International Journal of Computer Architecture and Mobility
(ISSN 2319-9229) Volume 2-Issue 9, July 2014

Dependable Fault-Tolerant Based Software Architectures

Vasudevan Janarthanan
Department of Information Technology
Fairleigh Dickinson University
v_janart@fdu.edu

Abstract -- **Dependable software architectures determine the method of integrating a fault tolerance technique with a given system in order to make the system dependable. Dependable architectures demonstrably possess properties such as safety, security, and fault tolerance. Enriching software architecture descriptions by including dependability attributes will enable and facilitate the reuse of software components. This paper summarizes the different trends in the development of dependable software architecture along with their inherent limitations. The possible modifications to these existing architectures are also illustrated in order to improve sufficiently the dependability features of a software system.**

Keywords – **Software Architectures, Dependability, Fault Tolerance, Safety, Security.**

I. INTRODUCTION

As the size and complexity of software systems increase, the design and specification of the overall system structure become more significant than the choice of algorithms and data structures of computation. The various structural issues of the system then form the software architecture at the design level [14]. Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. The architecture of a software system defines that system in terms of computational components and interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system, thereby providing the rationale for the design decisions.

Since most of the modern computing systems require evolving software built from existing software components, developed by independent sources, the construction of systems with high dependability requirements out of software components represents a major challenge, since few assumptions can generally be made about the level of confidence of external components. Dependability is the property of a computing system that allows reliance to be justifiably placed on the service it delivers. When software products are deployed in a high-integrity system, their dependability profile is key to the survivability of the system. In this context, an architectural approach for fault tolerance is necessary in order to build dependable software systems assembled from untrustworthy components [9, 13]. Enriching software architecture descriptions by including dependability attributes will enable and facilitate the reuse of software components [12, 16]. Also, the structure of a dependable architecture makes clear how to compose a dependable system from a base system.

Rest of the paper is organized as follows: In section II, issues related to the development of dependable software architecture is briefly discussed. Section III presents various architectural limitations, acting as a catalyst for future research and motivation for newer solutions. In section IV, the current approaches in resolving some of the limitations is introduced. Also, an in-depth analysis on each of those approaches is provided with the rational for each one of those approaches.

II. ISSUES RELATED TO DEVELOPMENT OF DEPENDABLE SOFTWARE ARCHITECTURE

In order to improve the effectiveness of software fault tolerance some problems need to be addressed. Among them are the high costs (both the run-time overhead and design cost), the ability to evaluate the impact of software fault tolerance structures and the usually very limited flexibility of software fault tolerance designs and their consequent inability to adapt to changing run-time conditions. But the most formidable obstacle in realizing dependable software architectures is the need to demonstrate that a centralized architecture failure cannot bring about simultaneous loss of functions utilizing shared resources [16].

Fault tolerance, reliability, and availability are characteristics that are intimately interdependent. In order to have dependable software architecture, not a single component but the entire software system needs to be fault-tolerant [7, 8]. Existing architectural styles, such as client-server, may not represent fault-tolerant mechanisms that allow obtaining trustworthy components from untrustworthy ones. Instead, new forms for representing software systems are necessary if there is the need to deal with dependability related architectural mismatches, which might be associated with the necessity for obtaining dependable services from untrustworthy components [10].

One of the problems when building large-scale software systems out of existing software components are the architectural mismatches that might occur between system components [5]. An architectural mismatch occurs when the assumptions that a component makes about another component do not match. Mismatches occur when building

dependable system out of untrustworthy components, which is essentially an evolution problem since the system, its components, and their interactions have to change according to the required dependable needs.

From the description of fault tolerant software architecture, it is clear that the software properties are an integral part of the fault tolerance aspects of a system, but in literature the researchers have left undefined a number of parameters indispensable for the configuration, deployment and correct functioning of the corresponding fault tolerant mechanism [12]. Such parameters include the degree of replication for a given failure probability of the software, hardware constituents of a system, and specific load-patterns.

Architecture for a large, complex system, and even some simple systems, will involve multiple levels of detail expressed in multiple architectural styles. The problem of gaining confidence in the correctness of the implementation is especially acute in the case of dynamic, dependable architectures, where exhaustive testing of architectural configurations is frequently prohibitively expensive [16].

Software fault tolerance has been traditionally attacking the practical aspect of a dependable system, basically the conception, design and implementation of fault tolerant mechanisms that can be used to confront a variety of failure events [12]. So when the scale of the software architecture is small and simple, it is manageable to use the product of software fault tolerance to obtain the desired dependability properties for the architecture. But in the case of large-scale distributed systems (architectures), the direct integration of a fault tolerance mechanism to obtain the desired dependability guarantees is no longer attainable.

III. LIMITATIONS AND MOTIVATION

The software architecture community has made great strides towards characterizing and capturing system descriptions appropriately and towards providing linguistic support for defining families of software products, but current Architectural Description Languages (ADLs) and their associated methodologies (like SADL) do not adequately address dependability [16]. So, there is a need to look into the aspects of design of an ADL that would ultimately guarantee reliability of the software architecture.

Fault tolerance at the architectural level is an area that has recently gained considerable attention [8]. Most of existing works in this area emphasize the creation of fault tolerance mechanisms and description of software architectures with respect to their dependability properties [12, 16]. Providing means to support the systematic analysis of fault tolerance software properties, and the reasoning on the correctness of their integration within software architecture is a crucial thing to concentrate on.

While specifying fault tolerance properties and software security, current works have problems in systematically combining security constraints with fault tolerance properties [12, 15]. In specific cases, the priority between those two nonfunctional aspects can be deduced by the correctness of their combination in a given order. Hence there is a need to look more closely into the combinations of fault tolerance software properties with other nonfunctional and algorithmic software aspects.

Software architectures do not provide the means to facilitate analysis of the system's dependability requirements in order to identify the corresponding fault tolerant mechanism, and to integrate it with the system architecture [13]. These facts give rise to an emerging need to provide support for the correct system design, which integrates dependability considerations in the system architecture.

The development of dependable software architecture is not based on inventing the mechanism that provides the desired dependability guarantees; rather, it is based on selecting from the existing techniques the one that best meets the system's dependability requirements [1, 16]. Hence there is a need for a more formal method of arriving at the conclusion of a particular technique in order to establish the reliability features of software architecture.

IV. CURRENT APPROACHES AND THEIR RATIONALE

A. Designing dependable software architectures using architectural prescriptions from goal oriented requirements specification [3,4]:

A prescription allows the architect to reuse all the components and the topology that are derived from particular goals (requirements), including dependability requirements [3, 4]. Generally, a new system design has a higher likelihood of failure than a well tested one. Another way that an architectural prescription favors the design of dependable systems is by enabling the reuse of the high level design of systems that, having been already deployed and demonstrated to be dependable. An architectural prescription lays out the space for the system structure by selecting the architectural components (processes, data, and connectors), their relationships (interactions) and their constraints. KAOS [11] is normally used as a goal oriented requirements specification language and Architecture Prescription Language (APL) is used to derive an architectural prescription from the KAOS requirements.

In a prescription, the fundamental characterization of components is given by the goals they are responsible for. Components are further characterized by their type, processing, data or connector. The processing components are those that provide the transformation on the data components. The data components contain the information

to be used and transformed. The connector components, which may be either implemented by data components, processing components or a combination of both, are the glue that holds all the pieces of the system together. The interactions of the components among each other, together with the restriction of their possible number of instances characterize the topology of the system. At the beginning, some candidate components for the architecture are proposed, and then the functional and non-functional goals are assigned, one at a time, to a subset of the potential components. Those components who do not contribute to the achievement of any goal are discarded from the system.

KAOS is composed of: Objects, Agents, Entities, Events, Relationships, Operations, Goals, Requisites and Assumptions. The high-level goals are gathered from the users, domain experts and existing documentation. These goals are then AND/OR refined till the goals achievable by some agents are derived. For each goal the objects and operations associated with it are identified. Each entity in KAOS that refers to a subset of the system specification can be corresponded to an APL entity that describes the constraints on the software architecture. Each object in the requirements generally corresponds to a component in the architecture. More specifically, an agent object corresponds to either a process or a connector. The events relevant to the architecture of the system are those internal to the software system. An entity corresponds to a data element, which has a state that can be modified by active objects. A relation corresponds to another type of data element that links two or more other objects. A goal is a constraint on one or more of the components of a software system. This higher-level architecture specification (APL) can be easily translated into an architecture description, in the solution domain.

B. **Use of Object Oriented design and certain architectural properties for designing true fault tolerant software architecture [15]:**

The distinct architectural properties of Canadian Automated Air Traffic System (CAATS) are used to achieve fault tolerance in software architectures [15]. A distinct property of CAATS is the existence of several lightweight object oriented frameworks. CAATS approach to fault containment is based on an observation that large grain objects offer a natural boundary for fault containment. For containment to be successful, an action needs to be taken such that further deterioration of the system is avoided. In an Air Traffic Control, for example, Flight is the essential large grain object, and while evaluating a method on a Flight object, if a residual bug is encountered, the system should mark that particular Flight object as erroneous, warn the human operator about it, and continue successful support of all other Flights as if nothing had happened.

CAATS strategy for dealing with residual software bugs transforms latent defects into operator workload in a balance manner – the increase in workload is proportional to the severity of the fault. One of the frameworks in CAATS called Pivot provides environment for implementing societies of cooperating objects. Pivot is object oriented infrastructure element in between traditional object oriented programming systems and the object oriented application frameworks. It is an environment where a family of objects is submerged, providing a means to both create and destroy objects and a means for an object to exhibit its autonomous behaviour. In this way Pivot plays a crucial role in providing systematic and orderly means for fault containment.

Object behaviour in Pivot is supported by a "sense of time" wherein the infrastructure provides a means for an object to periodically execute a private method, "social responsibilities" wherein the infrastructure supports methods sensitive to events produced by other objects in the society, and the "asynchronous interaction" wherein the infrastructure encourages interaction between objects.

The fault containment by the object-oriented framework is as follows: First it traps the exception triggered by a residual fault, then isolates the faulty object and finally verifies whether the containment was indeed successful or not. The encapsulation concept of object-oriented design offers not only to detect the residual faults in a software system but also to contain them as well. Architectures, which include object-oriented framework elements, have the necessary properties for systematic and orderly degradation of the system and the resulting increase in operator workload.

C. **Providing software designers with a repository of dependable software architectures to help them find out requirements for the system under design to be dependable [13]:**

At first, for a given formal framework, various dependability properties are defined which would serve to characterize dependability behaviours of software architectures [13]. The important point is that based on this approach of specification of dependability properties, it enables one to characterize the various behaviors of a system in the presence of failure, which are attainable using existing fault tolerance techniques. The set of these behaviors may further be expanded as new fault tolerance techniques emerge. Dependability properties fall into two groups:

1) Abstract properties specified in terms of system states, which are defined independently of any fault tolerance technique. They serve to characterize the dependability behaviour of an overall architecture, when this behaviour is too abstract to associate a specific fault tolerance technique with it. Some of the abstract

properties are dependability, safety, availability and reliability.

2) Concrete properties specified in terms of system actions, whose definition is closely related to some fault tolerance technique. They serve to characterize the dependability behaviours associated to architectural elements, with respect to a given fault tolerance technique. More specific dependability properties are Detection and Fmask, where the former characterizes failure detection and the latter the system capability to mask the occurrence of failures.

These properties are then formally specified using the extended predicate logic with precedence actions. Based on the proposed approach of the specification of dependability properties, one can define a refinement relationship over these properties. This relationship allows refinement of an initial dependability requirement into more concrete dependability properties, which ultimately correspond to the behaviour of fault tolerance mechanisms for which an implementation is available.

The proposed specification of dependability properties provides means to unambiguously describe the dependability behaviour of architecture, albeit of limited help from the standpoint of easing the development of dependable systems. To facilitate their use, one can attach to each dependability property, the structure (*i.e.* the software architecture) of the corresponding system with respect to the fault tolerance technique that is used to enforce the given property.

The refinement relation over dependability properties provides the adequate base ground to organize the repository of dependable software architectures. The repository is organized as a lattice structure defined according to the refinement relation, and each node stores the acquired knowledge about a given dependability property. For some property *P*, this knowledge includes: (i) the property name, (ii) the formal specification of the dependability property, (iii) the set of dependability properties (through references to adequate nodes) into which *P* may be refined, (iv) the dependable software architecture *AP*, associated to *P*.

The description of dependable software architecture includes at least the specification of the dependability behaviour of its components, and may be extended using the capabilities of existing ADLs (Architecture Description Languages). Considering the proposed description of dependable architectures, a system *S* may be modified so as to enforce a given dependability property *P* by mapping *S* onto each generic component of the architecture associated to *P* while ensuring the declared dependability behaviour, and providing an adequate implementation for the dependability-specific components. Alternatively, the repository of dependable architectures may further be exploited to find out more refined architectures, which possibly correspond to available fault tolerance mechanisms.

To systematically infer a dependable architecture from a property specification, one can structure the specification of dependability properties accordingly. Let P be defined as:

$$P \text{ (objects } O_i,\ 1 \le i \le n;\ \text{var}) \equiv \text{objects: } O'_i,\ 1 \le i \le n';$$
$$\text{Behaviours: } O_i\text{: } B_i,\ 1 \le i \le m;$$

To infer the architecture associated to P(): it consists of defining the interpretation of each constituent of the property specification in terms of architectural description. The treatment of the objects and behaviours parts of the specification is direct: each object given in the objects list translates into an architectural component whose type (dependable) is the one declared in the embedding list; and each object's behaviour given in behaviours is attached to the corresponding architectural component.

D. Use of SADL architectural descriptions as hierarchies to provide a way to bridge the gap between abstract dependable software architectural models and their concrete implementations [16]:

It is very common to describe a single architecture, or related class of architectures, at multiple levels of abstraction and from a variety of perspectives [16]. Proving that an architectural property of interest holds at an abstract level is much easier than proving that it holds at a more concrete level. A property that is easy to prove from a data-oriented description may be difficult or impossible to prove from a function-oriented description at a similar level of abstraction.

Another motivation for having several architectural descriptions, rather than just one, is to help fill the conceptual gap between a very abstract description of the architecture and its fully concrete implementation. Often, it can be difficult to determine whether an abstract architectural description is accurately describing the implemented architecture. Unless the description is considered accurate, there is no reason to believe that properties of the description will be true of the system. More concrete architectural descriptions also provide more guidance to system implementers and maintainers. Hence there is a need for a new architecture description language, called SADL that represents collections of architectural descriptions as hierarchies, with each description linked to others by interpretation mappings that have been shown to guarantee consistency of the collection.

SADL architectural description hierarchies provide a way of bridging the gap between abstract architectural models and their concrete implementations. If a hierarchy is developed using only refinement patterns that have been proven to preserve certain dependability properties, then the

most concrete description in the hierarchy must have all the dependability properties of the more abstract descriptions. If it can be shown that some abstract description has a desirable dependability property, the most concrete description must also have that property. Since it is easy to verify that the most concrete description matches the actual implementation, because the most concrete description is based on the architectural constructs employed in the implementation, confidence that the property holds of the implementation is easy to obtain.

Hence, if an architectural description hierarchy is developed using verified refinement patterns, the proofs of the patterns guarantee that a certain class of properties is preserved. If it can be shown that one of the properties in that class holds at any level of the hierarchy, it must hold at every lower level, right down to the implementation level.

E. **Component based approach for architecting reliable software systems, where object oriented framework is utilized to construct the components [7, 8, 9, 10]:**

FaTC2 is an object-oriented framework, which facilitates the construction of fault-tolerant component-based systems by giving support to fault tolerance techniques [7, 8, 9, 10]. FaTC2 is an extension of C2.FW, an OO framework that provides an infrastructure for building applications using the C2 architectural style. The C2 architectural style is a component-based architectural style, which supports reuse and flexible system composition, emphasizing weak bindings between components. The C2 style is used due to its ability to compose heterogeneous off-the-shelf components.

FaTC2 introduces forward error recovery in the original framework by means of an exception handling system (EHS). An EHS offers control structures, which allow developers to define actions that should be executed when an error is detected. This materializes by the capability to signal exceptions and, in the code of the handler, to put the system back in a coherent state. A forward error recovery mechanism manipulates the state of a system in order to remove errors and enable it to resume execution without failing. Forward error recovery is usually implemented by means of exception handling.

In component-based development, source code for the components, which make up a system, might not be available, especially if third party components are employed. Hence, it is not possible to introduce exception handling directly in the component. An architectural level EHS deals with this kind of problem by providing an infrastructure for defining exceptions and attaching the corresponding handlers to components without the need to modify them.

In the C2 architectural style components communicate by exchanging asynchronous messages sent through connectors, which are responsible for the routing, filtering, and broadcast of messages. Components and connectors have a top interface and a bottom interface. Systems are composed in a layered style, where the top interface of a component may be connected to the bottom interface of a connector and its bottom interface may be connected to the top interface of another connector. Each side of a connector may be connected to any number of components or connectors. Two types of messages are defined by the C2 style: requests, which are sent upwards through architecture, and notifications, which are sent downward. Requests ask components in upper layers of the architecture for some service to be provided, while notifications signal a change in the internal state of a component.

The C2.FW framework provides an infrastructure for building C2 applications. The C2.FW Java framework comprises a set of classes and interfaces which implement the abstractions of the C2 style, such as components, connectors, messages, and interconnections. In order to facilitate the development of fault-tolerant applications using the C2 style, the Java version of C2.FW with the concept of Idealized C2 Component (iC2C). The original C2.FW framework does not provide adequate support for the construction of fault-tolerant systems.

FaTC2 allows fault-tolerant systems to be built in a well-organized manner, using iC2Cs as structural units. The main advantage of this approach is the fact that framework users do not need to implement an EHS in order to create fault-tolerant applications. Only the functional requirements and exception handling of the component should be defined. FaTC2 manages connections between functional requirements and exception handling.

F. **Diversity-based software architectures for security purposes [17 - 28]:**

Security is an important attribute of software dependability [17]. Indeed, a fault embedded in software represents vulnerability. This may end up being successfully exploited by an external interactive malicious fault (i.e. attack) and ultimately enable the violation of the system security property (i.e. security failure). Redundancy as the traditional means to achieve fault tolerance and higher system reliability is not effective against software faults. That is every copy of faulty software will have an identical behaviour when provided with the same input. This explains the potential and interest of the diversity principle for security purposes. The main idea is that through diversity common vulnerabilities can be decreased if not eliminated. As a result, it is very difficult for a malicious opponent to be able to break into a system composed of a set of diverse components and functionally equivalent with the very same attack.

As a consequence, diversity has naturally caught the attention of software security research community. Forrest et al. in [18] argue that uniformity represent a potential weakness because any flaw or vulnerability in an application is replicated throughout many machines. The security and the robustness of a system can be enhanced through the deliberate introduction of diversity. This work outlines how to introduce diversity using randomized compilation. In particular, they discuss a specific extension to the GNU GCC Compiler, which pads each stack frame by a random amount to defeat stack-based buffer overflow attacks. Deswarte et al. review in [19] the different levels of diversity of software and hardware systems and in [20] distinguish different dimensions and different degree of diversity. Bain et al. [21] presented a study to understand of diversity on the survivability of systems to a set of widespread computer attacks including the Morris worm, Melissa virus, LoveLetter worm. The authors of [22] report on a discussion held by a panel of renowned researchers about the use of diversity as a strategy for computer security and to identify the main open issues requiring further research. It emerges from this discussion that there is a lack of quantitative information on the cost associated with diversity based solutions and the lack of knowledge about the extent of protection provided by diversity.

Moreover, diversity has been used in software architectures targeting the monitoring of system behaviour. For instance, the HACQIT system [23] uses the status codes of the server replicas responses to detect failures. Totel et al. [24] extend this work to do a more detailed comparison of the replica responses and proposed intrusion detection algorithm with higher accuracy. These initiatives specifically target web servers and analyse only server responses. Consequently, they are not effective against a compromised replica that responds to client requests consistently. N-variant systems provide a framework which enables executing a set of automatically diversified variants using the same inputs [25]. The framework monitors the behaviour of the variants in order to detect divergences. The variants are built so that an anticipated type of exploit can succeed on only one variant. Therefore, such exploits can be rendered detectable. The building of variants requires a special compiler or a binary rewriter. Moreover, this framework detects only anticipated types of exploits, against which the replicas are diversified. Multi variant code execution is a runtime monitoring technique which prevents malicious code execution [26]. This technique uses diversity to protect against malicious code injection attack. This is achieved by running several slightly different variants of the same program in lockstep. The behaviour of the variants is compared at some synchronization points, which are in general system calls. The divergence in the behaviour is suggestive of anomaly and raises an alarm.

The behavioural distance approach [27, 28] aims at detecting sophisticated attacks which manage to emulate the original system behaviour including returning the correct service response. These attacks are thus able to defeat traditional anomaly-based IDS. This approach uses a comparison of the behaviours of two diverse processes running the same input. It measures the extent to which two processes behave differently.

CONCLUSION

In this paper I have summarized the different trends in the development of dependable software architecture along with their inherent limitations. I have also illustrated the various modifications to these existing architectures in order to effectively improve the dependability features of a software system.

REFERENCE

[1] Allen R. and Garlan D., “Formalizing Architectural Connection”, Proceedings of the Sixteenth International Conference on Software Engineering, Italy, May 1994.

[2] Astley M. and Agha G., “Modular Construction and Composition of Distributed Software Architectures”, Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE '98), 1998.

[3] Brandozzi M. and Perry E.D., “Architectural Prescriptions for Dependable Systems”, ICSE Workshop on Architecting Dependable Systems, 2002.

[4] Brandozzi M. and Perry E.D., “Transforming Goal Oriented Requirement Specifications into Architectural Prescriptions”, International Workshop on “From Software Requirements to Architectures (STRAW'01)”, Toronto, Canada, May 2001.

[5] Garlan D., Allen R. and Ockerbloom J., “Architectural Mismatch: Why Reuse Is So Hard”, IEEE Software 12(6), November 1995.

[6] Garlan D., Monroe R. and Wile D., “Acme: An Architecture Description Interchange Language”, Proceedings of CASCON’97, November 1997.

[7] Guerra P., Rubira C. and R. de Lemos, “An idealized fault tolerant architectural component”, Proceedings of the 24th International Conference on Software Engineering Workshop on Architecting Dependable Systems, May 2002.

[8] Guerra P., Rubira C. and Romanovsky R. and R. de Lemos, “Integrating COTS software components into dependable software architectures”, Proceedings of the 6th ISORC, IEEE Computer Society Press, 2003.

[9] Guerra P., Rubira C., Romanovsky R. and R. de Lemos, “Architecting Dependable Systems”, Lecture Notes in Computer Science, Springer-Verlag, 2003.

[10] Guerra P., Rubira C. and F. de Lima Filho, “FaTC2: An Object-Oriented Framework for Developing Fault-Tolerant Component-Based Systems”, ICSE Workshop on Software Architectures for Dependable Systems, May 2003.

[11] Lamsweerde A.V., “Goal-Oriented Requirements Analysis with KAOS”, www-dse.doc.ic.ac.uk/events/policy-99/ pdf/01-vanLamsweerde.pdf.

[12] Saridakis T. and Issarny V., "Fault-tolerant software architectures", Technical Report 3350, INRIA, February 1999.

[13] Saridakis T. and Issarny V., "Developing Dependable Systems Using Software Architecture", Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'99), February 1999

[14] Shaw M. and Garlan D., "Software Architecture: Perspectives on an Emerging Discipline", Prentice-Hall, 1996.

[15] Sotirovski D., "Towards Fault-tolerant Software Architectures", Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), August 2001.

[16] Stavridou V. and Riemenschneider R.A., "Provably dependable software architectures", Proceedings of the Third ACM SIGPLAN International Software Architecture Workshop, 1998.

[17] Avizienis, Algirdas, Laprie, Jean-Claude, Randell, Brian, and Landwehr, Carl E. , "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Trans. Dependable Sec. Comput., 1(1), 11–33, 2004.

[18] Forrest, Stephanie, Somayaji, Anil, and Ackley, David H., "Building Diverse Computer Systems", In Workshop on Hot Topics in Operating Systems, pp. 67–72, 1997.

[19] Deswarte, Y., Kanoun, K., and Laprie, J.-C., "Diversity against accidental and deliberate faults", In Ammann, P., Barnes, B. H., Jajodia, S., , and Sibley, E. H., (Eds.), Computer Security, Dependability, and Assurance: From Needs to Solutions, p. 171181.

[20] Obelheiro, Rafael R., Bessani, Alysson N., Lung, Lau C., and Correia, Miguel, "How Practical are Intrusion-Tolerant Distributed Systems?", (Technical Report TR0615) Departamento de Informatica Faculdade de Ciencias da Universidade de Lisboa, 2006.

[21] Bain, Charles, Faatz, Donald B., Fayad, Amgad, and Williams, Douglas E., "Diversity as a defense strategy in information systems. Does evidence from previous events support such an approach?", IICIS'01, Vol. 211 of IFIP Conference Proceedings, pp. 77–94, Kluwer, 2001.

[22] Deswarte, Y., Kanoun, K., and Laprie, J.-C., "Diversity against accidental and deliberate faults", In Ammann, P., Barnes, B. H., Jajodia, S., , and Sibley, E. H., (Eds.), Computer Security, Dependability, and Assurance: From Needs to Solutions, p. 171181, Williamsburg, VA, USA: IEEE Computer Press, 1998.

[23] Reynolds, James C., Just, James E., Lawson, Ed, Clough, Larry A., Maglich, Ryan, and Levitt, Karl N., "The Design and Implementation of an Intrusion Tolerant System", In International Conference on Dependable Systems and Networks (DSN 2002), pp. 285–292, IEEE Computer Society, 2002.

[24] Totel, Eric, Majorczyk, Fr´ed´eric, and M´e, Ludovic, "COTS Diversity Based Intrusion Detection and Application to Web Servers", In Valdes, Alfonso and Zamboni, Diego, (Eds.), Recent Advances in Intrusion Detection, 8th International Symposium, RAID'05, Vol. 3858 of Lecture Notes in Computer Science, pp. 43–62, Springer, 2006.

[25] Cox, Benjamin, Evans, David, Filipi, Adrian, Rowanhill, Jonathan, Hu, Wei, Davidson, Jack, Knight, John, Nguyen-Tuong, Anh, and Hiser, Jason, "N-variant systems: a secretless framework for security through diversity", In USENIX-SS'06: Proceedings of the15th conference on USENIX Security Symposium, Berkeley, CA, USA: USENIX Association, 2006.

[26] Weatherwax, Eric, Knight, John, and Nguyen-Tuong, Anh, "A Model of Secretless Security in N-Variant Systems", In Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS), In the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Network (DSN2009).

[27] Gao, Debin, Reiter, Michael K., and Song, Dawn Xiaodong, "Behavioral Distance for Intrusion Detection", In Valdes, Alfonso and Zamboni, Diego, (Eds.), Recent Advances in Intrusion Detection, 8th International Symposium, RAID'2005, Vol. 3858 of Lecture Notes in Computer Science, pp. 63–81, Springer, 2006.

[28] Gao, Debin, Reiter, Michael K., and Song, Dawn Xiaodong, "Behavioral Distance Measurement Using Hidden Markov Models", In Zamboni, Diego and Kr¨ugel, Christopher,(Eds.), Recent Advances in Intrusion Detection, 9th International Symposium, RAID'06, Vol. 4219 of Lecture Notes in Computer Science, pp. 19–40, Springer, 2006.