AN APPROACH TOWARDS AN AUTONOMIC COMPUTING PROTOTYPE REFERENCE ARCHITECTURE

by

Trevon Williams

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Cyber Security

Charlotte

2019

ABSTRACT

TREVON WILLIAMS. An Approach Towards an Autonomic Computing Prototype Reference Architecture. (Under the direction of DR. TOM MOYER)

Autonomic Computing is a grand challenge that strives to create self-regulating environments. The end goal, is to create an environment that can self-optimize, self-protect, self-heal and self-configure based on built-in, formulated, and generated logic to address to concerns of the growing complexity in managing computer systems. To do so, Autonomic Computing requires advancement in a plethora of scientific and economical fields. In theory, Autonomic Computing implementations can handle system complexity by abstracting aggregated environment information used in policy bases systems. This allows autonomic elements in the system to administer configuration changes and policy at the moment remediation is needed. In this paper we outline the ideas and direction of Autonomic Computing, the reference architecture we formulated, and the autonomic features we focused our architecture around. The end goal of this research is to aid in the creation of light weight autonomic computing prototype reference architecture, which utilizes SaltStack as an autonomic element. Later, we explain why we thought it was important to utilizes specific open sources tools and academic ideas.

DEDICATION

This work is dedicated to the Lord. He is the author and finisher of my faith.

ACKNOWLEDGEMENTS

I would like to take time to acknowledge the faculty members and fellow students who helped get this work to its current state. I also am thankful for the privilege to be a GAANN fellow and to receive aid from UNC Charlotte's graduate school through the GASP grant.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACS	An acronym for Autonomic Computing System.
AFL	An acronym for American Fuzzy Lop.
CI/CD	An acronym for Continuous Integration and Continuous Deployment.
CIM	An acronym for Common Information Model.
DIE	An acronym for Distributed, Immutable, and Ephemeral.
DMTF	An acronym for Distributed Management Task Force.
IETF	An acronym for Internet Engineering Task Force.
LXD	An acronym for a Linux Container system.
NFV	An acronym for Network Function Virtualization.
OS	An acronym for Operating System.
OVS	An acronym for OpenVSwitch.
PMAC	An acronym for Policy Management in Autonomic Computing.
POP	An acronym for Plug-in Oriented Programming
SaltStack	An automation toolkit used to manage many systems.
SARA	An acronym for a Survivable Autonomic Response Architecture [19].
SDAR	An acronym for Sensors, Detectors, Arbitrators, Responders
SDN	An acronym for Software Defined Networking.
VM	An acronym for Virtual Machine.

CHAPTER 1: INTRODUCTION

Large computer networks can be found in data centers, universities, and well-established companies. Networked computer systems are typically found containing a composition of machine abstractions, network configurations, operating systems and novel defense strategies. These systems are arguably racing towards greater complexity, connectivity, and efficiency of use, as new methodologies and advancements are implemented in an ever-adapting eco-system. Systems are becoming more cumbersome to manage and advancements require more specialized knowledge to implement and maintain. Industry leaders, researchers, academics, and IT practitioners have long since agreed that this is an issue worth solving. Initiatives have led to novel advancements like high-level orchestrated deployment techniques, automated patches via agents, and advance configuration management tools. Efforts have pushed back the inevitable of not being able to fully manage legacy systems riddled with technical debt. The idea of Autonomic Computing is not a new idea and was described indirectly in modern literature as Computer Immunology in 1998 [1]. Burgess made the case that "the time is right to build not merely fault tolerant systems, but self-maintaining, fault corrective systems". Computer Immunology, similarly to autonomic computing, is heavily inspired by biological system's ability to self-regulate and how they manage themselves in a diverse breadth of environments.

The term Autonomic Computing was coined in 2001 by IBM [2] proceeding a keynote from Paul Horn and the release of the Autonomic Computing manifesto [3] as it marked the beginnings of the Autonomic Computing initiative. Autonomic Computing strives to abstract the complexity of optimal states and system policy, to enable executed actions and decisions by autonomic control loops and decision-making systems. Decisions then must govern a heterogeneous collection of computers based on proposed goal policies without causing system failures. From a security perspective, integrating responsive logic into globally aware systems improves structural integrity by decreasing the response time between threat detection and mitigation. In reducing the surface of direction human administration, current design implementations of orchestration technologies increase potential risk if attackers are able to control responses.

Initially, we embarked with the goal of developing a system that was able to react to custom triggers within the environment to create a more agile system. This led to iterations of environment designs spanning custom automation tools to utilizing open source solutions like Ansible and SaltStack. After we came across SaltStack and saw how we could utilize their reactor system to create an event driven environment, we returned to the common literature to find if work had been done in this area. This led us to the field of autonomic computing and the goal of creating a self-managing environment. We explored prototype and testbed implementations to explore solutions and how custom orchestration tooling was used in place of autonomic elements.

In this thesis, we sought to outline a rudimentary prototype architecture of a system with open source tooling and methodologies to explore security implications in future work. Utilizing Kephart's guidance from Research Challenges of Autonomic Computing [4], we make the claim that we have identified a tool that can be utilized to create autonomic elements. We then to detail how we propose to use this tool to create of an autonomic computing prototype. This research fits within one of the three sub-branches of the autonomic computing research framework which includes: autonomic elements, autonomic systems, and human computer interactions.

1.1 BODY OF WORK

Autonomic Computing has five fundamental goals, as referenced in the Internet Engineering Task Force's (IETF) RFC 7575 [5] and described in Research Challenges of

Autonomic Computing [4]. The system must be self-managing, self-configuring, self-optimizing, self-healing, and self-protecting. These self-x features are implemented by a collection of autonomic elements. Each element provides some sort of functionality to each automated processes through advanced utility deducing algorithms. When broken down into smaller elements, this body of work requires efforts in system design, learning integration, and algorithmic decision-making processes that can adapt to the changes they and other similar elements make in the environment. In this paper, we explore self-healing, self-protecting, and self-configuration as we structure our reference architecture around proposed work, drafted frameworks, and testbeds in both network and infrastructure rich environments. Contained in our goal of describing our architecture, we make an argument to show how SaltStack can be used as an autonomic element and why using an open source enterprise tool creates a realistic architecture that can increase the rate of feedback and adoption of autonomic research.

This paper details some design philosophies retrieved from the Autonomic Computing literature, information security engineers, and reference architectures. Then we detail related works spanning the technological implementations considered in the proposed design which is followed by the specific technologies. After providing this overview, we detail our reference architecture and give a high-level overview of what specific implementation details we believe can be accomplished with the specific components referenced.

1.1.1 Design Philosophies

In this section, we explore a few important design philosophies. Our intent when conducting this research, was to design a reference architecture that can enable a large collective of researchers to work in a similarly complex system. We hope that the architecture will be expanded, edited and fully created in future work. While considering the environment's structure, we focused on reducing the need to build custom tooling for a simulated environment by suggesting open source tools and tested technologies.

1.1.1.1 D.I.E

D.I.E is a design paradigm and resiliency model, that suggests systems should be distributed, ephemeral, and immutable. Our first encounter with this philosophy came from a presentation [6] by Sounil Yu in which he suggests design considerations for the next era of computer security. Yu highlights key problems, solutions, and architectural focuses to guide design decision for more robust and resilient systems. The struggle within the core challenges in IT environments have left some IT practitioners and security engineers at odds due to their different goals. As solutions and security teams matured, advancements in information gathering services, automation tools and pipelines have brought security and IT teams closer than before.

The blend of approaches, driven by modern advancements and research, are leading to shared design goals in IT and security disciplines which parallel inherit properties of Autonomic Computing. Autonomic systems are secure by design, resilient, and scalable. These properties align with the end goals of the D.I.E philosophy. We believe that if we can propose a distributed, ephemeral and immutable autonomic architecture that can work within the confines of the autonomic definition, we can negate some future challenges by solving the implementation obstacles required for an architecture to exhibit this philosophy.

1.1.1.2 Unified Autonomic Layer

When implementing an autonomic computing system, we believe that we need to create a plane of unified operating. Meaning, a layer where autonomic elements and orchestration elements operating within the same level of abstraction. This idea is similar to how operating system kernels work by abstracting low-level hardware details away from system developers, operators and users to create allow for feature extending when needed. In doing so, collectively we can increase the amount of applicable testing in a wider range of live environments and the adoption of advancements. With one level of unified communication for autonomic elements, system logic, and human interaction, we believe that researchers can focus on building, adapting, and expanding relevant work.

1.1.1.3 State Policies

Policy has two working definitions in autonomic literature. As presented in [7], policies can be in the form of a configuration used in rule-based engines or it can be a composition of a set of rules that determine the behavior of the system. From this paper, we focus on a definition of policies that is a synthesis of these two ideas.

Policy creation and implementation are arguably two of the most important aspects of securing any computer system. When drafting methods for generating policy to store in autonomic registries; we considered generating policy based on initial element requirements, dynamic policy state generate, or static policy generation. This is supported in literature, and requires advancements in hot swapping, machine learning, case-based reasoning, and data clustering techniques as they relate to Autonomic Computing. Drawing inspiration from the findings of [8], we wanted to understand how to suggest an interactive interface that bridges the intuition of humans and processing power of machines and algorithms. As we approached policy, we divided it into three layers where generation can be enforced in our environment: include network policies, custom software policies, and infrastructure policies. These policies do not take way from or add to active, goal, or utility function policies defined in [9].

We can enforce policy to effect lower networking layers when generated to suggest and enforce autonomic element or resource communication in an autonomic computing environment. This would require a registry to account for network connections and elements as applied to networked autonomic elements. From here, the abstract policies of [9] (action, goal, utility) can be applied. Following this point, we assume the autonomic environment contains a continuous integration and continue delivery (CI/CD) pipeline that can extract deployment resource requirements from template files. Using best practices and a CI/CD pipelines, smart swaps can be built into an interactive policy generator to enable a greater efficiency of software resource administration at the point of deployment. Assuming git is used as a registry, we can ensure that the registry is up to date at the moment software is committed and testing and resource extraction is complete in CI/CD pipelines.

We hypothesize by using elements present in continuous deployment pipelines, environments can take advantage of containerized services to extract pertinent information from configuration files. Traditional infrastructure elements like active directory, mail, or web host elements present a balance of use-case and security focus and deserve further discussion.

1.1.1.4 Response Loops and Analysis Criteria

Static policies are seen to be effective in the initial configuration steps of system and services but they lack the ability to change behavior beyond the originally defined scope. Thus, dynamically policy generation is essential for self-optimizations, self-healing, and self-configuration. Dynamic policy generation depends heavily on artificial intelligence, robust policy libraries, and the ability to process and understand system behaviors. The latter is arguably the most crucial, as building a behavioral model can not only provide guidance for optimization directions, but can lead to more secure policies that converge to an optimal state that supports defined software actions.

In order to build behavior models, one must understand how models have been typically generated by human operators. At this point, we find recommendations for suggestive reactive loops like observe, orient, decide, act (OODA), monitor, analyze, plan, execute (MAPE), and FOCALE foundation, observe, compare, act, learn rEason (FOCALE). These reactive loops each share a core philosophy. An intelligent system or structure needs a foundation to build upon or must create their own understanding to expand when influenced by environmental factors. They must then detect reactions based on policy, self-preservation algorithms, or efficient decision constructing logic. These feedback loops are utilizing when building software and compiling granular log and threat data, but we believe enabling systems to emulate these feedback processing loops can lead to more effective policy guidance.

1.1.2 Network Design

When creating a complete autonomic computing reference architecture, we found that we must understand the configuration and implementation of software providing resources just as much as network resources. We have found a lack of prototypes in literature that focus on including network, virtualized, and physical resource managing autonomic managing elements. Due to the nature of this problem, we decided to focus on proposing an architecture that includes the ability to impact both. This task comes with an inherit challenge, proposing a system that is usable by a greater breadth of researchers by utilizing technologies that work without disrupting testing. When drafting our architecture, we settled upon suggesting Software Defined Networking solutions via the implementation of OpenVSwitch (OVS).

1.1.2.1 Software Defined Networking

Software defined networking (SDN) is a relatively new, networking paradigm in which an SDN controller is utilized by centralizing route management of a group of decentralized switches. This concept is similar to how a collection of routers communicate amongst themselves to send route updates. By centralizing route polling and issuing, SDN implementation reduce network congestion of route determination from a N*N complex route determination algorithm to a N complex algorithm. SDN works as a framework that abstracts networks, to control them programmatically. This reduces the amount of interactions between network components and managing software. In common implementations of SDN networks, controllers utilize the control plane to update network flows to OpenFlow enabled switches in a southbound manner. Utilizing the high programmability of SDN controllers, we have found in literature that we can enforce access control policies, simulate one to many connections [10], dynamically update firewall rules, and implement fault tolerant design [11]. We perceive that when enabling autonomic elements to interact with network resources, we can understand how autonomic elements work when interacting in a complete system. It also enables us to understand how system resources are consumed or allocated in a dense environment.

Some challenges of using SDN include attempting to dynamically update network configuration depend on algorithmic considerations. Drawing from the guidance of Abstractions for Network Updates [2], we find that introducing consistent and well-defined behaviors when transitioning between network configurations or states is required. In kinetic, a system built on top of the NOX Controller, a group of researchers detail an algorithmic implementation that was created to handle certain cases for updating flows in real time while taking into consideration the packets in transit. This aligns with the goals of a complete autonomic element.

1.1.3 Virtualization

Virtual machines and containers are important technologies that allow us to simulate a large and diverse environment with minimal hardware. A container is a modern software abstraction that simply isolates a process from the rest of the system. Virtual machine takes this abstraction a step further and virtualizes all aspects of an operating system for use at the cost of utilizing hosts resources. Virtual machines and containers templates enables us to create an ephemeral environment that allows elements to rapidly deploy, scale and manage packaged software and user environments in a highly effective way. In our architecture, we suggest this technology to create an environment that can scale as more hardware is introduced to a virtualized cluster.

1.1.3.1 Linux Containers

In our architecture, we reference the container system, Linux containers (LXD). LXD is a container managing system that is designed around a privileged daemon process that provides a REST API for management. LXD's implementation abstracts many low-level details through the use of YAML configuration files and container templates to enable fast deployment and resource allocation in any environment. This feature is similar to docker and other containerized technologies. Containers are an important element of our architecture. As stated in the prior 'State Policies' section, we assume that the DevOps philosophy of rapidly encapsulating and deploying software projects into cloud and on-premise environments is implemented. We perceive that containers can enable IT practitioners to automate the creation of models from information from deployment pipelines. This strategy will allow systems to build an understanding of on initial behavior and to distribute accumulated knowledge to system agents assuming an effective implementation of learning models is deployed.

1.1.4 Orchestration

Orchestration facilitates rapid prototyping, deployment, and updating of infrastructure design and architecture. Orchestration centralizes system configuration and can easily can searched for unwanted system changes. Burgess's cfengine [1] is very similar to modern orchestration technologies such that abstract definitions and configurations are applied to a diverse lineup of computing systems. Orchestration can also be used to enforce policy, reduce misconfiguration, and simplify management all while reducing the man power required to setup and configure a large collection of machines. The deployment of orchestration technologies enables a higher efficiency of deployment to cloud or on-premise environments and will enable a new methodology of dynamic load balancing in autonomic systems unable to find resources in the on-premises environment. Dynamic scaling and re-scaling is an essential idea encapsulated in self-optimizing systems where autonomic elements need the ability to know when to scale and when to reduce resources used on the environment by the means of a resource broker. Self-healing and self-configuring systems are dependent on well-defined states and the implementation of extensible orchestration technologies.

CHAPTER 2: RELATED WORK

When considering a design strategy, we found it important to explore the relevant work around autonomic computing prototypes to understand approaches of other researchers. In this section, we mention some of the many novel approaches and frameworks for implementing autonomic environments and what design strategies the researchers focused on highlighting. Some of the methodologies and implementations below influenced our architecture direction due to their importance whereas others are referenced due to their ingenuity. Understanding that the nature of autonomic frameworks and architectures span a diverse breadth of implementations as mentioned in [12]. Our architecture we focuses on biologically inspired, agent computing and policy enforcement methodologies.

2.1 Autonomic Research

Autonomic system research spans a multitude of disciplines, approaches and implementations. We do not in any way intend to establish a complete synopsis of novel research contained in the literature. Below, detail approaches of related work to highlight strategies for enforcing autonomic environments.

2.1.1 Autonomic Computing Architecture

Work done in this paper [9] outlines architectural requirements, components, and system structure of an autonomic computing environment. It is important to understand the goals and expected behavior as we draft or prototype reference architecture. Reasons that lead us to suggest a unification of autonomic operating level and technologies stem from the expected autonomic behavior: an element must be self-managing, capable of establishing and maintaining relationships, and able to manage its behavior and relationships as defined by system policy [9]. Policies levels are later defined in this work and have been previously referenced.

2.1.1.1 Research Challenges of Autonomic Computing

Our research group came upon autonomic computing when exploring how realistic is it to implement reactive environments. Research challenges of Autonomic Computing [4] provides a brief history and high-level context for approach autonomic research. Our feature is relevant to a component contained in the autonomic elements area of research that suggests generic autonomic element architectures, tools and prototypes are formed to enable further research. [12] Kephart then describes attributes of autonomic elements, methods to be displayed in implementations and prospective challenges that can be focused on.

2.1.2 Agent Based Design

Multi-agent systems are a popular design consideration in Autonomic Computing implementations. Multi-agent systems typically utilize a centralized, policy possessing entity of varying definitions that provide agents with execution guidance? In a well-designed system, the multi-agent approach enables decentralized and distributed management in systems with greater complexity and fault potential. In doing so, multi-agent systems create the need for attestation of agents but allow for a highly efficient method of distributing environment information. We find that the multi-agent design in literature utilizes agents as autonomic elements, which reference a centralized registry to understate future courses of execution and provision.

2.1.2.1 Multi-Agent Systems Approach to Autonomic Computing

Unity [13] is a de-centralized architecture built by IBM that uses autonomic elements to create a complete multi-agent system to enable autonomic features in a software rich environment. The design focuses on abstracting compute resources into autonomic elements to enable self-optimizations and self-healing properties through the use of its three main components: a registry, policy repository, and sentinel elements. Unity uses utility functions to simplify how machines in the environment make decisions to manage themselves. Management is based on abstracted resource need provided via utility functions. Some of the important properties we find in this system include how Unity utilizes a registry, enables self-healing, and utility functions for self-optimizing.

The autonomic elements in the Unity, query a registry that provides information pertaining to how the system should handle it's initializing, what services it can communicate with, and what policies it should adhere to. This design feature abstracts the configuration of dependencies away from single elements to force elements to be disjoint when interaction is not required. The registry holds information relevant to each node in the cluster the registry is housed in. Unity then uses the cluster as a means for self-healing in a novel way. Unity is designed by the methodology that two elements of the same cluster should not be on the same machine which enables a cluster to recreate an element if it is not present in the whole environment. This self-healing functionality spans to all autonomic elements including the registry.

Unity also takes a unique approach towards policy generation and management. Unity implements a policy environment 'Policyscape' that utilizes manually constructed policies and aggregated information to create an approximate configuration for new policies.

2.1.2.2 A Policy Language Prototype Implementation Library

An implementation towards a unified policy language [7] is needed to condense policy research. Hypothesizing how to implement self-managing systems has led to the popular notion of policy-based management. Policy based management facilitates an ease of deployment for an entire base of applications depending on how policies are implemented. Static policies leave little room for the system to be dynamic without human interaction. This comes from the understanding that systems are complex as they interact in varying degrees of freedom and require different level of policy implementation. As reference in [9], [7] defines policies in different levels to address the dimensions policies need to be enforced. This includes fine grain policy control, medium-grained optimizations, and coarse-grained behavioral shifts.

2.1.2.3 Policy Management for Autonomic Computing

The Policy-Based Management of Networked Computing (PMAC) [14] systems is defined and outlined to provide a model for policy managers to interacted with a managed resource in an autonomic system. Drawing inspiration from the Common Information Model (CIM), PMAC builds upon the work of the Distributed Management Task Force (DMTF) to create a platform specifically for IBM's autonomic computing architecture by breaking the system into two parts. When together, the autonomic manger and managed resource create a system that can monitor and analyze computer resources to execute planned course of actions. Integrating this system into our architecture increase the prototypes use of standard technologies.

2.1.2.4 Architectures and Techniques in Autonomic Computing

In a survey conducted on the state of frameworks, architectures and techniques in autonomic computing [12], we find that there is are a plethora of approaches when attempting to create solutions to address autonomic system design requirements. To enable self-x features, designs utilize hot-swapping, machine learning, case-based reasoning, data clustering, hybrid survivability models, probing techniques, control theory and attestation techniques. This paper [12] presents interesting correlations between effective methods and self-x abilities. For example, hot swapping is an important technique that injects monitoring abilities into live code and is effective in enforcing self-configuration in many frameworks. The break down found of techniques and self-x abilities is important and has influenced proposed future work.

2.1.2.5 A Symbiotic approach to Autonomic Computing

Kephart highlights an important notion in this paper [8]. After the introduction and bid for utilizing smart swaps in systems that enable close human and computer interactions, Kephart concludes that researchers aren't as accurate as they assume when generating utility-functions for autonomic environments. This idea has influenced our approach directly through motivations to suggests work on to extract autonomic features in CI/CD pipelines and how we approach our recommendations for constructing autonomic prototypes. We envision it will take a prototype that is able to evolve to enable researchers to collective realize the goal of solving the grand challenge of self-managing systems.

2.1.3 Testbed

Testbeds are important when considering autonomic prototypes. We believe that merging the utility of testbeds and prototypes can enable researchers to leverage work in literature in more effective ways. We drew inspiration from design motivations of autonomic testbeds due to the specific need and use case some testbeds were created to address.

2.1.3.1 Towards an Autonomic Computing Testbed

Components and plug-ins drive the design of Automat [15]. Automat focuses on allowing users to install, manage and launch software experiments in a virtual environment by enabling pluggable controllers into a testbed system. The testbed then leverages aggregated system diagnostics to provide pertinent information in an interactive display. Automat strives to be an open and easy to test environment that leverages VM technology to provide an ability to test autonomic applications. Observing this, we want to understand how we could integrate enterprise grade components into an architecture that could also drive rapid feature testing. Virtualization advancements have now enabled a light-weight architecture that doesn't have to depend only on a collection of virtual machines for live migration, configuration, and snap shot features. Utilizing container systems, we suggest a way to create a more robust environments that enables the testing of autonomic elements without increasing the cost of research hardware.

2.1.4 SDN Research

In this section we observe autonomic like features implemented in SDN research. The research contains advancements in automated changes in SDN as well as functionality that will further support robust autonomic elements in feature rich environments.

2.1.4.1 Adaptive Flow monitoring

Adaptive flow monitoring, as covered in [16], provides an overview of current standards in SDN controller implementations. Describes alternatives for constantly polling SDN enabled switches for event data by implement a new procedure and method format which is implemented in OpenFlow 1.4. This research is important for when SDN technologies are enabled in an event driven environment. In larger scale environments, reducing the overhead of route retrieval can lead to more efficient and robust network implementations.

2.1.4.2 Customizable Autonomic Network Management

Autonomic network management and Software defined networks align with similar methodologies and goals. This paper [17] compares proposed frameworks such as autonomic networking integrated model and approach (ANIMA), autonomous network management (ANM), and Autonomic failure and recovery (AFRO) and then later introduces autonomic OpenFlow (AutoFlow). AutoFlow implemented in the GEANT testbed [17] and is proposed to address problems in future networks with unforeseen unmanageable levels of complexity. Like some autonomic systems covered previously, the GEANT testbed enforces a policy-enforcement layer in AutoFlow that allows for middle-box-specific traffic steering in a large network.

2.1.4.3 Virtual function placement and traffic steering

In literature, we see how network function virtualization (NFV) and SDN enabled environments can create a highly flexible network environment. Rich networking environments typically include interactions of SDN controllers with firewalls, proxies, and cache and policy engines. In efforts to increase the efficiency of route determination and assignment, [18] has reduced the challenges of routing and middle box network function interactions into a mixed integer linear programming problem. This greatly increases the ability to run SDN controllers which are interacting heavily with network resources in more light ware software abstractions like containers. We use this paper to explore how to increase the efficiency of use different SDN controllers in a dense virtual environment.

2.1.4.4 Cognitive Autonomic Fault Management

Control loops are important to autonomic and reactive environment experiments. The paper [11] propose a method for implementing control loop algorithms in an SDN controller. The hopes of doing so will enable a controller to respond to one or many faults in network routing immediately. This research was conducted due to most controls loops inability to fix failures that us outside the defined scope of failure cases. By implementing the fast flow setup (FFS) algorithm in controller code, the researchers are increase the time in which controller logic can redirect traffic when routes decrease in integrity or fail.

CHAPTER 3: APPROACH

We believe a robust reference architecture is a needed platform that can expand integrate past and current developments to create a uniform environment for testing autonomic decisions making, policy generation and execution, and machine learning model accuracy in deployed software projects and environments. Our goal is to influence the creation of a suite of dynamic components that drive testing of a complete autonomic system which is agnostic to the field of autonomic system research.

Our design is motivated by understanding how services and systems are introduced to systems and interact over time. One of the most popular paradigms for deploying software and services requires configuring a pipeline to increase the speed and efficiency of deploying software. DevOps effectiveness has motivated IT and security teams to collaborate in creating DevSecOps or SecOps systems that can deploy security features and functionality just as efficiently as software updates. Without deterring from an already effective method of deploying software, we theorize that an effective autonomic system can work effectively with software developers, service managers or security teams to define a baseline policy as the software is being tested, packaged, containerized and deployed by integrating autonomic policy generation into these pipelines. By suggesting the inclusion of an agent into this system in our reference architecture, we hope that a prototype will help researchers to understand how specific policy evolves from deployment to a simulated life-cycle.

SaltStack provides configuration management, orchestration and an agent or agentless ability to administrator machines. We believe that SaltStack can serve as the middle-ware for each autonomic computing component while being an autonomic element itself with the ability to enforce autonomic feedback loops and managed environment resources. SaltStack is industry tested and proven to be effective at state-based reactions. The system is highly extensible via its plug-in oriented programming (POP) system AND recently introduced a machine learning pipeline, umbra, that uses the POP system to create an effective way to train and use machine learning models.

In our experiment described below, we detail how we iteratively created an automated processes to launch the test environment. In our third version of our lab, we utilized terraform with the LXD formula to launch the environment which is described in the image in the preceding section. We detail the creation time of the environment and the amount of time it took to migrate a container from one network zone to the other using SaltStack's LXD formula.

CHAPTER 4: ARCHITECTURAL COMPONENTS

In this section we provide a baseline for our reference architecture and our progress towards this prototype. The technologies highlighted below can be integrated and extended into the OpenStack environment by implementing plug-ins or supported features. The structure of our components in our reference architecture are influenced by the Survivable Autonomic Recovery model (SARA) [19]. This is due to our motivations to enable security into the architecture by default.

4.1 Components

In order to visualize or proposed architecture, we have provided a high level via of the rudimentary system level architecture.

In this diagram below, we show four zones which are networked via an OpenVSwitch. This enables us to allocate each zone with its own unique network bridge. Each SDN enabled bridge features its own filtering and policy requirements. Each zone is containers an LXD cluster by nested a remote LXD host inside of a top level cluster. This enables us to test more fine-grained SDN routing algorithms. We found that by using LXD, we do not lose a feature parity with virtual environments as described in Automat [15]. Network configurations, LXD clusters and machine provisioning can be dynamically created using a automation programs to initialization configuration. We found this to be incredibly helpful for changing environments rapidly without losing progress.

Outside of the four zoned LXD cluster, we have a master container that hosts the salt master and SDN controller and a snort container that houses an open source IDS. Utilizing a default configuration template of RYU, we can forward network

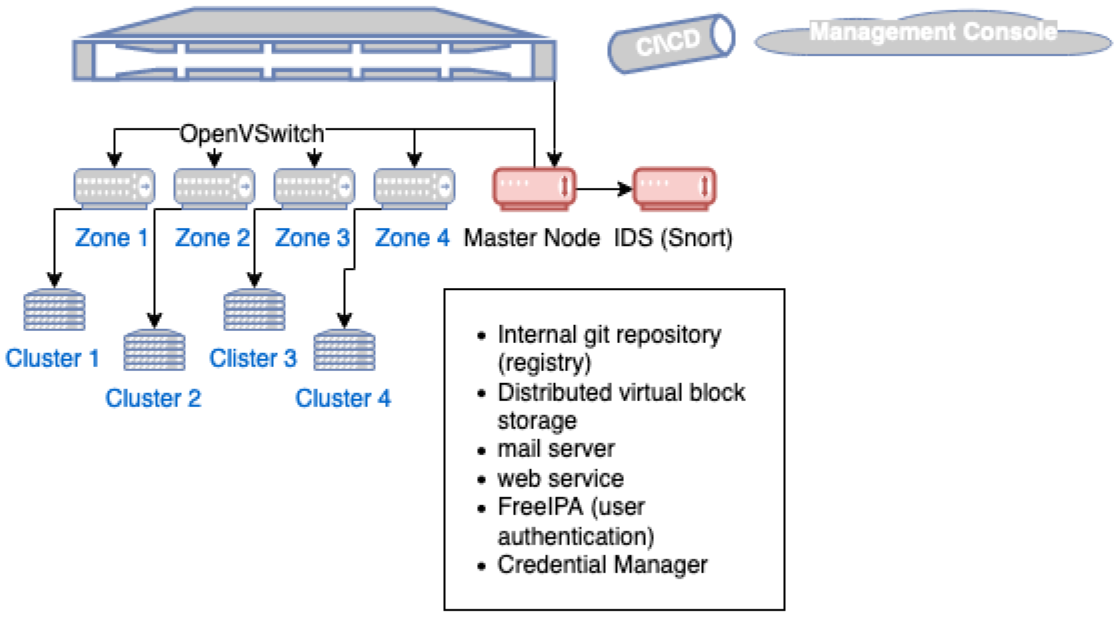

Figure 4.1: High level reference architecture diagram

packets through snort in a simple and manner at the moment of configuration. Within the environment, we provision five initial elements: a Gitlab self-hosted instance, a database, FreeIPA (an authentication server), a mail server, and a website. These services will serve to provide an easy baseline environment for testing autonomic elements. In this section, we will detail strategies for integrating these components and how they are relevant to creating an autonomic architecture.

4.2 SaltStack

SaltStack was originally designed to be a remote execution tool designed around being able to execute commands effectively and efficiently on a wide breadth of servers. By integrating ZeroMQ, Salt adapts well to increased scale. ZeroMQ is a synchronous messaging library that reliably uses multicast protocols to reduce network traffic between a group of hosts that typically receive similar messages. Salt also supports its own implementation of Reliable Asynchronous Event transport (RAET) to enable application layer level queuing and socket layer encryption as described in its documentation. In attempts to find a tool that support attestation of elements natively and could manage itself in a distributed way, we found SaltStack to be the go-to tool of choice. SaltStack also integrates with environmental utility effectively

4.3 Software Defined Networking

We used the RYU SDN controller in our baseline environment due to its ability to rapidly prototype SDN experiments. For our controller code, we attempted to implement a fault tolerant routing algorithm implemented in[11] to see how it would perform in our environment. We also tested utilizing a REST implementation of RYU to dynamic enforce firewall rules. To extend this feat, we realized that we needed to development a more generic suite of event triggers to align with the three abstract layers of policies defined in [9].

CHAPTER 5: BASELINE PROTOTYPE

When first embarking on creating a prototype based on our reference architecture, we experimented with creating it from scratch by automating the process in bash and python but we ultimately have decided to build around terraform. With terraform, we can provide the ability to quickly create and destroy the prototype in a Linux environments. In this section, we detail how long it takes to construct the baseline environments, the features each version contains, and the amount of time it took to execute a live migration in the environment.

5.1 Foundational Design

When embarking on our design, we initial set out to rapidly create an environment where SDN capabilities were built in without the need for further configuration. This was to aid any users who might not have an understanding of how to configure an environment to test customer controller code. LXD containers allow us to rapidly download, boot strap, and manage virtual environments. The testing that we include below assumes that LXD images have already been downloaded onto the host machine to provide a more uniform and realistic view of how long it takes to rapid create and destroy the environment. We didn't include the time it took to download the images, because the time it takes to download a 300MB image file can vary greatly depending on Internet speed.

In our experience of setting up the environment, we did not work in a static environment and baked in our changes into the automated setup to ensure that tested features, which almost always broke the environment, did not deter our advancement towards progressing the prototype design. This prototype does not have any autonomic features completely implemented.

5.1.1 Version 1

In the first iteration of our prototype, we utilize a bash script to initialize our environment. At this stage, our environment did not reach full maturity because the rate of complexity increased exponentially as we added more design considerations to our scripted environments. In this environment, we created a method for deploy 5*N SDN controlled LXD containers that span four zones and configured the hosts to be configured via ansible. [N specifies the amount of parallel environments created]. We must also note that the bash script crashes more often than we would have like and we have had to stop creation, delete the elements and restart the script to get an accurate run time.

5.1.2 Version 2

The second iteration of our prototype utilized a custom python suite to configure an environment from elements defined in a yaml file. This test bench enabled our prototype to have less hacked together methods of setting static IP addresses and container zones. In this iteration, we began to focus on implementing the ansible tower library but decided to look at other tooling when we realized the ansible's python libraries were in heavy development and could change without notice. Unlike the first version, we parallelized machine creation and network configuration in this setup.

5.1.3 Version 3

In our latest iteration, we utilize terraform to configure and bootstrap our environment. This environment is significantly more mature than the previous iterations as it creates four LXD zones and containers that are clusters that contain the prototypes LXD virtual environments. This environment also features SaltStack. We later detail how long it takes to migrate LXD containers when a fine-grained reactor is triggered

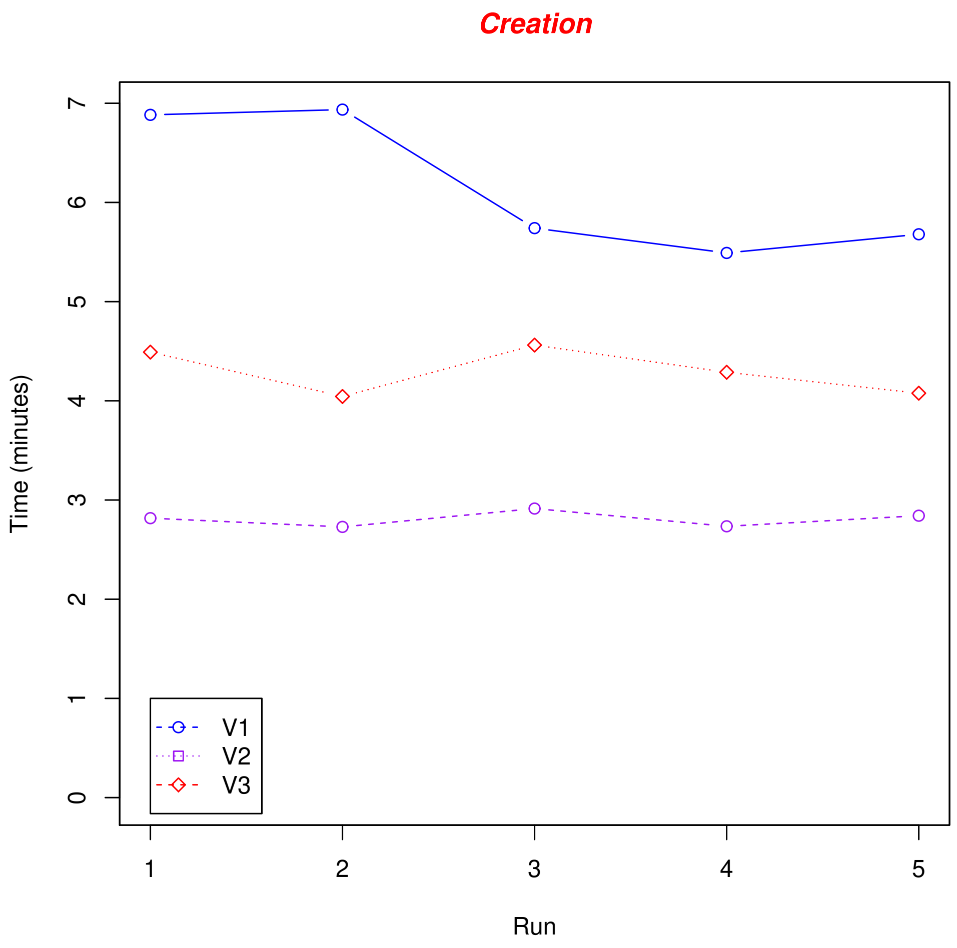

Figure 5.1: Time to create environment

via the SaltStack event bus.

5.1.4 Results

In the figure below, we capture the differences in the time to create our current baseline prototype. The data gathered was ran on a system with the following specs: Ryzen 3700X, 32 GB's of RAM, and a full SSD raid setup. We understand that this may influence our lab creation time and we felt that it was important to include prior to including our data. Our graphs are in the units of minutes where the seconds are out of 100.

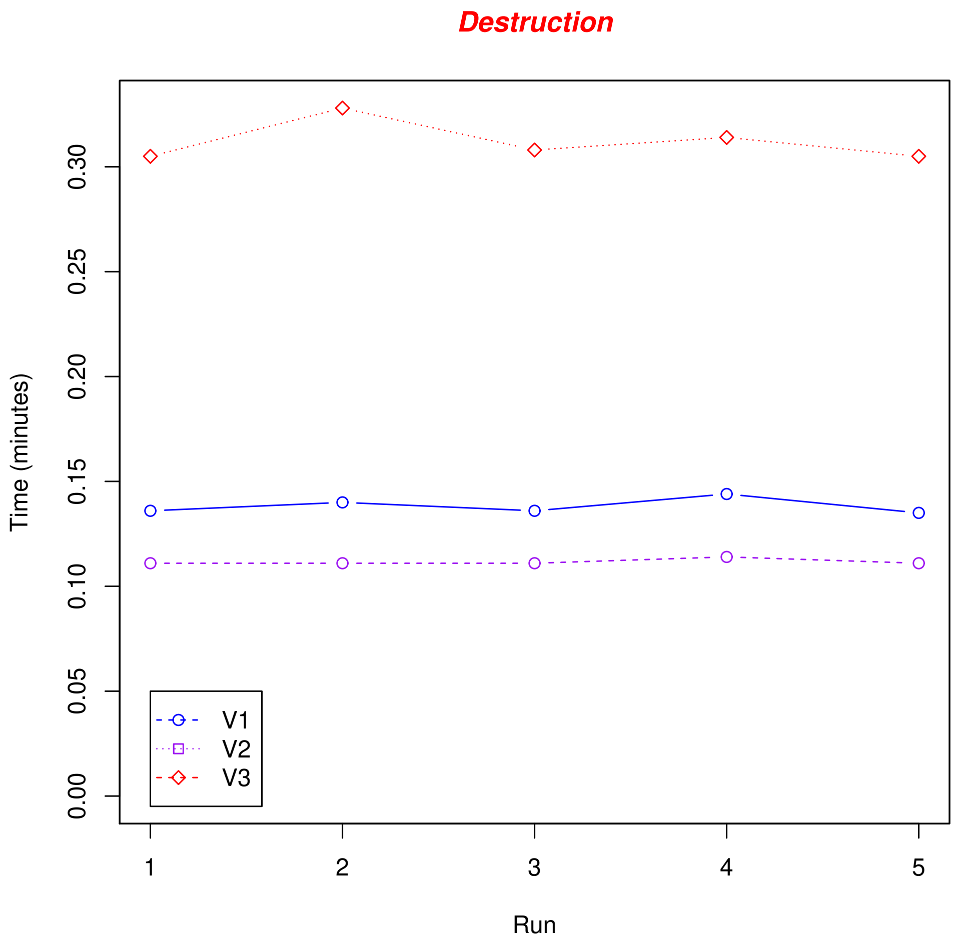

Figure 5.2: Time to destroy environment

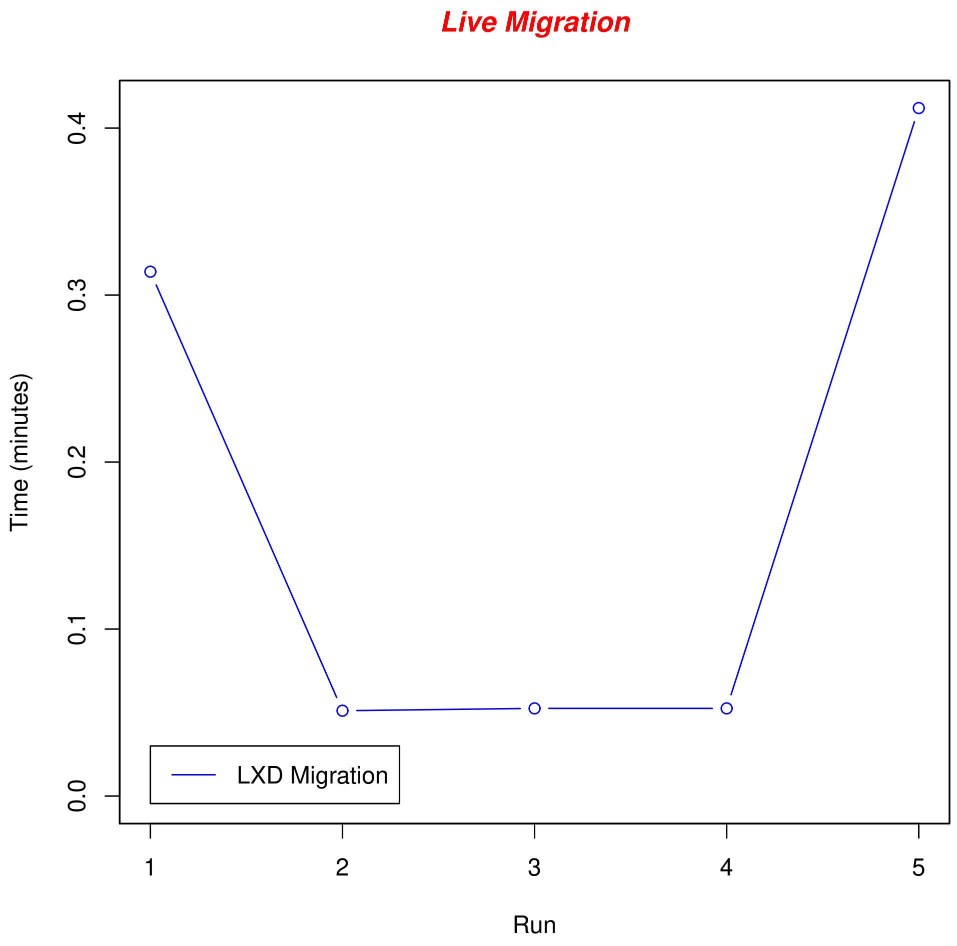

Figure 5.3: Time to migrate an image

5.2 Areas of Development

In regards to policy generation, there is work that needs to be completed to integrate SaltStack states and policy definitions defined in PMAC to work in Salt's render system. It is unknown if there is a way to utilize XML directly in the rendering system. We also need to explore dynamically generation polices in XML or any configuration syntax format. This will drive the testing effective of policies and will hopefully lead to new areas of research.

Autonomic elements in this environment cannot be truly classified as autonomic elements without adequate testing. We hypothesize that utilizing a network based approach of the american fuzzy lop (AFL), we can provide a substantial baseline for creating testing suites to drive autonomic behavior in a effective way. Incorporating the fuzzing methodology of testing can aid in finding gaps in assumptions to help enrich research efforts.

An agent also needs to be developed for the integration into a CI/CD pipeline to test how effective and efficient generating policy in an automated by guided fashion can be. After adequately training a model to understand how permissions correlate to container definitions, we think that dynamically generating policies can increase the integrity of autonomic environments.

CHAPTER 6: CONCLUSION

We provided an overview of reference architecture for an autonomic system prototype and details on the progress we have currently made so far towards this goal. Our system attempts to be a complete autonomic prototype and we reference the implementation of different technologies in an environment without a large resource constraint. In our results, we find that we can provide a relatively short time to bootstrap the baseline environment. We believe this trend will continue as we design a more complete prototype. We hope to enable autonomic network and virtualized resource testing by outlining how LXD containers and OVS can be combined with Salt to create an environment that is easy to test. The system attempts to reduce the amount of custom solutions by suggesting SaltStack as the unified operating level of autonomic elements. This proposal is to inspire the adoption of unified systems design, where research can be quickly integrated and tested in a shared environment.

CHAPTER 7: FUTURE WORK

In the future, we intend to continue extending and building upon the foundation that we created into a fully functioning prototype. Currently we have been utilizing terraform to build the entire environment so that other researchers can deploy their own local environment. Some examples of extending this work includes integrating SaltStack's Umbra, a machine learning pipeline, into our prototype to explore more intelligence signal's and reactions. This will also enable move robust research on smart-swaps and active learning contained in current literature. We also intend to explore provenance as it relates to SDN configurations to see how we can integrate granular network information into the umbra machine learning pipeline.

Throughout the development of this prototype, we realized that there was significant work to be done in the field of enriching test suites for autonomic computing environments. We hope to extend and release our testing suite for the use in future prototype and enforcing environments. Our initial focus will be on creating agents that are able to fuzz systems to try to find logic errors in self-optimization algorithms and reaction loops.

REFERENCES

[1] M. Burgess and O. College, “Computer Immunology,” p. 17, 1998.

[2] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, “Abstractions for Network Update,” p. 12.

[3] J. Kephart and D. Chess, “The vision of autonomic computing,” vol. 36, no. 1, pp. 41–50.

[4] J. O. Kephart, “Research challenges of autonomic computing,” p. 8.

[5] M. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. Carpenter, S. Jiang, and L. Ciavaglia, “Autonomic Networking: Definitions and Design Goals,” Tech. Rep. RFC7575, RFC Editor, June 2015.

[6] S. Yu, “How to solve cybersecurity: New paradigms for the next era of security,” 2019.

[7] R. Anthony, “A policy-definition language and prototype implementation library for policy-based autonomic systems,” in *2006 IEEE International Conference on Autonomic Computing*, pp. 265–276, IEEE.

[8] J. O. Kephart and J. Lenchner, “A symbiotic cognitive computing perspective on autonomic computing,” in *2015 IEEE International Conference on Autonomic Computing*, pp. 109–114, IEEE.

[9] S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart, “An architectural approach to autonomic computing,” in *International Conference on Autonomic Computing, 2004. Proceedings.*, pp. 2–9, IEEE.

[10] W. Han, Z. Zhao, A. DoupÃ©, and G.-J. Ahn, “HoneyMix: Toward SDN-based Intelligent Honeynet,” in *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization - SDN-NFV Security ’16*, (New Orleans, Louisiana, USA), pp. 1–6, ACM Press, 2016.

[11] S. Kim, “Cognitive Model-Based Autonomic Fault Management in SDN,” p. 103.

[12] A. Khalid, M. A. Haye, M. J. Khan, and S. Shamail, “Survey of frameworks, architectures and techniques in autonomic computing,” in *2009 Fifth International Conference on Autonomic and Autonomous Systems*, pp. 220–225, IEEE.

[13] G. Tesauro, D. M. Chess, W. E. Walsh, R. Das, A. Segal, I. Whalley, J. O. Kephart, and S. R. White, “A multi-agent systems approach to autonomic computing,” p. 8.

[14] D. Agrawal, Kang-Won Lee, and J. Lobo, “Policy-based management of networked computing systems,” vol. 43, no. 10, pp. 69–75.

[15] A. Yumerefendi, P. Shivam, D. Irwin, P. Gunda, L. Grit, A. Demberel, J. Chase, and S. Babu, "Towards an autonomic computing testbed," p. 5.

[16] S. Baik, Y. Lim, J. Kim, and Y. Lee, "Adaptive flow monitoring in SDN architecture," in *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, (Busan, South Korea), pp. 468–470, IEEE, Aug. 2015.

[17] K. Tsagkaris, M. Logothetis, V. Foteinos, G. Poulios, M. Michaloliakos, and P. Demestichas, "Customizable Autonomic Network Management: Integrating Autonomic Network Management and Software-Defined Networking," *IEEE Veh. Technol. Mag.*, vol. 10, pp. 61–68, Mar. 2015.

[18] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. K. Ramakrishnan, and T. Wood, "Virtual function placement and traffic steering in flexible and dynamic software defined networks," in *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*, (Beijing, China), pp. 1–6, IEEE, Apr. 2015.

[19] S. Lewandowski, D. Van Hook, G. O'Leary, J. Haines, and L. Rossey, "SARA: Survivable Autonomic Response Architecture," in *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, vol. 1, (Anaheim, CA, USA), pp. 77–88, IEEE Comput. Soc, 2001.