not even sending null messages. The event is handled by the Failure Detection component. If the event is generated at more than two-third members of the group, then p_j will be eventually excluded from the group.

Gossip When a replica is created, its target group (the group that it should join) is specified. The Gossip component in the new replica periodically announces its existence by broadcasting a signed gossip message. Upon receiving this message, the Gossip functionality at replicas that are already members of the target group checks whether the sender is authorized to join the group, and if so notifies the Group Membership component, which initiates a agreement protocol that causes current members to update their membership lists to include the new replica. Once agreement has been reached, the members notify the new replica of its addition to the group. Upon receiving signed notifications from at least two-third members of the group, the new replica updates its membership, and becomes part of the target group.

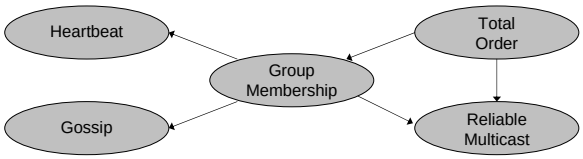

Figure 2. Dependency Graph for the Prototype GCS. An Edge from Component A to Component B Means that A Depends on B.

Heartbeat The Heartbeat component implements a simple heartbeat mechanism to detect crash failures. Periodically, the component multicasts (through unreliable/best-effort multicast) a heartbeat message with a logical timestamp. If the heartbeats from a particular replica have not been received after more than a threshold number of heartbeat periods have elapsed, the component generates a *Suspect_Report* event for that replica that is handled by the Failure Detection Component.

4.2. Dependency Graph for the CoBFIT GCS

Figure 2 gives the dependencies between the service components in the prototype CoBFIT GCS. The dependencies can be explained briefly as follows. The Group Membership component relies on the Reliable Multicast component to consistently deliver its messages across all correct replicas and to prevent malicious replicas from sending mutant messages. The Group Membership component relies on the Gossip component to hear from new replicas that want to join the group and on the Heartbeat component to detect replica crashes. The Total Ordering component requires the Group Membership component to remove corrupt members that refuse to send their assigned sequence numbers and thereby stall the progress of the total ordering protocol. The dependence of the Total Ordering component on the Reliable Multicast component is due to the requirement that any totally ordered application-level message multicast by one replica must be reliably delivered at all correct replicas.

4.3. Support Provided by the CoBFIT Framework

In addition to the dependencies among the service components, there is also a general dependency of any service component on the CoBFIT framework. For example, all ser-

vice components rely on the Cryptography component for digitally signing/verifying messages, on the primitives provided by the Secure Data Manager component for various message marshalling/demarshalling operations, on the Network component to communicate with peer service components on remote CoBFIT systems belonging to the same replication group, and on the Event Manager component for communication with other service components within the same CoBFIT system. The Group Membership component depends on the Failure Detection component to receive the *Failure_Detect* event, based on which it removes faulty replicas from the group.

The Failure Detection component handles *Suspect_Report* events generated by the service components. A service component at correct group member p_i generates a *Suspect_Report* event for another group member p_j if the peer service component at p_j has deviated from the service specifications. We place no restrictions on when a corrupted group member generates a *Suspect_Report* event. The *Suspect_Report* event carries the following information: (1) the suspected group member for which the event was generated, (2) the type of anomalous behavior exhibited by the suspected group member, and (3) (for certain kinds of faults) a justification that can be used to convince other group members that the suspected group member indeed exhibited the anomalous behavior. Justification is not possible for timing faults or message omissions in an asynchronous system model (e.g., when the total ordering protocol is stalled because neither application-level messages nor null messages have been received from a particular group member for a long time). However, justification is possible for certain types of Byzantine faults (e.g., when a corrupted group member tries to reliably multicast different messages with the same sequence number).

When the Failure Detection component at group member p_i handles a *Suspect_Report* event from a local service component for another group member p_j, the Failure Detection component at p_i generates a digitally signed *Suspect_Report* message that is sent to the Failure Detection components at other group members. The message carries all the information contained in the *Suspect_Report* event. The Failure Detection component at p_i generates the *Failure_Detect* event for a group member p_j if (1) a valid signed *Suspect_Report* message (or a local *Suspect_Report* event) with justification has been received for p_j, or (2) valid signed *Suspect_Report* message without justification have been received from more than two-third members of the group for p_j. At this point, we say that p_j has been *convicted* at p_i. The Failure Detection component at p_i also forwards the relevant *Suspect_Report* messages to other group members, so that eventually other group members will also convict p_j. This will lead to the Group Membership service components at the correct group members undergoing an agreement protocol and eventually removing p_j from the group.

We now describe the functionality provided by the Replication Manager component with an example. This functionality is currently being implemented in the CoBFIT GCS. Consider a distributed system with N available nodes. Suppose that for providing intrusion-tolerance, an application has to be replicated at a subset of the nodes in the distributed system using the state-machine replication approach. Also, suppose that the intrusion tolerance requirements for the application can vary at run-time. A CoBFIT system is manually instantiated at each of the N available nodes. The CoBFIT system will contain all the framework components and the service components of the CoBFIT GCS. However, at this point, the application component (that implements all the application-specific functionality) is not yet instantiated at any CoBFIT system. The CoBFIT system at each node is a process, and all the CoBFIT systems together form a process group. One way by which the user can specify the dependability requirements for the application is in terms of desired fault-resilience. For example, the user may specify that the state-machine replicated application must be resistant to t Byzantine faults. This specification is given by the user to the Replication Manager components at the N nodes. Assuming that $N \geq 3t + 1$, the Replication Manager components execute a distributed protocol in which they agree on the set S of $3t + 1$ nodes where the application replicas will be instantiated, based on factors such as the load conditions at various nodes. At a node $i \in S$, the Replication Manager component will instruct the Constructor component to instantiate the application component. The application components that have been instantiated at the $t + 1$ nodes will discover each other by using the Gossip service (provided by the Gossip service component described in Section 4.1), and will form a replication group. After some time, the user may desire to increase the fault-resilience to t' and will accordingly instruct the Replication Manager components at the N nodes. The Replication Manager components will again undergo an agreement on which additional nodes should instantiate the application component. Agreement will be reached provided that no more than $\lfloor \frac{N-1}{3} \rfloor$ nodes are corrupted. After agreement, the Replication Manager components at those CoBFIT systems already running the application components will instruct the application components to finish executing pending requests and reach a consistent state before allowing new replicas to join the replication group. Once a consistent state has been reached at all existing replicas, the Replication Manager components at the CoBFIT systems where new replicas are to be started are notified. This will result in the instantiation of the application component at those systems (by their respective Constructor components). The new replicas will get the updated application state from the

old replicas (through a state transfer protocol that would be implemented as a service component). The new replication group can then continue with normal application processing. It is important to note here that replica state consistency is not the responsibility of the Replication Manager. That has to be ensured through appropriate protocols implemented as service components (e.g., the Total Ordering and Reliable Multicast service components in the CoBFIT GCS).

5. Conclusion

This paper introduces CoBFIT, a component-based framework that provides specialized support for intrusion-tolerant services. The design and implementation principles of the CoBFIT framework stress characteristics that are essential for dependability in the face of attacks. These characteristics include portability, reconfigurability, flexibility, and adaptability. The framework components were designed using well-scrutinized software patterns with the goal of providing a framework for building intrusion-tolerant services that is itself robust. We built a prototype intrusion-tolerant group communication system using the CoBFIT framework and described the current implementation status. Future work includes (1) refining and extending the prototype implementation of the CoBFIT framework components, and (2) exploring additional supporting software mechanisms for intrusion tolerance that can be added to the CoBFIT framework.

Acknowledgments: We would like to thank the anonymous reviewers for their helpful comments, and Jenny Applequist for her editorial assistance.

References

[1] ACE FAQs. http://www.cs.wustl.edu/˜schmidt/ACE.FAQ.html.

[2] Cryptlib Toolkit. http://www.cs.auckland.nz/ pgut001/cryptlib/.

[3] The ITUA Project. http://itua.bbn.com.

[4] K. Birman, B. Constable, M. Hayden, J. Hickey, C. Kreitz, R. van Renesse, O. Rodeh, and W. Vogels. The Horus and Ensemble Projects: Accomplishments and Limitations. In *Proc. DARPA Information Survivability Conference and Exposition (DISCEX-2000)*, pages 149–161, 2000.

[5] H. Bruyninckx, B. Koninckx, and P. Soetens. A Software Framework for Advanced Motion Control. http://www.orocos.org/documents/motconframe_hc.pdf.

[6] C. Cachin and J. A. Poritz. Secure Intrusion-Tolerant Replication on the Internet. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN-2002)*, pages 167–176, 2002.

[7] M. Correia. *Intrusion Tolerance Based on Architectural Hybridization*. PhD thesis, University of Lisbon, 2003.

[8] Y. Deswarte, L. Blain, and J. C. Fabre. Intrusion Tolerance in Distributed Computing Systems. In *Proc. IEEE Symp. on Research in Security and Privacy*, pages 110–121, May 1991.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2001.

[10] R. Guerraoui and A. Schiper. The Generic Consensus Service. *IEEE Trans. on Software Engineering*, 27(1):29–41, Jan. 2001.

[11] M. Hiltunen. Configuration Management for Highly-customizable Services. In *Proc. 4th Intl. Conf. on Configurable Distributed Systems*, pages 197–205, May 1998.

[12] M. A. Hiltunen, R. D. Schlichting, and C. A. Ugarte. Building Survivable Services using Redundancy and Adaptation. *IEEE Trans. on Computers*, 52(2):181–194, February 2003.

[13] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proc. 31st Annual Hawaii Intl. Conf. on System Sciences (HICSS)*, volume 3, pages 317–326, January 1998.

[14] J. Koziol. *Intrusion Detection with Snort*. SAMS, May 2003.

[15] P. Luenam and P. Liu. The Design of an Adaptive Intrusion Tolerant Database System. *Foundations of Intrusion Tolerant Systems (OASIS 2003)*, pages 14–25, December 2003.

[16] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels. In *Proc. 21st Intl. Conf. on Distributed Computing Systems (ICDCS-21)*, pages 707–710, Phoenix, USA, April 2001.

[17] M. Pease, R. Shostak, and L. Lamport. Reaching Agreements in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[18] D. Powell. Group Communication. *Communications of the ACM*, 39(4):50–97, April 1996.

[19] H. V. Ramasamy, A. Agbaria, and W. H. Sanders. Semi-Passive Replication in the Presence of Byzantine Faults. Technical Report UILU-ENG-04-2202, University of Illinois, Urbana-Champaign, February 2004.

[20] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN-2002)*, pages 229–238, 2002.

[21] M. K. Reiter. The Rampart Toolkit for Building High-Integrity Services. *Theory and Practice in Distributed Systems (LNCS 938)*, pages 99–110, 1995.

[22] D. Schmidt and S. Huston. *C++ Network Programming: Mastering Complexity Using ACE and Patterns*. Addison-Wesley Longman, 2002.

[23] D. Schmidt and S. Huston. *C++ Network Programming: Systematic Reuse with ACE and Frameworks*. Addison-Wesley Longman, 2003.

[24] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley, 2001.

[25] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[26] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2001.