ChameleonSoft: A Moving Target Defense System

Mohamed Azab
Bradley Department of Electrical and Computer Engineering, Virginia Tech
Email: mazab@vt.edu

Riham Hassan
Computer Science, Virginia Tech
Email: rhabdel@vt.edu

Mohamed Eltoweissy[1]
Pacific Northwest National Laboratory
Email: mohamed.eltoweissy@pnnl.gov

***Abstract*— Ubiquitous cyber systems and their supporting infrastructure impact productivity and quality of life immensely. Their penetration in our daily life increases the need for their enhanced resilience and for means to secure and protect them. One major threat is the software monoculture. Latest research work illustrated the danger of software monoculture and introduced diversity to reduce the attack surface. In this paper, we propose a biologically-inspired defense system, ChameleonSoft, that employs multidimensional software diversity to, in effect, induce spatiotemporal software behavior encryption and a moving target defense. The key principles are decoupling functional roles and runtime role players; devising intrinsically-resilient composable online programmable building blocks; separating logic, state and physical resources; and employing functionally-equivalent, behaviorally-different code variants. Given, our construction, ChameleonSoft is also equipped with an autonomic failure recovery mechanism for enhanced resilience. Nodes employing ChameleonSoft autonomously and cooperatively change their recovery and encryption policy both proactively and reactively according to the continual change in context and environment. In order to test the applicability of the proposed approach, we present a prototype of the ChameleonSoft Behavior Encryption (CBE) and recovery mechanisms. Further, using analysis and simulation, we study the performance and security aspects of the proposed system. This study aims to evaluate the provisioned level of security by measuring the level of induced confusion and diffusion to quantify the strength of the CBE mechanism. Further, we compute the computational cost of security provisioning and enhancing system resilience. A brief attack scenario is also included to illustrate the complexity of attacking ChameleonSoft.**

Keywords— Cyber security, Ubiquitous computing, Software diversity, Online programmability, Biologically-inspired security.

I. INTRODUCTION

Biological inspiration in computer security dates, at least, to the definition of the term "computer virus" in the early 1980's [23]. Self-propagating malware and computer worms have clear life-like properties [24]. In contrast, currently used defenses predominantly lack biological flavor. In nature, diversity provides a defense against such self-propagating threats by maximizing the probability that some individuals will survive and replenish the population with a defense against that particular threat. It has been noted that much of the vulnerability of our networked computing systems can be attributed to the monoculture or lack of diversity in our software systems [1]. It is practically inevitable that software will contain flaws. Our software monoculture enables attack spread thus exposing the systems to large-scale attacks by well-informed attackers.

Inspired by the resilience of diverse biological systems in the sea chameleons, we propose a diversity-based defense mechanism against software attacks, termed ChameleonSoft. Sea chameleons or cephalopods employ multi-layer diversity for different purposes. For example, they leverage their capability to change their body color, texture and appearance to induce diversity. Diversity is used to camouflage for defense, disguise for hunting, and change color for communication [22]. Similarly, ChameleonSoft utilizes spatiotemporal software diversity to enhance software system security, survivability and resilience.

ChameleonSoft is based on our Cell-Oriented Architecture (COA). COA is a biologically inspired architecture with active components called cells that support the development, deployment, execution, maintenance, and evolution of software. Cells separate logic, state and physical resource management. Cells are dynamically composable into organisms that are bound to functional roles at runtime. Such construction supports online programmability, hot code swapping and automated recovery. These features together enable what we term as "ChameleonSoft Behavior Encryption (or CBE)" akin to message encryption.

CBE applies spatiotemporal diversity in a way that makes the attack target in continual random motion evading attackers. CBE leverages the COA intrinsic separation of concerns to realize temporal and spatial diversity. Temporal diversity is applied by shuffling multiple functionally-equivalent, behaviorally-different software variants at runtime. In addition, CBE realizes spatial diversity by enabling runtime seamless migration of cells from one physical host node to another. The goal behind that is to hide the potentially targeted software flaws that might be used to penetrate the system.

COA divides the large missions of a huge software program into smaller tasks. Each of these tasks is assigned to one or more cells in the form of manually or automatically generated sets of similar function and different-behavior executable variants. These sets might have different objectives targeting different quality attributes. Reliability, performance, robustness, and mobility are examples of such attributes. ChameleonSoft shuffles variants and sets to induce diversity. The scope of shuffling extends beyond security goals to the other quality attributes. The system might shuffle to a variant that aims at high system performance in overloaded but low security risk situations. Alternatively, the system would resort to a higher security, perhaps lower performance variant in higher risk situations.

Researchers in [4] mentioned that multi-variant systems without appropriate recovery mechanism might face a larger amount of coincident failures. ChameleonSoft is equipped with multimode recovery mechanisms providing different levels of fault tolerance granularity. Such feature increases the system resilience against international and unintentional failures.

[1] The author is also affiliated with the Bradlev Department of ECE niversity of Arizona

COLLABORATECOM 2011, October 15-18, Orlando, United States

DOI 10.4108/icst.collaboratecom.2011.247115

Inspired by the sea chameleon dynamic change in response to frequent changes in the environment; ChameleonSoft autonomously and seamlessly change the shuffling and recovery polices at runtime to suite the continual dynamic changes of the surroundings. This dynamic policy change enables ChameleonSoft to support legacy software packages that cannot be chameleon-ized (re-programmed to enable check-pointing needed for temporal diversity). ChameleonSoft will use only space diversity to encrypt the software behavior of such packages. In this case, ChamelonSoft will use only one failure recover mode to support this packages, the fine grain recovery. The details of software chamelionaization are beyond the scope of this paper. It is part of our future work to address this issue in our sequel papers.

Our main contributions in this paper can be outlined as follows:

1) A biologically inspired architecture as an employment of a mission-oriented application design and inline code distribution to enable adaptability, dynamic re-tasking, and re-programmability;
2) CBE mechanism that applies multidimensional spatiotemporal diversity to mobilize attack target;
3) A multimode, autonomous situation-aware recovery system for enhanced system resilience;
4) An elastic software platform that dynamically and autonomously change shuffling, and recovery policies to match the surroundings frequent changes.

In order to test the applicability of the proposed approach, we developed a prototype of the CBE mechanism. Further, using analysis and simulation, we studied the performance and security aspects of the proposed system. This study aims to evaluate the provisioned level of security by measuring the level of induced confusion and diffusion to quantify the strength of the CBE mechanism. We also estimated the computational cost of security provisioning, and enhancing system resilience in ChameleonSoft with regards to the amount of consumed resources, task completion time, and recovery downtime. We also illustrated by brief attack scenario the complexity of attacking ChameleonSoft.

The balance of this paper is as follows. Section 2 presents a brief literature survey. Section 3 describes our COA. Section 4 presents the moving target security mechanism. Section 5 discusses the evaluation of the proposed system. Finally the paper concludes in section 6.

II. RELATED WORK

Software diversity has a long history of research work in the field of software security and fault tolerance dated back to the 70's. Basically software diversity was presented as multiple independent solutions for the same problem.

The realization of that is to develop multiple independent versions of a program with different teams using different languages. The main goal from that approach was to increase the attacker confusion by changing the behavior of the software; which will make harder system exploitation. They expected that at any given time the majority of these versions will be working correctly [1, 2].

Some research work showed that there is a high probability that a multi-variant software approach might face many coincident failures [3, 4]. On the contrary other research work suggested that from the cost and the reliability point of view, the multi-variant approach is much better than the one "good" version, especially in mission critical applications where the cost of failure could be very high [20].

Diversity has been realized in various ways. Some work presented it in the form of confusion induction paradigm [5, 6]; where diversity is used to confuse the attack in order to complicate the attack process. A good example for leveraging diversity for confusion induction is presented in the form of a load-time binary transformation by [7]. Others presented different solution for diversity realization based on virtual machines called "private machine architecture" [8]. They used randomization to promote heterogeneity at the machine level aiming to increase the cost of broad-based binary attacks. Moreover, some commercial operating systems realized the ideas of operating system randomization [9, 10]. System call mappings, global library entry point, and stack placement randomization are used to induce diversity as mitigation for buffer overflow attacks.

Component diversity was investigated in Genesis [11], were the idea of providing both design diversity in the form of multiple variants representing different designs of the same specification as well as data diversity were proposed. Data diversity uses multiple copies of a single implementation operating on different data inputs but yielding the same desired results.

Massive-scale software diversity was presented by the help of automated variant generation and utilizing multicourse platforms. Compiler guided code variance approach aims to present such automation [12]. A realization of this massive-scale software diversity approach for the purpose of detecting anomalies by replicated execution was first presented by [13, 14, 15] they mixed diversity with parallelism and check pointing. They execute different variants of a program on a muticore environment while monitoring any deviation in the program flow to issue an intrusion alert.

A major drawback of existing solutions was the need for virtualizing every input to the whole set of executing variants at the same logical point to be able to detect the abnormal deviation of the execution flow. More advanced approaches with the objective of anomaly detection through detecting flow deviation but with much less constraints were presented in [16, 17, 18, 19].

These approaches generally apply different types of diversity mainly for reliability by replication or for intrusion detection by program flow deviation detection at runtime. Based on our knowledge utilizing runtime hot shuffling of software variants for behavior encryption was not previously investigated. Further, existing solutions used diversity to target specific quality attribute. Failure recovery mechanisms were not investigated as most of these solutions presented static diversity with low probability of failure. None of them investigated the idea of presenting a comprehensive solution that provides elastic, autonomous, resilient, situationally-aware platform targeting different quality attributes, while dynamically shuffling its software components to suit changes in the surroundings. Another drawback of these

solutions was the massive use of resources to realize diversity using virtualization techniques and multicore or multiprocessor platforms. ChameleonSoft is designed to support legacy systems with limited resources. It can dynamically tailor its tasks to suit the dynamic change in resource availability.

III. THE CELL ORIENTED ARCHETECTURE (COA)

In biology, sea chameleons, or chameleons for short, are well known for their capability to induce diversity. A chameleon colony may be perceived as a formation of a group of chameleon organisms playing some roles for missions covering different objectives. Each organism comprises a set of cells that cooperate to accomplish an organism mission. Each organism cell has a dedicated task that helps in the mission accomplishment. Some cells might have a directly related task to the organism mission while others might work to facilitate the success of other cells.

Our COA is inspired by the chameleon colony architecture The COA is an employment of a mission-oriented application design and inline code distribution to enable adaptability, dynamic re-tasking, and re-programmability. The *cell* is the basic building block in COA. It is the abstraction of a mission-oriented autonomous active resource. Generic cells termed stem cells, are seamlessly created by the middleware or the chameleon cell DNA (CCDNA). Further, they participate in emerging tasks through a process called specialization. The CCDNA is a middleware program that allows a physical workstation to host cells and facilitates cell physical resource allocation and management. Stem cells are free resources that abstract host node resources. They can encapsulate any of these resources to represent a part of an organism.

Once specialized, cells exhibit application specific behavior. Specialized cells have mission objective that are being continuously sought. The cell monitoring and analysis components are used to monitor performance parameters, mission objectives, and other phenomena of interest.

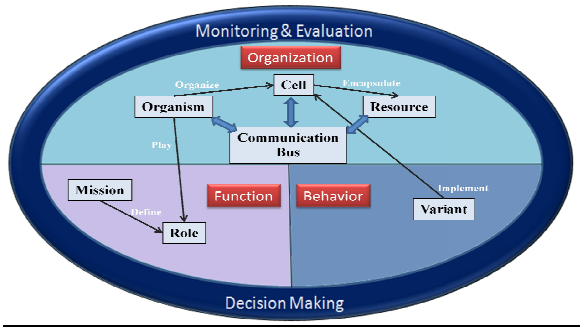

Figure 1. Components of our COA

We envision applications built over COA as a group of cooperating roles representing mission objectives. The term organism is used to represent a role player that performs a dedicated mission. An organism might be composed of a single or multiple cells based on its objectives. Fig. 1 illustrates the different components of the COA. The following subsections illustrate the design aspects of the COA architecture components namely the cell, the organism, and the management layer.

A. The Cell

Conceptually, the cell is the smallest active resource in a distributed computing platform. Cells are generic virtual computational units that acquire, on the fly, application specific functionality in the form of an executable code variant.

A single workstation can host one or more cells, providing a flexible way to share the physical resources among multiple applications. A cell can operate independently as a unicellular organism that possesses autonomous existence. It also could be part of a larger structure that resembles multi-cellular organisms. Fig. 2 illustrates the different components that contribute to construct the cell.

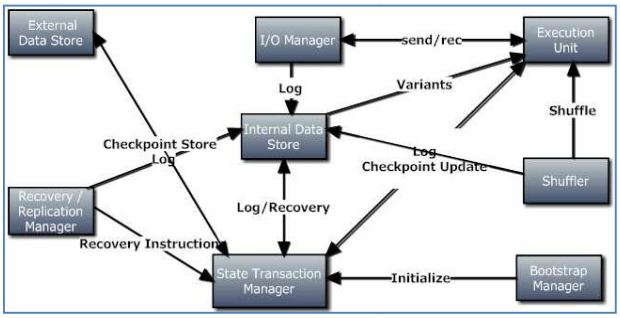

Figure 2. The Cell

Cells are instantiated at bootstrapping when the bootstrap manager initializes the cell components and ports with the appropriate parameters based on the bootstrap context. The I/O manager handles local and remote I/O communication setup, I/O logging, and IP/Port/Virtual naming resolution. The specialization process occurs when the execution unit receives the executable variant that represents the application specific functionality that the cell should acquire. Further, it is responsible for the runtime termination and replacement of the executing variants based on incoming shuffling commands. Shuffling commands is the responsibility of the shuffler unit; while the execution state preservation is the responsibility of the state transaction manager (STM). The shuffler applies a predetermined logic based on multiple feedback inputs from different sources supporting shuffling decisions. Most of these sources are representations of the situational awareness units within the cell, the organism and the Management Layer. The STM provides real-time monitoring and preservation of the executing program states and sensitive data. Further, it cooperates with the recovery and replication manager (RRM) to successfully restore the current state of an executing program in case of failure. The STM is responsible for storing the recovery data externally and internally in the data stores with the appropriate committing frequency for each store.

B. The Organism

An organism is an autonomous logical execution unit that follows the logic patterns of role providers. A role is an interpretation of a dedicated mission dynamically assigned to organisms. An organism might comprise a number of cells wired together dynamically (at runtime) to form a software structure having an independent execution context.

The simplest organism is composed of only a single cell. A more complex organism may span any number of cells that can be distributed among multiple physical computing hosts.

The organism is the underlying physical structure for the role functional element. Accordingly roles can transparently span multiple physical hosts through network-wide execution contexts. Exactly like any computing cloud, hosts with limited capabilities can collectively participate in the execution of complex autonomous roles.

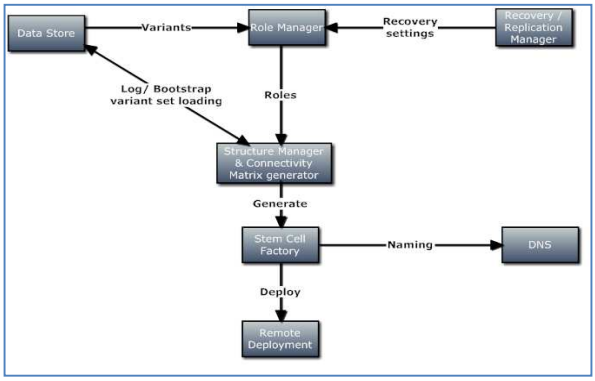

Figure 3. The Organism

Fig. 3 shows the different components that form an organism. The role manager is responsible for decomposing the organism designated role into a set of tasks in the form of executable variants to be assigned to the participating cells. It is also responsible for profiling the needed resources for each task to be executed. The stem cell factory is responsible for instantiating generic cells that will acquire functional variants to specialize. The structure manager and connectivity matrix generator generate the organism composition structure of cells. It is also responsible for drawing the connectivity diagram that guides the DNS (responsible for resolving the real host IP/Port mapping) to the virtual cell and organism names. The working cells use this mapping at runtime to direct incoming and outgoing communications. This is an intrinsic to the COA's separation of concerns that enables CBE space diversity features. In case of cell movement, the DNS will be instructed by the shuffler to maintain communication redirection; while the STM handles state, logic, and data maintainability. Finally the remote deployment unit will instruct the stem cell factory to deploy the generated cell on the selected remote host with the help of a remote deployment agent installed on that host as a part of the CCDNA.

C. *The Management Layer*

This layer is responsible for the organism creation, the overall platform management and the host side APIs. It has a major role in enhancing the cell situational awareness by utilizing a set of monitoring and analysis units. The Management Layer also monitors the working cells for recovery assessment. Further, it provides the necessary management tools for system administrators to manage, analyze, and evaluate the working organisms. Fig. 4 illustrates the main components of the management layer briefly described as follows.

Policy Manager: generates and assigns communication policies, and bootstrap recovery and shuffling policies. It is also involved in policy manipulation for management purposes.

Monitoring and Tracking: monitors executing organisms to facilitate the administration tasks, and recovery assessment.

Remote Deployment agent: host-based unit that generates a suitable environment for the cell to be executed over the host.

Mission Manager: converts missions into executable roles.

Organism Factory: generates organisms according to the mission interpreter output to play the designated role.

Data Store: a self-managed client server distributed database component. It is responsible for storing the logging and recovery data.

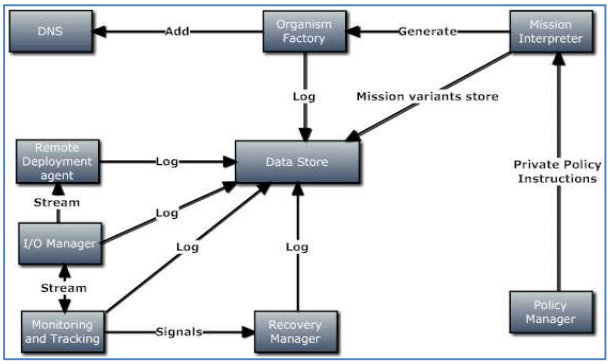

Figure 4. The management layer

IV. CHAMELEONSOFT MOVING TARGET DEFENSE APPROACH

We promote the novel moving target approach by ChameleonSoft as a defense mechanism against software attacks. Inspired by the chameleon diversity employment for camouflaging, ChameleonSoft encrypts software behavior by employing multidimensional diversity. The outcome is a continuous spatiotemporal change of the network behavior to, in effect, move the attack target away from the attacker. Our system is equipped with an autonomous, situational aware, multi-mode failure recovery mechanisms. Such recovery mechanisms enhance the system resilience against both intentional and un-intentional failures.

A. *Behavior encryption*

Typical encryption entails transforming the plain text into an unrecognizable message to the interceptor. Strong encryption schemes have two major properties namely confusion and diffusion. The confusion property virtually prohibits interceptors from predicting the ciphertext resulting from changing one character in the plaintext. An effective confusion is enforced via a complex functional relationship between the plaintext, key pair and the ciphertext. Confusion aims at maximizing the time that the attacker consumes to determine the relationship between the plaintext and the key pair. Diffusion is the other property of strong encryption schemes. Diffusion enables the cipher to spread the plaintext information over the entire ciphertext so that the changes in the plaintext affect many parts of the ciphertext [21].

Behavior encryption in ChameleonSoft is analogous to typical encryption in the way it exhibits the confusion and diffusion properties. ChameleonSoft induces confusion by dynamically changing the behavior of the executing software variant using runtime shuffling of code variants. The dynamic software behavior change makes it more difficult for an attacker to generate a profile with the possible flaws of the executing variant. The shuffling pattern is a supervised

reflection for the continuous change in the environs. In ChameleonSoft, an effective confusion is determined by how complex to correlate the change in the output behavior relative to a single induced change in the environment.

ChameleonSoft induces diffusion by generating a random virtually intractable significant change in the spatiotemporal network behavior using the cell independent decision making capability. Each cell in the network takes its own shuffling decision regarding when to shuffle the current variant, the shuffling frequency, and the variant selection for the next shuffle. These decisions are guided by a continuous feedback from the situational awareness units that monitor the cell surroundings and the shuffling policy.

For example, an attacker might be able to induce a change in the system surroundings "like overloading the network" to force the system to shuffle the currently executing variant. The cells close to the induced event change their variant set to target a different quality attribute (e.g. performance) that suits the induced change in the environment. Further, an alert is announced based on a predetermined announcement policy to other remote cells to inform them about that event. Based on that announcement, these cells make independent shuffling decisions regarding their currently executing variants. Those who decide to shuffle shall replace the current variant by another one from the same set to preserve their previously targeted quality attribute. These independent decisions make the attack target "a flaw in a specific variant" in continual random motion evading attacks.

We propose a variant layout randomization technique to increase the level of CBE's confusion induction. The system assigns the variant shuffling index based on a predetermined sequence. Variants' indices are shuffled internally within each cell based on a cell independently generated random number that changes over time. This random number is used to shift the next executing variant selection index to a random location in the variant layout space.

Software behavior encryption by runtime hot shuffling of software variants is a realization of ChameleonSoft temporal diversity. ChameleonSoft realizes space diversity by seamlessly moving the cell at runtime among different physical hosts. During this process, the COA autonomously maintains communications, cell sensitive data, and state logic.

ChameleonSoft can follow different shuffling policies at runtime to suit the dynamic change in the surrounding environment. A policy change might induce a change in the shuffling frequency for more security, or the shuffling orientation to favor time over space diversity or vice versa. ChameleonSoft can use space diversity only mode to encrypt the software behavior of legacy software packages that cannot be chameleon-ized (by enabling check-pointing). In this case ChameleonSoft will deploy multiple remote replicas for the cells executing such legacy packages. All replicas will receive same inputs, communication redirection between cells output ports will be used to achieve space diversity among these replicas. Software chameleonization is beyond the focus of this paper, we intended to present the details of this process in our sequel papers.

The overall diversity induced by our system can be expressed in the form of X missions represented in Y roles. These roles are played by M organisms, composed of K cells. Each cell has P quality attribute sets containing Z software variants, to be executed on Q nodes all over the network with average of R shuffling events/sec.

Our current work focuses on the behavior encryption through hot shuffling of software variants. We anticipate employing the variant generation process of [17] for an automated variant generation. Also functional programming tools are helpful in manually generating these variants.

B. ChameleonSoft multi-mode failure recovery mechanism

Chameleons employ different diversity techniques to increase the resilience of their camouflaging process against attacker visual observation. Changing body color, texture, and appearance are examples for such techniques. They recover from a technique failure by switching to another technique. Similarly, ChameleonSoft applies different diversity techniques for camouflaging to enhance the system resilience against attacker utilization of possible software flaws. Applying diversity might involve multiple interruptions of the executing variants. Doing so might lead to multiple coincident failures. Therefore, we designed an autonomous, dynamic, situational aware, multi mode failure recovery mechanism to resolve possible coincident failures. A major outcome of this recovery mechanism is the failure resilience enhancement not only against coincidental failures, but also against malicious induced failures by adversaries.

ChameleonSoft can dynamically and autonomously change the cell recovery policy to cover different fault tolerance granularity levels. Such levels might target reliability, survivability and resource usage optimization. For fine grained recovery against logical failures, a cell can have one or more replicas on the same physical host. Further, for more fine grained recovery against logical or physical node failure, a cell might have one or more replicas on different physical hosts. In ChameleonSoft replicas need to only replicate the STM and the data store units of the cell. The remaining cell components stay in hibernation waiting for resurrection when the replicas take over. ChameleonSoft does that to minimize the resource usage by these replicas.

In a resource constrained environment, ChameleonSoft can follow a more coarse grained recovery that might save some of the resources used by replicas while compromising some of the execution states. The cell is designed to send a periodic behavior change beacon messages containing its sensitive data, the currently executing set, variant, and the last executed state to be saved on a secure remote data store. These messages are mainly used to server communication and recovery objectives; while they might have other uses as illustrated in subsection C. In case of failure ChameleonSoft retrieves the last stored message for the failed cell. It leverages the message content and any available communication logs to restore the failed cell to its prior state before failure.

The coarse-grained recovery mode is always on by default enabling the support of multiple concurrent recovery policies. The remote safe store is updated regularly with beacon messages from all working cells. Each cell will independently and dynamically set its own message update frequency. Such update frequency could be influenced by the change of the current recovery policy. The update frequency might decrease in fine grained recovery mode; while they should increase with lower granularity recovery.

ChameleonSoft can change cell recovery policy at runtime to respond to changes in the surrounding environment. For example, in a resource constrained situation, the system might choose the coarse grained mode until more resources are available at which time the system could go for the finer grained mode.

Fine grain recovery by replication is the most suitable failure recovery option for cells executing legacy software packages as it can work with or without check pointing. In this case the whole cell will be replicated at bootstrap time ChameleonSoft will connect all replicas to the same input channel to guaranty correct synchronization. A replica takes over when it controls the output of the cell.

C. Decision making in ChameleonSoft

In chameleons, color shuffling decision making source and location depends mainly on the targeted changing speed. In fast changing chameleons, shuffling decisions are mostly controlled by the brain with dedicated connections "nerves", or through distributed decision making cells all over the body. In ChameleonSoft, we favor the later approach as it is more realizable and computationally cost effective from the communication and resource consumption point of views. The decision making unit in ChameleonSoft is an intrinsic cell component enabling independent decision making. More complex decisions affecting a group of cells or organisms are handled by distributed decision making units. These units are responsible for directing the network behavior change for global purposes. The decision making unit depends mainly on the situational awareness unit to guide its decisions. The details of these units are described in the following subsections.

1) Sensing and situational awareness:

In chameleons, color change is used for exchanging messages between colony members. Chameleons send commands, alerts, and guidelines through changing their body texture or color. In ChameleonSoft the behavior change beacons are akin to the chameleon color change messages. We use these beacons to send commands, alerts, and guidelines to other cells or organisms in the system.

Automated management and analysis units use these stored beacons to generate more meaningful status reports. These reports contain information, directions, and commands that the management want to deliver to a certain area in the network. The reports are classified according to the geographical area that they target. Each cell checks for new reports targeting its area while updating its own beacons on regular basis. Such reporting mechanism extends the situational awareness limits of each cell for more accurate decision-making. Specifically, the management to guide performance boosting, attack resolution, or attack containment in a certain area can use the reports. The details of these usages are not the focus of this paper. We only focus on using these reports for diffusion induction and recovery policy change direction.

Local situation awareness is achieved by the use of a group of sensors in the form of API's. These sensors are frequently used between cells and the CCDNA hosting them to sense any phenomena of interest. The sensors' feedback with the regular global report feeds are the main source of information supporting shuffling and recovery policy change to be discussed in the next subsection.

2) Shuffling and recovery dynamic policy change:

Shuffling decisions can be classified into two main categories. The first is shuffling the currently executing set to suit a local change in the environment. These changes might include performance overload or security issues. The second is to randomly shuffle the currently executing variant for behavior encryption. Such shuffling could be based on a randomly adjusted timer in each cell for confusion induction. It could also be for diffusion induction where cells randomly shuffle the current variant to diffuse the change in the network behavior in response to an induced change in the surrounding environment.

Shuffling decisions for diffusion induction are guided by the regular report feeds targeting the cell's area. A set shuffling announcement in a message targeting a specific network area means that each cell in this area is encouraged to shuffle its current variant for diffusion induction. The management attaches these announcements to the next generated report when it detects a variant set change in any part of the network. Each cell takes its own decision whether to shuffle its variant or not based on the available resources, current workload, and the allowable downtime.

ChameleonSoft may dynamically change the cell recovery policy at runtime. The change is guided by the application requirements and host conditions. In a stable situation with non-mission critical application, a coarse-grained recovery policy can be used, while in a more hazardous situation, a fine-grained recovery is preferred. The cell utilizes the available information about the current working environment with the application profile to decide the appropriate recovery policy to use. As the surroundings change, the cell changes the current recovery policy to suit these changes.

D. Attacking ChameleonSoft

A resourceful software attacker might use multiple sophisticated tools to target our system. She might use scanning tools searching for specific flaws in the executing variants that would mostly exist in a performance oriented variant. These tools direct the attacker when and where to start utilizing these flaws to attack the cell. The whole attack would only succeed if the attacker manages to do all of the above within the execution time of the variant containing the targeted flaw. ChameleonSoft situational awareness unit would be searching for such scanning and penetration attempts. Any sign of such actions necessitates a variant set change to a security oriented set or even an increase in the variant shuffling frequency in the current set.

Even if the attack succeeds, it can only cause a variant crash. This event simply calls for recovery and the variant is replaced autonomously by another variant. The state is probably restored to the last execution checkpoint before crash.

An adversary that has physical access to the host machines might launch a more drastic attack targeting the CCDNA program to crash all the executing cells on this machine. The fact that the CCDNA program should have a static behavior makes it vulnerable to attacks even if it was built with secure tools. The COA is designed to increase the level of granularity of mission execution by fractionizing the missions into a set of interconnected parts distributed over different hosts. Doing so increases the system resilience against massive failures. ChameleonSoft defense missions

are divided over multiple cells that might be hosted over different hosts. Crashing one host does not kill the whole application. Further, these cells might have replicas on other hosts that will automatically take over upon failure. The worst case scenario is that these cells might have no replicas. In this case, the management layer will sense the failure either by detecting a discontinuation of the beacon message postings from these cells, or by a notification from other cells that where in communication with the failed cells. The management layer autonomously replaces these failed cells and use communication logs and stored check points to restore the last execution state before failure. In a more targeted attack where the attacker attempt to fail a specific cell in the process of disabling the execution of a certain mission. ChameleonSoft Space shuffling makes it even harder for the attacker to determine the exact physical location of the targeted cell to attack. The attacker must tailor her tools to launch his attack on a moving cell. In doing so he must have access to all other physical hosts that this call might move to.

V. CHAMELEONSOFT EVALUATION

We use simulation to evaluate the security and performance of ChameleonSoft. Further, we developed a prototype with multi mode recovery policy as a step towards realizing the proposed system.

A. Security analysis

We simulated the behavior encryption module using Matlab to assess the provisioned level of security. We mainly measure the level of induced confusion and diffusion to quantify the strength of ChameleonSoft behavior encryption mechanism. Table I shows the parameters used in the simulation. The network parameters are mainly static parameters used to setup the experiments. The shuffling event parameters represent the spatiotemporal distribution of shuffling commands to induce confusion while the attack or change in environment parameters show the spatiotemporal distribution of attack events and the event type that necessities variant set change to respond to the change. Events shuffling variants selection parameters represent the selection criteria of the next variant to be shuffled while the independent shuffling decision on each cell parameter represents when the cell should take shuffling decision for diffusion induction.

TABLE I. SIMULATION PARAMETERS

Classification	Parameter		P_Type	Run1	Run2	Run3
Network	Network size		Static	10*10	10*10	10*10
	# shuffling variants in each set		Static	8	8	8
	# shuffling sets		Static	5	5	5
	Exp_Time		Static	15	15	15
Event	Normal Shuffling event	Timing	Poisson	20	18	16
		Location	Normal	10,2	8,3	6,5
	Attack or change in environment event	Timing	Poisson	21	20	21
		Location	Normal	11,3	9,4	10,2
		Type	Uniform	10	10	10
Software	Shuffling Variants Selection		Uniform	10	10	10
Shuffling	Independent shuffling decision on each cell		Uniform	10	10	10

1) Simulator Design:

We devised a cell representation to simulate the COA behavior encryption module. Our simulator starts by deploying cells all over the network based on the input parameters. Each cell should have a representation for a group of software variant sets for each possible induced change in the network. Each of these sets contains a group of similar functionally different behavior variants. After automatically deploying these cells, our attack event generator produces different events following the user predetermined settings.

The variant shuffling at each cell works seamlessly for confusion induction. The set shuffling occurs only in response to an induced change in a specific network location. Further, independent variant shuffling decision is taken at random locations to increase the level of behavior change diffusion all over the network.

2) Simulation Results:

We examined the behavior encryption module through three experiments with different simulation parameter values. The experiments aimed to measure the effect of changing attack arrival rate and location with the change of shuffling event generation on the behavior output as illustrated in Fig. 5 and 6. The effect of continuous variant shuffling with our diffusion induction mechanism on the output behavior was obvious. A simple change in any of those inputs leads to a significant change in the output. Our primary goal in this study is to illustrate the effect of our behavior encryption on the network behavior after attack events. This study focuses only on the security analysis of the system by showing the level of induced confusion and diffusion. Performance analysis will be discussed in the next subsection.

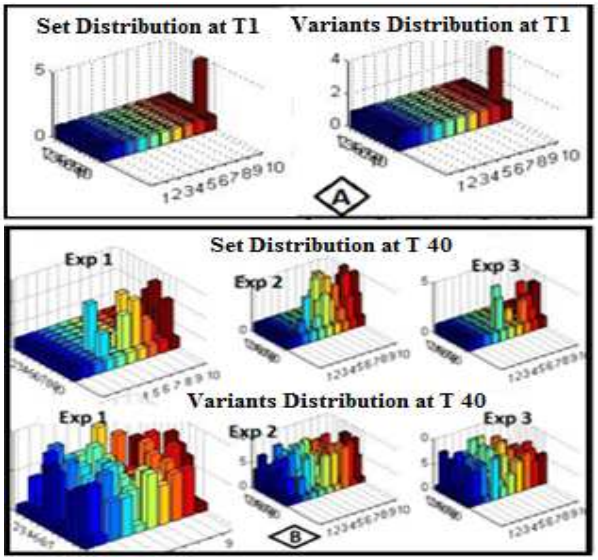

Figure 5. Induced Confusion and Diffusion

Fig. 5A gives a snapshot on the set and variant distribution over the cells at the bootstrap. Each column represents a cell in the network where the value represents the current executing set index or variant index. In Fig. 5B we illustrate the behavior output after short period of continuous behavioral encryption for the three experiments. We notice that the behavior changes are diffused all over the network. This can be seen by the massive change in the behavior of the whole network by the end of the experiment.

We plotted the number of induced changes in the network cells over time as shown in Fig. 6A reflecting the level of induced confusion at each timer tick. The reason for that is to track the continuous change in the overall network behavior through the whole experiment. Fig. 6B illustrates the accumulating change in the network behavior over time reflecting the effect of re-encryption and the increase in complexity of correlating the input to the output over time.

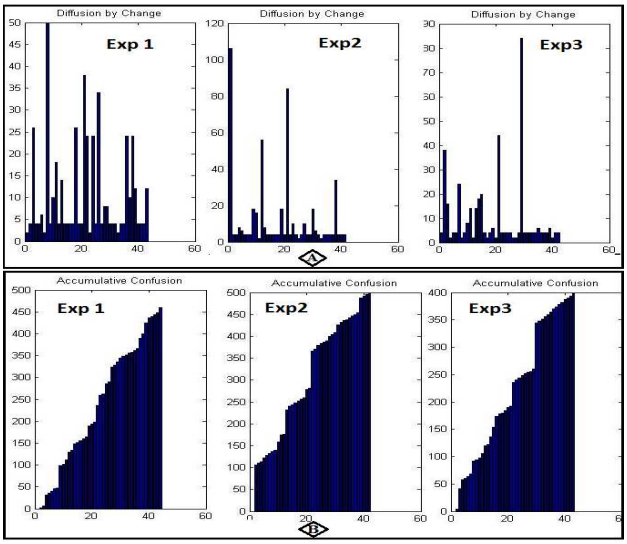

Figure 6. Induced Confusion and Diffusion

B. *Performance Analysis*

In this section, we scrutinize an analytical study on the performance aspects of ChameleonSoft. The goal of this study is to estimate the computational cost of security provisioning, and enhancing system resilience using the amount of consumed resources, task completion time, and recovery downtime measures. Table II shows the parameters that we used for this study.

TABLE II. PARAMETERS USED FOR THIS STUDY

Parameter	Symbol	Assumed Value
Shuffling decisions	F	10 ,20,30
Time to load a variant	T	0.01
Time to instantiate a cell	C	0.02
Average time to process communication logs	L	0.09
Average time for execution	E	5
Processing time	P	0.02
DNS Processing time	D	0.01
Space shuffling decisions	S	5
Frequency of failure	U	10,20,30

We define equations (1,2 and 3) to estimate the total down time of the cell for time T with shuffling frequency F towards evaluating the effect of CBE on the task completion time. Further, we define equations (4, and 5) to evaluate the effect of ChameleonSoft multi-mode recovery mechanism from the system stability, resource consumption and recovery downtime perspectives.

ChameleonSoft apply temporal and space diversity to induce the needed confusion to encrypt the software behavior. These employments have a clear impact on the overall task completion time. The following equations aims to express that effect in case of applying only temporal diversity, spatiotemporal diversity and in case of a static environment with no diversity.

$$P + C + E + T \tag{1}$$

$$\mathrm{C} + \mathrm{E} + (\mathrm{T} + \mathrm{P}) \times (\mathrm{F} + 1) \tag{2}$$

$$(\mathrm{C} + \mathrm{E} + (\mathrm{T} + \mathrm{P}) \times (\mathrm{F} + 1)) + (\mathrm{D} \times \mathrm{S}) \tag{3}$$

Equation (1) estimates the execution time of a program executing in a static behavior software environment without any appliance for diversity. Equation (2) aims to estimate the effect of applying only temporal diversity over the execution time of the program. The main difference between (1) and (2) is the added values reflecting the time needed to load the variant after each shuffle with some processing time consumed through this process. Usually these values are small compared to the overall execution time of the program. Further, the COA divides large missions into smaller tasks to be executed over the cells. The execution overlap of the independent tasks might even lower the overall execution time of the mission. Further, with the cell independent decision making at each cell can set its own shuffling frequency to satisfy the overall application requirement.

The spatiotemporal diversity appliance effect on the execution time is estimated in (3). The main difference between (2) and (3) is the effect of the space diversity. In space diversity a cell migrates from one physical host to another. The management layer instantiate a new cell in the new physical location to replicate the migrating cell. This cell will be in full time synchronization with the main cell exactly as a replica. The execution process will be interrupted only for the duration of redirecting the communication to the new location. The time needed to terminate the old cell does not affect the execution time of the task as it will occur after migration and operation restoration. We used (1, 2 and 3) with the parameters in Table II to draw Fig. 7 in order to visualize the effect of shuffling frequency change over the task completion time.

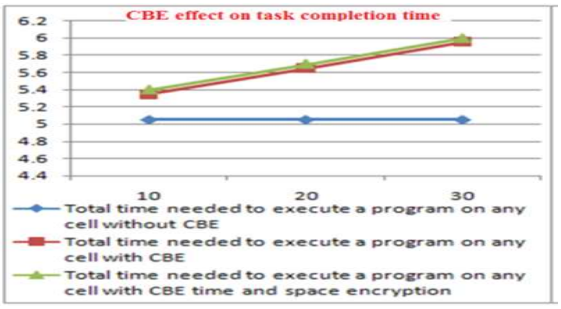

Figure 7. shuffling decision frequency effect on the task completion time.

Fig. 7 illustrates that increasing the level of induced confusion by increasing the frequency of shuffling towards enhancing the level of provisioned security linearly affects the task completion time. Similarly, complicating the correlation between the input and output network behavior by employing both the temporal and space diversity add more time to the overall task completion time. As mentioned

before ChameleonSoft is capable of changing its diversity appliance technique at runtime to suit the surrounding environment change and the application dynamic requirements. The reason behind enabling such feature is to provide some guarantees that the system shall always consider using the right resources at the right time towards balancing the security and performance output of the system.

ChameleonSoft employ different recovery mechanism with different levels of granularity to suit the dynamic change in the surroundings. Fine grained recovery by Cell replication might consume more resources in order to guarantee short recovery downtime and successful restoration of all the previous states before failure. As mentioned before ChameleonSoft optimize the replication resource usage by replicating only the STM and data store components of the cell. The remaining components of the cell remain in hibernation waiting for resurrection when the replica takes over. Equation (4) aims to estimate the total recovery time of a failed cell in case of fine grained recovery by replication. The overall recover time depends on the processing time that mainly represents the time needed to resurrect the hibernated replica components with the time spent to detect the failure.

ChameleonSoft usually uses coarse grained recovery mode in resource constrained environments to save the resources used by the replicated cell components. Restoring a failed cell with no replica might involve remote data store queries, collecting communication logs from other cells, and analyzing these logs for unsaved lost states. This process increases the overall recovery time without any guarantee for a successful restoration for all states before failure. Equation (5) estimates the overall time needed to recover a filed cell that has no replicas. The main difference between (5) and (4) is that in (5) we added the time consumed in instantiating a new cell to replace the failed cell and the time needed to reload the variant. This process is eliminated in case of replication as the replica instantiation and the variant loading occurs in parallel with the main cell instantiation and variant loading. Further, we added the time needed to process communication logs in order to restore lost states; which might be a significant amount of time. We used (4) and (5) with the parameters in Table II to draw Fig. 8 in order to visualize the effect of the recovery policy on the recovery downtime with respect to the change in the frequency of failure.

$$\mathrm{P} \times \mathrm{U} \tag{4}$$

$$(C + T + P + L) \times U \tag{5}$$

Fig. 8 shows the effect of replication on the failure recovery downtime. The figure shows that the fine grained recovery decreases the recovery downtime to a great extent compared with the coarse grained recovery especially when the cell faces large failure incidents. The main reason behind that is the elimination of the time spent in re-instantiating the cell and processing the communication logs.

ChameleonSoft might leverage its capability to change the shuffling and recovery policy at runtime to optimize the resource usage, the recovery downtime, and the level of provisioned security. The shuffling frequency can be lowered with a coarse grained recovery mode in secure stable situations. In more hazardous conditions, this value is increased and the recovery mode can be more fine grained to enhance system security and robustness.

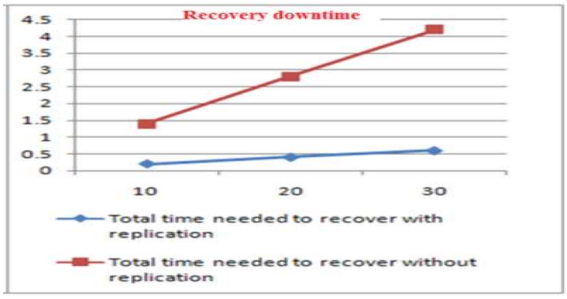

Figure 8. The replication effect on the recovery downtime

C. *Hot-shuffling prototype*

We developed a prototype of the CBE based on our COA. The prototype realized an organism composed of multiple cells deployed on different physical hosts. The prototype encrypts the executing module behavior by continuous hot-shuffling at runtime. We managed to implement hot-shuffling on multiple communicating cells that apply different failure recovery policies with different granularity levels.

Assumptions: We assume that the variant designer will include a checkpoint at each transitional stage of the program. This checkpoint will be sent through a dedicated channel to the state transaction manager each time the program enters a new checkpoint. At bootstrapping, the program must ask the state transaction manager about the last known checkpoint and begins the execution process at that point. The system can also work with an automated variant compiler like the one presented in [17], with simple modifications to realize our assumption in the output variants. We also assume that a copy of all the volatile memory content being used by the variant will be attached to each message sent to the STM. Doing so will guarantee data restoration validity at each shuffle because the checkpoints will always be synchronized with the data in memory.

1) *The Shuffling process:*

At bootstrapping, the executing variant retrieves the last checkpoint through the state transaction manager that will send zero if the variant is starting fresh. The executing variant keeps sending checkpoint updates throughout its execution lifetime.

At a shuffling event, the execution unit will immediately terminate the executing program and launch the new variant. The new variant retrieves the last known checkpoint through the state transaction manger to pursue the task of the previous variant. Before variant execution, the STM starts memory restoration process to synchronize the volatile memory content with the execution checkpoint.

We employed the regular checkpoint update solution versus one time update upon termination of the current executing variant. Our rationale for such design decision is to enforce autonomous resilience to failure since the termination event might not occur due to some malfunction. In such a case, the state transition manager will not receive the most recent checkpoint update.

Space diversity was not in the scope of this prototype version, but we intend to realize that in the next versions. The realization will be as follows, after a space shuffling decision, the shuffler will instruct the remote deployment unit to deploy a new cell with the same profile t a remote

location. The new cell will have the same logic. The STM unit of both the new and the old cells will communicate with each other to maintain states and sensitive cell data. Upon shuffling, the DNS will have to maintain communication profiles based on the new cell location. At that point, the old cell will be terminated and the new one will take over. The old cell will be working through the whole shuffling process. The cell downtime will only include the time needed for the DNS to maintain communications, which is a simple update in a database record.

2) The auto recovery mechanisms:

We developed hot and cold recovery schemes. The hot recovery is based on replicas where a cell is replicated at bootstrapping. The replica monitors checkpoint updates in an attempt to detect a failure event in which case the whole cell is killed and the replica takes over. The cell resumes its work automatically from the last known checkpoint and a new replica for it is generated.

In the cold recovery mode, the state transaction manager is instructed at bootstrapping to send the checkpoint update to an external data store with a predetermined frequency. The frequency of checkpoint update is a tradeoff between the recovery time and accuracy of restoring the last checkpoint and the network load. In this case, the management layer monitors the cell failure and instantiates a new cell with a recovery signal. When the newly instantiated cells detects this signal, it retrieves the last stored checkpoint to resume work. Communication logs between the cell and the surrounding cells are used for fine grained recovery of any lost execution steps.

VI. CONCLUSION

In this paper we presented ChameleonSoft as a moving target defense mechanism against software attacks. The system is built over our novel cell oriented architecture. ChameleonSoft leveraged COA to employ multidimensional spatiotemporal diversity and hot shuffling of variants, hence effecting software execution behavior encryption. ChameleonSoft also employs multi-mode, autonomous, situationally-aware recovery system. Further, it adjusts system shuffling and recovery policies at runtime to meet the continuous change in its operational environment. A prototype implementation, and a simulation and analytical study were presented to discuss the performance impact of CBE on the system and to illustrate the applicability of the presented approach. The studies showed that CBE can encrypt the execution behavior by confusion and diffusion induction at a reasonable overhead. There are several interesting challenges to be addressed in the future. These include autonomous detection and profiling of behavior; adjusting shuffling decisions based on that profile; realizing the space diversity; software chameleon-ization including formalizing an automated variant generation system, and presenting alternatives for legacy non chameleon-izable software; and rigorous simulation and experimental evaluation of confusion, diffusion and shuffling policies and mechanisms.

VII. REFERENCES

[1] A. Avizienis and L. Chen, “On the implementation of n-version programming for software fault tolerance during execution,” IEEE COMPSAC 77, pages 149–155, 1977.

[2] B. Randell, “System structure for software fault tolerance,” IEEE Transactions on Software Engineering, 1:220–232, 1975.

[3] J. C. Knight and N. G. Leveson, “An experimental evaluation of the assumption of independence in multiversion programming,” IEEE Transactions on Software Engineering, 12(1):96–109, 1986.

[4] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. J. P. Kelly, “An experimental evaluation of software redundancy as a strategy for improving reliability,” IEEE Transactions on Software Engineering, 17(7):692–702, 1991.

[5] F. Cohen, “Operating system protection through program evolution,” Computers and Security, 12(6):565–584, Oct.1993.

[6] C. Pu, A. Black, C. Cowan, and J. Walpole, “A specialization toolkit to increase the diversity of operating systems,” ICMAS Workshop on Immunity-Based Systems, Nara, Japan, Dec. 1996.

[7] J. E. Just and M. Cornwell, “Review and analysis of synthetic diversity for breaking monocultures,” ACM Workshop on Rapid Malcode (WORM ’04), pages 23–32, 2004.

[8] D. A. Holland, A. T. Lim, and M. I. Seltzer, “An architecture a day keeps the hacker away,” SIGARCH Computer Architecture News, 33(1):34–41, 2005.

[9] M. Chew and D. Song, “Mitigating buffer overflows by operating system randomization,” Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, Dec. 2002.

[10] J. Xu, Z. Kalbarczyk, and R. K. Iyer, “Transparent runtime randomization for security,” 22nd International Symposium on Reliable Distributed Systems (SRDS’03), pages 260–269, 2003.

[11] J. C. LKnight, J. W. Davidson, D. Evans, A. Nguyen-Tuong, C. Wang, "Genesis: A Framework for Achieving Software Component Diversity," Technical Report AFRL-IF-RS-TR-2007-9, University of Virginia, January 2007

[12] S. Forrest, A. Somayaji, and D. Ackley, “Building diverse computer systems,” 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), pages 67–72, 1997.

[13] B. Salamat, T. Jackson, A. Gal, and M. Franz, “Intrusion detection using parallel execution and monitoring of program variants in user-space,” Eurosys 2009, April 2009.

[14] B. Salamat, A. Gal, and M. Franz, “Reverse stack execution in a multi-variant execution environment,” Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS’08), June 2008.

[15] B. Salamat, A. Gal, T. Jackson, K. Manivannan, G. Wagner, and M. Franz, “Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities,” International Workshop on Multi-Core Computing Systems (MuCoCoS 2008), March 2008.

[16] T. Jackson, B. Salamat, G. Wagner, Ch. Wimmer, and M.Franz, “On the Effectiveness of Multi-Variant Program Execution for Vulnerability Detection and Prevention,” International Workshop on Security Measurements and Metrics (MetriSec 2010), September 2010.

[17] M. Franz, “E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism,” New Security Paradigms Workshop 2010 (NSPW 2010), September 2010.

[18] T. Jackson, Ch. Wimmer, and M. Franz, “Multi-Variant Program Execution for Vulnerability Detection and Analysis,” Sixth Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW’10), April 2010.

[19] B. Salamat, T. Jackson, G. Wagner, Ch. Wimmer, and M. Franz, “Run-Time Defense against Code Injection Attacks using Replicated Execution,” IEEE Transactions on Dependable and Secure Computing. IEEE Computer Society, 2011. doi:10.1109/TDSC.2011.18

[20] L. Hatton, “N-version design versus one good version,” IEEE Software, 14(6):71–76, 1997.

[21] Charles P Pfleeger, Shari Lawrence Pfleeger., Security in Computing, Prentice Hall, Third Edition., 2003, page 62,ISBN:0-13-035548-8.

[22] James Wood, Kelsie Jackson. (2011, Jun). How Cephalopods Change Color. Available: http://www.thecephalopodpage.org/cephschool/

[23] F. Cohen, “Computer Viruses,” PhD thesis, University of Southern California, 1985.

[24] E. H. Spafford, “Computer viruses as artificial life,” Journal of Artificial Life, 1(3):249–265, 1994.