# Poly$^2$ Paradigm: A Secure Network Service Architecture *

Eric Bryant, James Early, Rajeev Gopalakrishna, Gregory Roth, Eugene H. Spafford,
Keith Watson, Paul Williams, Scott Yost
Center for Education and Research in Information Assurance and Security (CERIAS),
Purdue University
656 Oval Drive, Purdue University,
West Lafayette, IN 47907
{bryante, earlyjp, rgk, groth, spaf, kaw, pdwillia, syost}@cerias.purdue.edu

## Abstract

*General-purpose operating systems provide a rich computing environment both to the user and the attacker. The declining cost of hardware and the growing security concerns of software necessitate a revalidation of the many assumptions made in network service architectures. Enforcing sound design principles while retaining usability and flexibility is key to practical security. Poly$^2$ is an approach to build a hardened framework for network services from commodity hardware and software. Guided by well-known security design principles such as least common mechanism and economy of mechanism, and driven by goals such as psychological acceptability and immediate usability, Poly$^2$ provides a secure platform for network services. It also serves as a testbed for several security-related research areas such as intrusion detection, forensics, and high availability. This paper discusses the overall design and philosophy of Poly$^2$, presents an initial implementation, and outlines future work.*

## 1. Introduction and Philosophy

Poly$^2$ (short for poly-computer, poly-network) is a hardened framework in which the mission critical network services of an organization can operate. This framework is intended to provide robust protection against attacks to the services running within its domain. The design and implementation of Poly$^2$ is based on good design principles—principles developed in the field of computer security over the last 30 years [5] but are infrequently and often inconsistently applied in actual systems.

The basic operating tenet is that the use of a general-purpose operating system and the consolidation of network services on a single system (as is typical of modern networking environments) often leads to compromised services and systems. Vulnerabilities in the operating system or a specific network service allow attackers to subvert the entire system or to disable or modify other network services running on the same machine. Compromised systems are also used to attack other systems, to breach systems internal to an organization, to gather sensitive information, and to covertly monitor organizational activities. Furthermore, these general purpose systems are difficult to protect—detecting anomalous behavior in systems rich with services is a difficult task [19].

The Poly$^2$ approach is to separate network services onto different systems, to use application-specific (minimized) operating systems, and to isolate specific types of network traffic. Trust in the entire architecture comes from the separation of untrusted systems and services. The separation of network services helps contain successful attacks against individual systems and services. Therefore no single compromised system can bring down the entire architecture. The minimized operating systems only provide the services required by a specific network service. Removal of all other services reduces the functionality of the system to a bare minimum. Specific types of network traffic such as administrative, security-specific, and application-specific traffic are isolated onto special sub-networks. Because the nature of the traffic on each sub-network is specific and known in advance, deviations in normal traffic patterns are more easily detected [1] [19].

This effort builds on ongoing work at CERIAS. A preliminary design was created using good security design principles. This paper covers the Poly$^2$ initial philosophy and ongoing work, with particular emphasis on the overall architectural design and initial implementation.

* We acknowledge some initial inspiration to explore this architecture from conversations with Peter Neumann and William Hugh Murray. The initial design of Poly$^2$ has been funded by a gift from the Intel Corporation, and by sponsors of CERIAS.

Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)
1063-9527/03 $17.00 © 2003 **IEEE**

## 2. Challenging Conventional Wisdom

A trend in modern information technology (IT) architectures is server consolidation. In many cases, this is an attempt to reduce the overall costs of owning and operating IT systems by consolidating services spread throughout an organization into centralized locations, and run them on a single, large computer system, or onto clusters of servers all performing related or similar tasks. In both cases, however, these services are often deployed on systems running a general-purpose, full-featured operating system. Such systems are designed to be simple to install and use, and require little time to configure. Subsequently, they provide a large number of services, many of which are not necessary or used, except by attackers.

This paradigm is especially troublesome in the context of security. The consolidation of many network services on a single system is problematic if one of the services is compromised. If an attacker gains system-level access through a vulnerable service, she will likely disable, modify, or in some way compromise the other network services or damage the entire system. Additionally, general-purpose operating systems often have unnecessary services running by default. Often these services are overlooked or assumed to be required for stable operation of the system. Vulnerabilities in these services may provide an attacker with unauthorized ways into the system. A general-purpose operating system might also have libraries and utilities (such as compilers, shells, and file-transfer programs) that are not required by the critical network services deployed on a system or for the maintenance of the system. These provide an attacker, who has subverted the system, with a full set of tools that could be used to create and launch attacks against other systems.

While recent attempts have enhanced security by emphasizing the disabling of unneeded services [7] [12], determining which services are necessary and which are not is a difficult task. Thus, services that can be disabled are overlooked, assumed to be required for stable operation, or left active for future needs. To further complicate the issue, there are many capabilities and services built into modern operating systems that are very difficult or impossible to disable using the available configuration interfaces.

The Poly$^2$ architecture is based on a simple philosophy: apply well-known design principles and supply hardware to support those principles. The resulting system is composed of simple components, exhibits good principles of separation and structure, and is easier to verify, operate, and maintain while being more secure.

### 2.1. Core Design Principles

Principled development is central to the Poly$^2$ project. The historic Saltzer and Schroeder principles [13] and Neumann's augmented and refined versions [11] of them as presented in Bishop's text [1] inspired our work.

Currently, the primary principles of focus are:

- Economy of Mechanism: Security mechanisms should be as simple as possible. Economy of mechanism is a key driver in Poly$^2$ design and development. Its application to the network services provided by the Poly$^2$ architecture eschews the "bundling" that is prevalent in today's software systems.
- Least Privilege: A subject should be given only those privileges needed to complete its task. Ensuring network services run with the least privileges possible is critical to the Poly$^2$ infrastructure.
- Separation of Privilege: A system should not grant permission based on a single condition. Security mechanisms in Poly$^2$ are separated such that the trust necessary to compromise the system is not placed into a single mechanism.
- Complete Mediation: All accesses to an object must be checked to ensure access is allowed. The principle of complete mediation is applied in layers of protection throughout the system by using sound authentication, authorization, and comprehensive accountability both locally to specific network services, and globally, protecting the internal systems from each other.
- Fail-Safe Defaults: Unless a subject is given explicit access to an object, access should be denied. The system as a whole initially embodies no trust relationships—the authorized movements of data and control through the system must be explicitly defined by the security policy and allowed by the security mechanisms.
- Least Common Mechanism: Mechanisms used to access resources should not be shared. Shared resources, such as operating system services and broadcast networks, may offer both vulnerabilities and springboards for attackers who have successfully compromised part of the system. Interference between Poly$^2$ components is minimized by isolating network services on individual computer hosts and partitioning the network traffic into physically-separate, type-specific networks.
- Open Design: Security of a mechanism should not depend on secrecy of the design or implementation. The combination of policy and mechanisms designed to support the principles explained above will provide sufficiently robust protection that the design of a Poly$^2$

system need not be protected in order to keep the system safe.

- Psychological Acceptability: Security mechanisms should not make the resource more difficult to access than if security mechanisms were not present. The tension between usability and security is a fundamental concern—a robust and secure system is of little value if it is too difficult or onerous to operate in real environments. Therefore Poly$^2$ is designed to balance security and usability.

## 3. High-Level Design

### 3.1. Physical Separation versus Logical Separation

Multiple users and multiple processes share CPU, disk, memory, and network resources, all of which are logically separated over time and space. The applications and the underlying operating system are required to enforce this separation. However, software tends to be more complex than hardware, and is therefore more vulnerable to flaws in design and implementation. Hardware is also more difficult to configure or tamper with remotely, requiring physical access in most cases. Therefore, enforcing physical separation using hardware allows more assurance than doing the same in software. Migrating security functionality to the hardware is desirable, but this has to be finely balanced with manageability and flexibility. Examples of the use of physical separation include stripping unnecessary services and applications from server hosts, isolation of services on separate machines, distinct physical networks for different traffic classes, the use of write-once and read-only media where applicable (inspired in part by [14]), and the use of one-way only network links where possible (inspired in part by the concept of a Data-Diode as discussed in [4]).

### 3.2. One Application — One Machine

Running a single application on a machine provides isolation of services, thus providing immunity from flaws in other applications. Additionally, because the underlying operating system need only support a single, specific application, the O/S can be tuned to best support that application, both in terms of performance and security. Examples of performance characteristics that can be tuned include scheduling algorithms and file systems. From a security standpoint, behavior of such a specialized system is simpler to specify and deviations from normal behavior are easier to detect [1][19].

### 3.3. Isolated Networks

Networks provide a communication medium for different types of information by using a variety of protocols. The possibility of monitoring, injecting, or replaying traffic exists when systems share a network channel. These issues are dealt with using different protocol mechanisms (such as addressing, time-stamping, and sequence numbers) and cryptographic techniques (such as message authentication codes and encryption) [6]. Again, though, these are software mechanisms that can be broken given sufficient time and resources (information theoretically) [2]. The Poly$^2$ solution to this problem is to use isolated networks for carrying different types of traffic with no traffic routed between the networks.

Types of information can be categorized based on various attributes. One such categorization is based on the intended purpose of the information. Poly$^2$ currently defines three categories of traffic—application-specific, maintenance/administrative-specific, and security-specific. These three types of traffic are defined in Poly$^2$ to be at different sensitivity levels with the security and administrative traffic being more critical to the functioning of the whole system than the application traffic. Using separate networks for carrying traffic of different sensitivities allows for better separation of concerns, reduces interference, and increases confidence in the authenticity of the information.

### 3.4. Operating Environment and Specific Network Types

The operating environment of Poly$^2$ is intended to be the network presence located outside the organization's primary firewall in what is known as a demilitarized zone (DMZ). The equipment is physically protected by isolating it from common users both inside and outside the organization. As discussed above, the network infrastructure is partitioned into several physically and logically separate components:

- Application Network: This is the public-facing and only publicly visible part of Poly$^2$. The system, while internally partitioned into functional sub-systems, presents a single interface to the external world. This satisfies the principle of least privilege in that external users do not need to know the internal organization of the system.

- Administration Network: This network is used to administer the servers and services in the system, enabling both control over and awareness of the health of Poly$^2$ systems.

- Security Network: This network is used exclusively for security-specific traffic such as intrusion detection and auditing.
- Internal Data: This network is used as a link from Application Servers to data not available in a DMZ or on a Poly$^2$ server (e.g. databases, web server, etc.).

The separate networks limit sharing of information between components of the system which satisfies the principle of least common mechanism. The data stored on and moving throughout the system is also categorized as Application, Administration, Security, and Internal Data.

### 3.5. Security Policy Development

The security policy defines how information can and cannot move about the Poly$^2$ system. It defines requirements, not mechanisms. The requirements for supporting the policy are defined during system design.

**3.5.1. System Users** The three user groups of Poly$^2$ are:

- Administrators: Personnel responsible for configuring and maintaining the deployed services.
- Internal users: Internal organization users that need access to services located in the Poly$^2$ domain.
- External users: Users outside the organization that access the deployed services.

**3.5.2. High-level Security Policy** Poly$^2$ is intended to offer protection against unauthorized use of organizational resources by external users. The following requirements have been established:

1. Each application server component provides a single service to external users. All network traffic related to that service goes directly from the gateway to that server, traverses only the application network, and is not visible to any other service. However, outside users see only a single address for the entire Poly$^2$ system.
2. Each service runs at the lowest possible security level. Additionally, only the internal services and applications required to provide the service are available to processes running at the service's security level. This means that a compromise of a server application will not present to the attacker with a full suite of applications and services from which to launch further attacks.
3. The various networks are separated, both logically and physically.
   (a) Information cannot flow from one network to another without the intervention of a specific trusted and controlled process. Specifically, traffic coming in on the Application Network, which may contain commands from an attacker, cannot flow onto any other network. Nor can a user who has entered the system via the Application Network, compromised a service running on a server, and has that server's privileges, see or modify traffic on any other network.
   (b) Information can only move from one network to another through the actions of security or administrative mechanisms. In these specific cases, the information is moved and handled in a way that will prevent any malicious code from being executed. Any malicious code will not be entered into trusted data stores, but may be used as part of security, forensics, or administration tasks.
4. No traffic is to propagate from one Application Server to another Application Server inside the system.
   (a) All communications on the Application Network are initiated from the outside, and involve only a single service on a single host. An exception to this is a mail server, which initiates outgoing connections.
   (b) All communications on the administration network are initiated by the Administration Server. Each communication only involves the Administration Server and a single Application Server.
   (c) Communications on the Security network can be initiated by either the Application Server or the Security Server. It must flow only from the Application Server to the Security Server, or from the Security Server to the Application Server.
   (d) Communications on the internal data server are initiated only by the application server. Further, any interactions between the application and internal data sources must be explicitly defined, both in terms of connections, and content.
5. All communications inside the Poly$^2$ system must be authenticated and encrypted.
6. A compromised application server should not be able to flood any internal network to the point that other services are disabled or become unstable.
7. External users can only access Application Servers, must initiate the connection to the Application Server, and must connect only through the gateway. External users cannot log onto or access any services on the security, administrative, or internal data servers.
8. Administrative users can access Application Servers only through the Administration Servers. They cannot directly log onto or access any services on an Application Server through any network without going through the Administration or Security Server. Any action which modifies the security policy or enforcement

mechanisms must be done through a trusted administrative account which requires physical logon to the server hardware and two person integrity controls.

### 3.6. Specialized Operating System Through Minimization

An operating system may be viewed as an organized collection of software components that controls access to the hardware resources and provides an environment for the execution of programs. The control routines and utilities supported by an operating system are to a great extent dependent on the needs of the target environment. The ubiquitous use of computing systems in a variety of environments has given rise to general-purpose operating systems that typically combine support for all the target environments into one assorted collection of software artifacts. Such general-purpose operating systems provide more functionality than is needed in a simplified server architecture such as Poly$^2$. This unnecessary code poses a security risk.

The initial research in Poly$^2$ incorporates commodity software, (modern operating systems based on the 4.4BSD kernel [8] and common UNIX applications) for reasons of immediate usability, psychological acceptability, and experimental repeatability. The goal is to minimize a general-purpose operating system such that it supports only those specific services that are supposed to run on a system, thus eliminating the threat from vulnerabilities in unnecessary subsystems of the operating system that can be removed. The process of determining what layers and levels of functionality to remove is one of the primary research areas. While there are many areas in which unnecessary functionality can be removed, the following highlights two simple candidates: network stack and file system stripping.

**3.6.1. Network Stack Stripping** The network communication subsystem of the 4.4BSD kernel that includes network protocols and generic network utilities constitutes nearly 30 percent of the kernel (measured in lines of code) [8]. The protocol suite includes internet protocols, ISO protocols, X.25 protocols, and XNS protocols. Systems with well-defined structure and functionality operating in a very specific environment such as Poly$^2$ would need only a small subset of these protocols. A combination of ARP, IP, TCP, UDP, ICMP, and DNS is sufficient to provide the necessary network communication support. Retaining only these protocols will reduce the kernel size by more than 20 percent.

Certain protocol features can also be pruned depending on the functioning environment. Consider the fragmentation and reassembly component of the IP protocol. Because fragmentation and reassembly can be done at the gateway to Poly$^2$, and the maximum transmission units (MTU) of all the networks within the framework is defined in advance, no internal components need this functionality on the application network. Reassembly algorithms are complex and their faulty implementations have led to several vulnerabilities including Ping o' Death (CVE-1999-0128), Teardrop (CAN-1999-0015), Jolt2 (CVE-2000-0305), and Bonk (CAN-1999-0258) [9]. Fragmentation capability on a system can help exploit such vulnerabilities in other systems. Eliminating these features on systems where they are unnecessary will prevent the perpetration and propagation of related attacks.

Additional areas in which protocol stripping will be useful include ARP and DNS. The physical addresses of all machines are known and do not need to be resolved dynamically.

**3.6.2. File System Stripping** The stability of applications and the content they deliver can also be leveraged to reduce extraneous functionality in the file system. Consider a web server application delivering static web pages. Such a system would not need the ability to create or delete files, directories, or links, meaning that the supporting kernel for this application system would not require the code to provide this functionality. This not only eliminates the vulnerabilities that might be present in kernel code but also eliminates functionality that could be used by an attacker to store tools or conceal activity.

A conventional technique for the web server application might be to use a file system mounted as read-only, relying on the kernel to enforce the read-only policy. If an attacker is able to take control of the system through elevated privileges then she will be able to re-mount the file system with write privileges, thus circumventing the policy. In contrast, the kernel operating in the Poly$^2$ architecture does not possess the *ability* to re-mount or otherwise alter the file system. The same attacker in this scenario would not be able alter the file system—even with elevated privileges. The removal of extraneous code in Poly$^2$ effectively isolates the attacker.

The mishandling of symbolic links has been, and continues to be, a source of vulnerabilities in many applications [9]. Vulnerabilities in Kerberos 4 (CAN-2001-0417), `mgetty` (CVE-2001-0141), and Perl (CVE-1999-1386) allow an attacker to delete arbitrary files. An attacker can elevate privileges through a symbolic link vulnerability in the FreeBSD `mount_union` command (CVE-1999-0963) and can cause a denial of service in many versions of the Linux kernel (CAN-2001-0907) via a series of deeply nested symbolic links. By removing the capability to create symbolic links when an application does not need them, this class of vulnerabilities is eliminated from Poly$^2$.

Approximately twelve thousand lines of code are used in the 4.4BSD kernel to manage vnodes and support file system operations [8]. Of these, roughly fourteen hundred lines relate specifically to the write system call. This does not include the various write permission checks that are made

in other portions of the file system code. If the application does not need write access, all of this code can be eliminated from the kernel without sacrificing application functionality. Further, any programming errors or vulnerabilities present in this kernel code are also removed.

All application systems in the Poly$^2$ architecture with different file system requirements can be similarly customized to deliver only the functionality necessary. The aggregate customized systems contain fewer code vulnerabilities than conventional systems built on general-purpose O/S platforms. Also, individual application environments become hostile to attackers because they do not provide the needed resources to propagate and conceal attacks.

## 4. Implementation

The initial implementation of the Poly$^2$ architecture focuses on network services and operating system minimization. These areas are discussed below in the context of mechanisms satisfying the security policy.

### 4.1. Network Service Architecture

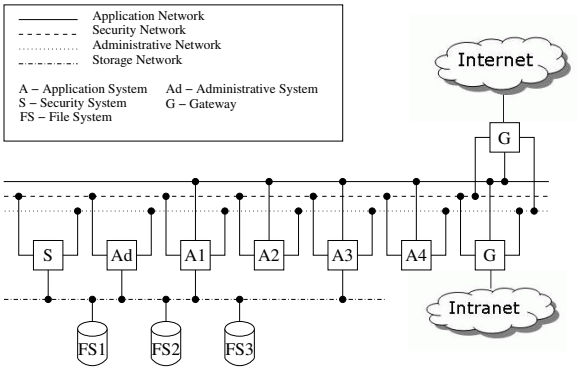


**Figure 1. Poly$^2$ architecture with a single application running on each system. The DMZ has four physically separate networks and separate Security and Administration systems**

Figure 1 illustrates the Poly$^2$ architecture. Each network application runs on an individual server with a minimal operating system and is connected to all the four networks. Additionally, there are two specialized systems: a *Security* system and *Administration* system. Both these systems are connected only to the Security and Administration networks.

The Security Server consolidates all the security-related tasks and can be host to a variety of security tools including intrusion detection systems and integrity checkers. Application Servers send all the security-related traffic (such as audit logs and alerts) to the Security Server on the Security Network. The Administration Server consolidates all activities related to maintaining, operating, upgrading, and patching the Application Servers. Traffic related to these tasks is limited to the Administration Network.

The directionality of communication (from the perspective of Application Servers) in the Poly$^2$ framework is logically represented in Figure 2 in a simplified form. Application Servers can send and receive traffic on the Application Network. This is how users from both the Internet and intranet access the network services. The Application Servers can only send traffic to the Security Server on the Security Network and can only receive traffic through their Administration Network interfaces from the Administration Server alone. The Security Server can only receive traffic from the Application Servers and Administration Server on the Security Network interface and can only receive traffic from the Administration Server on the Administration Network interface. The Administration Server can only send traffic to the Application Servers and Security Server on the Administration Network and only send security-related traffic to the Security Server on the Security Network.

This directionality is enforced using firewall rules on individual servers. While the long-term goal is to adopt technology similar to the Data-Diode, in the near-future separate firewall systems for each individual server in the framework will be used. This separation is necessary because the firewall rules enforced within a server cannot be trusted if the server is subverted.

### 4.2. Operating System Minimization Methodology

An application binary and its dependencies are the only essential components for an application to execute on a system. These dependencies include dynamically linked libraries, system calls, and any necessary subsystems of the kernel. All other code on the system is superfluous.

Five stages of operating system minimization are in process. The completion of each stage results in an increase in difficulty for an attacker to compromise the individual server. The work ranges from simple (utility and unneeded application removal) through difficult (the removal of unnecessary system calls and functionality). The completion of each stage is intended to increase the work factor of an attacker [15]. The early stages will force the attacker who has successfully exploited a vulnerability in a service application to import her own utilities to further compromise a server, whereas completion of the later stages may force the

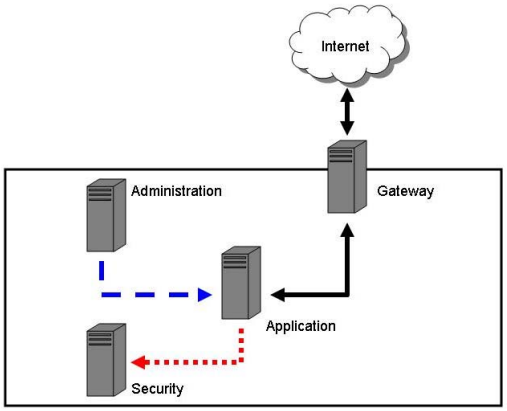


**Figure 2. Logical diagram of Poly$^2$ architecture indicating the directionality of information flow among the Security, Administration, and Application Servers**

attacker to obtain physical access to the server in order to compromise its security.

1. Services and Utilities: A variety of language tools (such as assemblers, compilers, interpreters, and debuggers), database utilities, desktop utilities (such as desktop environments, browsers, email, and chat clients), development tools (such as editors and shells), gaming tools, networking utilities (such as DHCP, DNS, FTP, SSH, and Telnet), and system tools (such as system configuration, monitoring, and security tools) are part of standard installations of most operating systems. None of these are required once an application is running and can be removed.

2. Libraries: A collection of libraries is part of many operating systems. The goal is to offer users access to standard implementations of functions required in various programming environments instead of burdening them with the development of their own versions. Unless the library dependencies of a program are statically linked, they are resolved at run-time. No libraries, except for those required by the server application, are needed and can be removed. For example, the Apache web server needs only three libraries: `libc`, `libcrypt`, and `libmm` at runtime in addition to the various modules that implement SSL, PHP, and other features. Most common applications can be statically linked and therefore, many, if not all, of the dynamic libraries can be removed. Since only single application is running on a server, the removal of shared code (from dynamic libraries) does not increase the disk storage required by each application server.

3. System Calls: System calls provide user applications access to routines in kernel space. Many system calls, however, have stubs or wrapper routines as part of a library. An operating system typically has over a hundred system calls (FreeBSD currently has 323 system calls), but applications require only a small subset of these. Removing the system calls not used by an application limits access to the kernel, eliminating access to functionality that may be exploited by an attack.

4. Library Functions: A library for a particular environment encapsulates several functions that might be required in that programming environment. For example, `libc` contains all the standard functions required while programming in the C language. Similarly, `libcrypt` contains encryption/decryption routines. An application might use only a small subset of all the functions in any given library. By retaining only those necessary functions in a library, vulnerabilities can be avoided in other parts of the library code as well as prevent attack code from using unneeded library routines.

5. Kernel Subsystems: The operating system kernel has numerous routines that perform process management, memory management, file-system management, interprocess communication, and I/O handling. A wide range of devices, file-systems, and protocols are supported to enable the operating system to be general-purpose. Removing those kernel subsystems that are of no use to the single application running on a server will not effect its operation. An attacker needing functionality not remaining in the minimized server will be forced to replace the O/S kernel—a task that may require physical access to the machine.

## 5. Quantitative Assessment of Poly$^2$'s Security Properties

A significant research contribution is an evaluation of the increased Poly$^2$ security compared to conventional network service architectures. To answer "why" and "how much", work is underway to define measurable contributions and benefits of Poly$^2$'s architecture. A quantitative assessment provides numerical feedback of the improvements and allows impact assessment of any proposed measure. Two vulnerability metrics and two hypothetical attack scenarios that can be used to understand and quantify the security properties of Poly$^2$ are discussed below.

### 5.1. Vulnerability Metrics

The first metric indicates the number of possible ways an unauthorized user can remotely gain superuser privileges. For purposes of this metric, only two attributes of vulnerabilities—their range (locally versus remotely exploitable) and the extent of privilege escalation they provide (user versus superuser) are considered.

An unauthorized user can remotely gain superuser privileges on a system in two ways: directly through remotely exploitable vulnerabilities that result in superuser access or by first targeting remotely exploitable vulnerabilities that provide user-level access and then exploiting local vulnerabilities that give superuser rights. Therefore, the number of ways by which an unauthorized user can exploit one application to obtain superuser privileges can be denoted by the equation:

$$VM_1 = N^1_{R->S} + (N^1_{R->U} * N^1_{L->S})$$

where $VM_1$=Vulnerability Metric for one application, $N^1$=Number of vulnerabilities in application one, R=Remotely exploitable, L=Locally exploitable, S=Gives Superuser privilege, and U=Gives User privilege.

If two applications running on the same server, the corresponding metric is given by the equation

$$\begin{aligned} VM_2 &= (N^1_{R->S} + N^2_{R->S}) \\ &+ (N^1_{R->U} * N^1_{L->S}) \\ &+ (N^2_{R->U} * N^2_{L->S}) \\ &+ (N^1_{R->U} * N^2_{L->S}) \\ &+ (N^2_{R->U} * N^1_{L->S}) \end{aligned} \quad (1)$$

which can be reduced to

$$\begin{aligned} VM_2 &= (N^1_{R->S} + N^2_{R->S}) \\ &+ (N^1_{R->U} + N^2_{R->U}) \\ &* (N^1_{L->S} + N^2_{L->S}) \end{aligned}$$

If the two applications are running on separate servers then the combined vulnerability metric for the two servers is:

$$\begin{aligned} VM^{separate}_2 &= (N^1_{R->S} + N^2_{R->S}) \\ &+ (N^1_{R->U} * N^1_{L->S}) \\ &+ (N^2_{R->U} * N^2_{L->S}) \end{aligned} \quad (2)$$

Because the fifth and sixth terms in the RHS of equation (1) are greater than or equal to zero, the vulnerability metric with applications running on separate systems is less than or equal to the metric with all applications running on the same system. Equations (1) and (2) are generalized to *n* applications as:

$$VM_n = \Sigma^n_{i=1} N^i_{R->S} + (\Sigma^n_{j=1} N^j_{R->U}) * (\Sigma^n_{k=1} N^k_{L->S})$$

and

$$VM^{separate}_n = \Sigma^n_{i=1} N^i_{R->S} + \Sigma^n_{j=1} (N^j_{R->U} * N^j_{L->S})$$

Comparing the above two equations, $VM^{separate}_n$ is less than or equal to $VM_n$, which shows that running applications on separate servers can reduce the number of attack paths. Reducing the numbers of attack paths is a clear benefit applications on separate servers instead of aggregating them on a single server. Although the benefit may seem obvious, arriving at metrics that can mathematically corroborate the beliefs and conjectures provides needed rigor in the area of information security.

A second metric is a measure of the Total Lines of Code (TLOC)—an indicator of program complexity. If one can empirically infer that every *n* lines of code has a certain number of faults on an average, then TLOC can be used as an approximate indicator of the number of faults in a program. Reducing the TLOC of a program will, therefore reduce the number of faults present (if new ones are not introduced in the reduction process). Hence, TLOC can be used as a simple metric to measure the effectiveness of operating system minimization.

### 5.2. Attack Scenarios

Vulnerabilities in software components exist and will continue to exist. Poly$^2$ reduces the number of vulnerabilities exploitable by attackers and isolates those that remain to minimize damage from a successful attack on the architecture as a whole. The attack scenarios described below provide a view of how different features of Poly$^2$ interact in achieving those goals.

Consider a software component running in the Poly$^2$ framework with a buffer overflow vulnerability resulting in superuser privileges. An exploit for such a vulnerability usually results in the attacker gaining a root shell providing a platform for further perpetration. However, Poly$^2$ servers do not have shells or other common `/bin` and `/sbin` utilities. This forces the attacker who wishes to use a shell to "inject" the shell code or binary onto the server—a task much more difficult in a minimized system without utilities such as linkers, loaders, compilers and file-transfer programs. Even if an attacker has the skill and motivation to successfully exploit such a vulnerability, the amount of effort involved will provide a larger window of opportunity to detect and react to such anomalous behavior.

Next, consider a situation where a server has been compromised by exploiting a vulnerability in a running application, giving the attacker complete control over the compromised server. In this situation, the Poly$^2$ architecture is designed to eliminate the ability of the attacker to penetrate other servers in the framework. No information can be sent out on the Administration Network interface because it is unidirectional in the opposite direction (enforced perhaps by a separate hardware firewall or a Data–Diode). Traffic

sent out on the Security Network can only reach the Security Server because other servers cannot receive traffic on this network. The only network interface on other servers that the attacker can send traffic to is the Application Network interface, and the only remote programs that can be reached through that network interface are the applications running on other minimized servers. Therefore, the only way to penetrate another system is by exploiting vulnerabilities in applications running on those systems. Per the security policy, mechanisms are in place to prevent denial of service through flooding (e.g. the network infrastructure is switched rather than broadcast, and the switches won't route traffic between internal hosts). Thus, the attacker's presence on a compromised internal server gives no more advantage than being on any other host on the Internet that can access those applications. In other words, the Poly$^2$ framework isolates attacks and avoids single points of failure.

## 6. Future Work

The implementation described is the initial work to achieve the broad goals of Poly$^2$. The prototype is designed to serve as a platform for future research and development in areas of intrusion detection, forensics, high availability computing, and others.

### 6.1. Intrusion and Misuse Detection

Intrusion and Misuse Detection (ID) is a rapidly developing field because most deployed computer systems are vulnerable to an ever increasing threat of attack. Among the factors that make intrusion detection in generalized networking environments difficult is the wide range of services and protocols that must be protected. By forcing the security system designer to cover a wider range of resources, the defensive assets are, in a sense, "stretched thinner" than they would be in the highly focused Poly$^2$ environment. The ability to concentrate defenses on the minimized operating systems and well defined interfaces between Poly$^2$ components allows the exploration of intrusion detection systems that are both efficient in terms of resource usage, and are more robust since the possible legitimate activities of the system are well defined, with aberrations or anomalies easier to detect.

### 6.2. Computer Forensics

Computer forensics focuses on the aftermath of a computer security incident. The purpose of computer forensics is the collection, preservation, analysis, and presentation of computer-related evidence. This evidence is used to determine exactly what happened and who was responsible in such a way that the results are useful in a legal proceeding. The Poly$^2$ architecture isolates security incidents with its modular design, limited functionality, and well-defined network pathways. Further research in this area will include studying the data flows within the networks and consolidating log files appropriately.

### 6.3. High Availability

High Availability (HA), defined as systems that are continuously operational for long periods of time, is becoming increasingly important as businesses become more dependent on computers for their operation. High availability typically includes knowledge of all of a system's failure modes, including networks and applications. High availability also requires that the recovery times for all known failures have a known upper bound. Several approaches exist for high availability. One option is to use single, fault-tolerant systems consisting of redundant components such as power supplies, RAID, environmental monitoring, fans, and network interface cards. Another solution involves the use of several units of non-redundant hardware arranged in a cluster so that each node in the cluster is able to take over from any failures of partner nodes. Research involving the combined benefits of these approaches, failure modes, and recovery procedures is desirable to maintain high availability for scientific and engineering applications.

## 7. Related Work

Others have conducted research similar to the Poly$^2$ philosophy. The differences and similarities of Poly$^2$ to these other approaches are discussed below.

1. Composable High-Assurance Trustworthy Systems (CHATS): The CHATS project at the SRI Computer Science laboratory [10], among other things, analyzes general security principles such as trustworthiness, assurance, and composability. Many of the security principles that guide the development of Poly$^2$ are similar to this work.
2. Solaris Operating Environment Minimization for Security: Work at Sun Microsystems [12] describes techniques for minimizing the Solaris operating system to run a specific family of applications. The technique involves installing the core *O/S*, then installing all appropriate patches, removing all unnecessary packages, and finally testing the resulting system. Some of the techniques described are similar to those proposed in Poly$^2$ for minimizing the operating system to specifically suit the desired applications. However, minimization in Poly$^2$ is at a finer granularity (such as libraries, library functions, and system calls) than at the level of packages.

3. Extremely Reliable Operating System (EROS): EROS [17] is an operating system with a focus on reliability and security. The primary goal of the EROS kernel is to strictly enforce system policies, whereas traditional operating system services, such as memory management, are left to the user application to implement. EROS is so radically different from more standard operating systems that porting modern applications such as Apache to it is extremely difficult. The Poly$^2$ architecture also focuses on reliability and security, but is better able to run common applications.

4. Network Appliances: Appliance servers are network-enabled devices explicitly designed to provide a single dedicated service such as web caching, email, firewall, or a predefined suite of services. These servers are often nonprogrammable, fully pre-configured, sealed systems that run on a variety of functionally optimized and/or streamlined operating systems and chip architectures. The application systems in Poly$^2$ address similar needs although from a security-focused perspective.

5. Hardened Operating Systems and Tools: Hardened operating systems such as TrustedBSD [18], Security-Enhanced Linux [16], Bastille Linux [7], and Immunix [3] typically add new security mechanisms, replace existing modules with more secure ones, and/or disable certain features to make the system more resistant to attacks. Although Poly$^2$ has similar goals, the means of attaining them are not through addition of more software and disabling of features, but by removing unnecessary software and functionality.

## 8. Conclusion

The underlying design philosophy of Poly$^2$ challenges some conventional wisdom. The initial implementation focuses on segregating applications and networks, and minimizing operating systems. Preliminary metrics for quantifying security properties of Poly$^2$ have been identified, and areas of future work have been laid out.

Two approaches are common to improving the security of an existing system. The first involves retrofitting security by patching known vulnerabilities and adding more controls. The second involves re-designing the entire system from scratch, with security built-in. Most current security mechanisms fall in the first category. The few approaches in the second category are yet to prove their utility in an application-rich environment. Poly$^2$ takes a middle ground approach to this problem—an ongoing effort to retrofit well-known security design principles into commodity systems without sacrificing usability in a network service environment.

## References


[1] Matt Bishop. *Computer Security, art and science*. Addison Wesley, San Francisco, CA, 2003.

[2] M. Blaze, W. Diffie, R. Rivest, B. Schneier, T. Shimonnora, E. Thompson, and M. Wiener. *Minimal Key Lengths for Symmetric Cyphers to provide Adequate Commercial Security*, 1996. www.crypto.com/papers/keylength.pdf.

[3] C. Cowan and C. Pu. Immunix: Survivability through specialization. In *Proceedings of the Information Survivability Workshop*, February 1997.

[4] Myong H. Kang and Ira S. Moskowitz. A pump for rapid, reliable, secure communication. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 119–129, 1993.

[5] Paul Karger and Roger Schell. Thirty years later: Lessons from the multics security evaluation. In *Proceedings of the 18th Annual Computer Security Applications Confference, Las Vegas, Nevada*, December 2002.

[6] S. Kent and R. Atkinson. *Security Architecture for the Internet Protocol, RFC 2401*, 1998. www.ietf.org/rfc/rfc2401.txt.

[7] Bastille Linux. *Bastille Linux*. www.bastille-linux.org.

[8] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Longman, Inc, Boston, 1996.

[9] MITRE. Common vulnerabilities and exposures. http://cve.mitre.org.

[10] P.G. Neumann. Architectural frameworks for composable survivability and security. www.csl.sri.com/users/neumann/chats.html.

[11] P.G. Neumann. Practical architectures for survivable systems and networks. Technical Report Phase two, Project 1688, SRI International, Menlo Park, California, June 2000.

[12] Alex Noordergraaf and Keith Watson. Solaris operating environment minimization for security: A simple, reproducible and secure application installation methodology. In *Sun BluePrints Online*, December 1999.

[13] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[14] SCARABS. 2 Heads Hard Disk Drive. www.scarabs.com.

[15] Gregg Schudel and Bradley J. Wood. Adversary work factor as a metric for information assurance. In *Proceedings of the New Security Paradigms Workshop, Cork, Ireland*, September 2000.

[16] SELinux. Security-Enhanced Linux. www.nsa.gov/selinux.

[17] Jonathan S. Shapiro and Norm Hardy. EROS: A principle-driven operating system from the ground up. *IEEE Software*, pages 26–33, January 2002.

[18] TrustedBSD. TrustedBSD. www.trustedbsd.org.

[19] Paul Williams, Kevin Anchor, John Bebo, Gregg Gunsch, and Gary Lamont. CDIS: Towards a Computer Immune System for Detecting Network Intrusions. In *Proceedings of the 4th International Symposium, Recent Advances in Intrusion Detection 2001*, pages 117–133, Berlin, 2001. Springer-Verlag.