The Cracker Patch Choice:
An Analysis of *Post Hoc* Security Techniques[1]

Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole
crispin@wirex.com, http://immunix.org/
WireX Communications, Inc.[2]

Abstract

It has long been known that security is easiest to achieve when it is designed in from the start. Unfortunately, it has also become evident that systems built with security as a priority are rarely selected for wide spread deployment, because most consumers choose features, convenience, and performance over security. Thus security officers are often denied the option of choosing a truly secure solution, and instead must choose among a variety of post hoc security adaptations. We classify security enhancing methods, and compare and contrast these methods in terms of their effectiveness vs. cost of deployment. Our analysis provides practitioners with a guide for when to develop and deploy various kinds of post hoc security adaptations.

1 Introduction

It has long been known that security is easiest to achieve when it is designed in from the start. Unfortunately, for a variety of reasons, systems seem to be chosen primarily on the basis of features, and occasionally performance, and rarely on the basis of security. The result is that secure systems are not common, common systems are not secure, and those desiring security must choose among a selection of techniques for *post hoc* security enhancement [2, 7, 17]. The basis for selecting a *post hoc* security enhancement may even place security as a secondary consideration, i.e. "whatever bothers the users the least."

In this paper, we consider the space of *post hoc* security enhancements and categorize them according to *what is adapted* (either the *interface* or the *implementation*) and *how it is adapted* (either *restricted* or *obscured*). The classification scheme suggests that some adaptations will have better cost:benefit ratios than others. Existing and proposed *post hoc* security enhancements, including some of our own unpublished techniques, are then placed into this classification scheme, and analyzed for cost:benefit ratios. These experiences tend to confirm the model's predictions on the relative merits of various kinds of *post hoc* security enhancements.

For this paper, the security problems we consider mediating with *post hoc* security methods are primarily implementation errors (e.g. buffer overflow vulnerabilities or bad logic) and occasionally design errors (e.g. dependence on reusable passwords). We characterize these problems as *security bugs* and refer to the class of *post hoc* security enhancements that address them as *security bug tolerance* [7].

1.This work supported in part by DARPA contracts F30602-96-1-0331 and N66001-00-C-8032.

2.At the inception of this work, Crispin Cowan, Calton Pu, and Jonathan Walpole were at the Oregon Graduate Institute, and Heather Hinton was at the Ryerson Polytechnic University. Crispin Cowan is now with WireX Communications, Inc., Calton Pu is now at the Georgia Institute of Technology, Heather Hinton is with Tivoli, and Jonathan Walpole remains with the Oregon Graduate Institute.

Table 1: Security Bug Tolerance Techniques

	Interface	Implementation
Restriction	• File system access controls • Firewalls • TCP Wrappers • Java sandbox • TCP SYN time out	• *Small* TCB & code removal, i.e. bastion hosts • Static Type checking • Dynamic Checking: array bounds checking, assertion checking, StackGuard, non-executable segments
Obfuscation	• Winnowing and Chaffing • Deception Toolkit	• Random TCP initial sequence number • Random code or data layout • Random QoS change

1.1 Background

The goal of the `Blinded` project is to enhance operating system survivability with respect to security bugs using *dynamic specialization* [5, 23, 29] to induce *diversity* in the operating system implementation. The *diversity* is intended to make exploits (security attacks) *non-portable* across systems, and also make software failures (bugs) *independent* across redundant nodes. The result should be a redundant cluster of computers, each in a different configuration, such that while an attacker could crack some of the nodes some of the time, they could not crack all of the nodes all of the time.

This method is a form of "security through obscurity." The usual reason for rejecting security through obscurity is that the protection is provided by a single "secret" (the obscured implementation) which is difficult to change, and protection is lost when the secret is discovered or disclosed. I.e. if an encryption scheme depends on a secret algorithm, then most of the security is lost when the algorithm is disclosed. This objection does not apply to *dynamic* obscurity, where the "obscurity" secret is repeatedly regenerated by dynamically reconfiguring each node, seeded with a sufficient source of entropy.

However, we *did* find it difficult to construct implementation specializations that were *effective* in defeating exploits. To be effective, a diversity technique must *break* some assumption that the attacker depends on, and yet *not* break any assumptions that legitimate applications depend on. Preferably, the attacker's broken assumptions should not be easy to accommodate, lest the attacker respond with adaptive exploits. In practice, we found it difficult to create implementation diversities that provided all these properties.

Without abandoning the dynamic diversity defense, we sought to expand our options by generalizing on the *adaptations* that can be applied to make software security bug tolerant. We categorize adaptations in terms of *what* is adapted, and *how* it is adapted:

What is Adapted: "What" is adapted is either a component's *interface* or its *implementation*. Naturally, what is "interface" and what is "implementation" is a relativistic view: an adaptation is an "interface" adaptation if it affects *other* components, and it is an "implementation" adaptation if the adaptation has no externally visible effects other than reduced vulnerability.

How it is Adapted: "How" a component is adapted to reduce vulnerability is either a *restriction* or an *obfuscation*. A "restriction" is an adaptation that characterizes some behaviors as "bad", and *a priori* prohibits or prevents those behaviors.An "obfuscation" radomizes some aspect of the component while obscuring the current configuration from would-be attackers. This makes it difficult for an attacker to deploy an attack(s), as the configuration details required for the attack cannot be reliability and repeatedly predicted by the attacker.

The two values for each of the two dimensions produce a quadrant of security bug tolerance adaptations. Table 1 shows this quadrant, populated with example security bug tolerance techniques.

Some cells in the quadrant are old ("well-understood") and thus heavily populated with known examples and techniques, while others are relatively new and unexplored. It is one benefit of our approach that we are

able to evaluate the worth of these new and unexplored techniques without having to learn through experience (the "hard" way).

The rest of this paper is organized as follows. Section 2 populates our model with example techniques. Section 3 describes the model's predictions for relative strengths and weaknesses. Section 4 evaluates these predictions by considering the strengths and weaknesses of actual security bug tolerance adaptations from each cell. Section 5 discusses the implications of our evaluation, providing a preference order for security bug tolerance techniques. Section 6 presents our conclusions.

2 Populating the Model

This section describes our model for security bug tolerance adaptations in more detail. In Section 1.1, we defined interface adaptations as those that affect external components, and implementation adaptations as those with no external visible effects. Section 2.1 and Section 2.2 describe interface and implementation restrictions, respectively, and Section 2.3 and Section 2.4 describe interface and implementation obfuscations, respectively.

2.1 Interface Restrictions

Interface restrictions are more classically known as access controls, which exist to selectively give principals access to objects. Access controls provide security bug tolerance in that the software on either side of the interface may have bugs, and the access control mechanism will restrict the set of operations that can be performed through the interface under various circumstances. Restrictions can be either *who* can perform an operation, e.g. Bob can and Alice can't, or *what* they can do, e.g. read but not write.

2.1.1 File System Access Controls

File system access controls specify which users and processes may perform what operations on files. The interface restriction is to prohibit certain operations by certain subjects on certain objects. The long history of file system access control schemes provides a rich variety of bases on which to make this restriction decision.

2.1.2 Firewalls

Firewalls control access from the *outside* of a LAN to the *inside* of the LAN. Access control decisions are made on a *per packet* basis: a packet arriving at the firewall is either accepted, returned to the sender, or dropped on the floor. The basis for the decision varies, depending on the firewall architecture: packet filters (which inspect only the packet) statefull packet filters (which consider previous packets as well) and application proxies (which consider packets in the application's context).

The interface restriction involved with a Firewall is that only allowed packets are passed through the firewall. For these packets, the firewall is "invisible." Packets that are not allowed through the firewall are dropped, an action that has an effect noticeable by the sending agent/component.

2.1.3 Wrappers

A wrapper is a program wrapped around a program suspected of having bugs. The wrapper takes input intended for the subject program and does various integrity checks on it. If the input passes muster, it is passed on to the subject program, otherwise it is rejected. The interface restriction is that, like firewalls, only approved input is passed on to the wrapped program.

TCP Wrappers [28] is an example, which acts like a small firewall on the host, restricting access to particular ports and services. Wrappers have also been applied to vulnerabilities in application programs. Many privileged programs are sloppily written, and thus vulnerable to "creative" input, such as large strings that induce buffer overflows or other errors. Wrappers have been developed that restrict the syntax of input to privileged programs to finite-length strings and "safe" character sets [1, 31].

2.1.4 Java Sandbox

The JVM imposes different restrictions on a Java program, depending on whether it was loaded from the local file system (an application that just happens to be written in Java) or loaded from the network via a

web server (an “applet”). The interface restriction is that downloaded applets are only given access to a restricted subset of the resources that Java applications can access.

2.1.5 TCP SYN Time Out

An attack that is difficult to defend against is the TCP SYN flood denial-of-service attack. Schuba et al [26] present a solution using a *dynamic* interface restriction that adjusts the time-out window for TCP connection requests. When the system feels that it is under attack via SYN flooding, the TCP time-out window is shortened, making it more difficult for the attacker to consume all of the TCP buffers. This also has the effect of making it more difficult for distant users to make connections, but it does protect *some* of the service by allowing nearby users to make connections.

2.1.6 Encryption

Encryption is a marginal case; it is either an interface restriction or an interface obfuscation, depending on one’s perspective. If one views an encryption *algorithm* as a black box that simply prevents 3rd parties from obtaining the clear text from the cipher text, then an encryption *protocol* (such as SSH, SSL, or IPSec) is an interface restriction that only allows parties with the proper keys to access the interface.

However, if one opens the encryption black box, then the transposition and substitution operations performed by encryption and decryption are actually *obscuring* the clear text. If one can obtain the proper key and algorithm, the clear text can be recovered.

For purposes of this paper, we will consider encryption to be an interface restriction. The rationalle is that we are discussing the security considerations for system & application software, and consider cryptography to be a separate field of study. This leads to the “black box” view of encryption, and thus the view that cryptographic protocols are interface restrictions.

2.2 Implementation Restrictions

Implementation restrictions are techniques to prevent programs from engaging in undesired behavior. Unlike interface restrictions, implementation restrictions have no visible effect outside of the adapted component. The following are example techniques for preventing applications from performing “bad” operations.

2.2.1 Code Minimization

Since the probability of bugs occurring proportionate to the size of the code [25] reducing the amount of software included in a component reduces the chance for potential vulnerabilities. This technique constitutes an implementation restriction in that any unnecessary but potentially vulnerable software is removed, *a priori* preventing the exploitation of those bugs. This is the basis for keeping TCB’s small, and also the basis for minimizing the functionality offered by a firewall machine.

2.2.2 Static Type Checking

Strong typing prevents a broad class of errors, but in particular allows use of the type system to protect a security monitor from the programs it monitors by preventing the program from converting arbitrary integers into pointers that point at the internal state of the security monitor. Strong typing constitutes an implementation restriction in that programs are prevented from performing arbitrary operations on typed objects.

Strong typing can be implemented in a number of ways, e.g. the Java type system is enforced both by the Java compiler and the Java bytecode verifier. Proof-carrying code [20] can be viewed as a special case of type-checking, where the code provided carries a proof that the code will not perform some class of “bad” operations.

2.2.3 Dynamic Checking

Dynamic checking, unlike static checking, detects “bad” operations on objects at run-time rather than compile or load time. The classic example is array bounds checking, which is not decidable at compile time, and so some run time checks must be used to ensure that array bounds are respected. The implementation restriction is that programs may not access array members outside of the bounds of the array, i.e. they may not treat arbitrary chunks of memory as array members.

StackGuard [8] presents a different form of run checking, where the integrity of the process state is inspected to ensure that a buffer overflow attack has not been used to corrupt the program's state. The implementation restriction is that programs may not dereference code pointers that show evidence of corruption.

Yet another approach to dynamic implementation restriction is marking some segments of the process's address space as *non-executable*. The general concept is to make the *data segment* of the victim program's address space *non-executable*, making it impossible for attackers to execute the code they inject into the victim program's input buffers. This is actually the way that many older computer systems were designed, but more recent UNIX and MS Windows systems have come to depend on the ability to emit dynamic code into program data segments. Thus one cannot make *all* program data segments non-executable without sacrificing substantial program compatibility. However, one *can* make the *stack segment* non-executable and preserve most program compatibility. Kernel patches are available for both Linux and Solaris [9, 10] that make the stack segment of the program's address space non-executable. Since virtually no legitimate programs have code in the stack segment, this causes few compatibility problems.

2.3 Interface Obfuscations

Interface obfuscation changes the *interface* to a component such that only one who knows the "secret" of the current configuration can successfully use the interface. Consider, for example, an object with three methods: "read", "write", and "execute". The canonical configuration for this object might be for "read" to be the first method in the object's method table, "write" the second method, and "execute" the third method. A simple interface obfuscation would randomly re-order the mapping from the method table to the actual methods, so that only a client that knows the current configuration can effectively use the object. There are relatively few instances of tools to systematically obscure interfaces. The only ones we know of that are effective are as follows:

2.3.1 Deception Toolkit

The *deception toolkit* (DTK) [3] provides tools to spoof the existence of service, e.g. a fake mail server. These faux servers area form of honeypot, intended to draw the attacker's attention away from the machine that is running an actual service. To hold the attacker's attention, the faux servers emulate vulnerable servers, producing results that are interesting to the attacker in response to common attacks against known vulnerabilities.

The DTK is an interface obfuscation with a small search space. The obscured component of the interface is the name of the machine running the actual server. This search space is problematic, because it is only as large as the number of machines in the defender's domain, compounded by the fact that it is often easy to discover the true location of many servers, such as mail and web servers. The DTK compensates for this small search space by substantially enhancing the "mystery" effect: the attacker sees what appears to be a forest of servers.

2.3.2 Chaffing & Winnowing

This novel cryptographic technique [24] was designed to subvert various legal restrictions on strong cryptography. Winnowing and chaffing does *not* alter the clear text data in any way; the clear text is sent *in the clear*. Instead, winnowing & chaffing *obscures* the clear text by surrounding it with a great deal of "noise" data. The "key" is an *authentication* protocol to select true data bits and "winnow" away the "chaff" bits.

Like encryption, chaffing & winnowing is a marginal case between interface restrictions and interface obfuscations. It nominally presents the same properties as encryption: technique in that both ends of the communication must be aware of the protocol, and they share a secret (the current key for selecting data from chaff). Thus chaffing & winnowing could be considered an interface restriction.

However, for legal purposes, chaffing and winnowing *emphasizes* the fact that the clear text is transmitted in the clear. In particular, chaffing & winnowing allows for *3rd parties* to do the chaffing for you, without sharing your keys. For this reason, we classify chaffing & winnowing as an interface obfuscation.

2.3.3 Capability Enforcement

Capability systems leave resources "in the open", accessible to anyone with the right "ticket" to access the resource. Most capability systems *enforce* the access control semantics of capabilities through interface re-

strictions that make the "tickets" *unforgeable*, i.e. the adversary cannot create a capability out of thin air. Some systems do this with tagged memory systems (e.g. AS/400) while others do this through strict type checking (e.g. Java).

An interface obfuscation version of capability enforcement is the large single address space model, where all resources reside in a single, very large address space, i.e. with 128 bit addresses. It is conjectured that the adversary will not be able to *find* resources unless given the correct address, since the address space will consist mostly of un-mapped pages that produce faults when probed.

2.4 Implementation Obfuscations

Implementation obfuscations do not remove vulnerability to bugs, but rather seek to make attacks that exploit implementation bugs *non-portable*, so that the attacker has to adapt the attack program to each configuration of a permutable implementation. By synthetically diversifying implementations in this way, one can employ physical redundancy (i.e. redundant servers) as a fault-tolerance technique against security attacks. In the same way that a *biodiverse* population survives a plague, a diverse population of implementations can have some members survive a single security attack.

The challenge of implementation obfuscation is to find a systematic way of permuting the implementation such that:

- The program continues to function as specified, and
- Hypothesized attacks against the program do *not* function as intended.

Again, there are very few successful techniques. The following techniques have succeeded in employing implementation obfuscation to enhance security.

2.4.1 Random TCP Initial Sequence Number

During the setup of a TCP connection, the TCP protocol negotiates an *initial sequence number*. If the initial sequence number is easy to guess, then the attacker can hijack the TCP session [14]. Early TCP implementations chose '1' as the initial sequence number. Newer TCP implementations try to choose an initial sequence number that is harder to guess.

At first glance, this might appear to be an *interface* obfuscation. However, because the selection of the initial sequence number is entirely one-sided (the server chooses the number, and then tells the client) it does not require any changes in the client's behavior. Thus since this technique does not require any client changes, we define it to be an implementation obfuscation (see Section 1.1).

2.4.2 Random Code Or Data Layout

This class of adaptations tries to break attacks that depend on specific properties of code or data layout in memory. Almost all attacks that depend on memory layout are buffer overflow attacks in which the attacker exploits weak array bounds checking (inherent to C programs) to enable corruption of adjacent program state. The implementation obfuscation is to randomize key properties of implementations that buffer overflow attacks depend on: adjacency and location.

Forrest et al [12] deployed a simple gcc enhancement that randomly permutes the size of stack frame activation records. Like StackGuard [8] and segment execution restrictions [9, 10] (see Section 2.2.3) this is designed to stop stack smashing attacks. Unlike the implementation restrictions, this technique tries to make the critical data structures in memory difficult to hit. Memco[3] has developed a similar product called STOP: Stack Overflow Protection [18] that allocates stack buffers in random locations.

2.4.3 QoS Change

This is a class of defenses against attacks that critically depend on timing to function. The notion is to randomly change the QoS (Quality of Service) delivered to different system components, so that the attacker cannot depend on the expected time the system will take to perform a given operation. Again, this technique obscures an aspect of the implementation that an attacker depends on: the expected time to perform a particular operation. This technique is most often deployed to defend against *timing covert channels*: the range

3.Nee Platinum Nee Computer Associates.

of randomness in the quality of service imposes an upper bound on the bandwidth of a timing channel, but does not close the channel [21].

2.4.4 *Natural* N-Version Programming

Koopman and DeVale [16] studied *natural* diversity in operating system implementation. While not a security study, this work examines operating system robustness with respect to failure modes by presenting erroneous input to various system call interfaces for each operating system. The study compares 13 diverse operating systems that all supported a POSIX interface. In this study, they tested each OS implementation for its correct response to various erroneous inputs. They found that *large* amounts of diversity are effective in enhancing robustness, in if one takes the best case behavior of all 13 tested operating systems, less than 5% of all failure conditions found were common to all 13 implementations. However, they also found that *low* degrees of diversity are relatively ineffective, in that if one chooses the two most diverse OS implementations possible (FreeBSD and AIX), then 9.7% of the failures are common to both OS implementations.

3 Model Predictions

This section considers what our model predicts in terms of the relative strengths and weaknesses of each cell in the quadrant in Table 1. Section 3.1 compares the two columns: interface vs. implementation adaptations. Section 3.2 compares the two rows: restrictions vs. obfuscations.

3.1 Interface vs. Implementation Adaptations

Interfaces are where accesses happen. As such, the interface semantics is what *specifies* our security policies. Interface adaptations act to improve the control over the operations subjects can perform on objects. These techniques might improve the abstraction or granularity of the access control specification (improving the defender's understanding of the configuration) or may act to obscure the specification so as to make attacks against configuration errors more difficult.

Implementations are what *enforce* the interface semantics. Implementation adaptations act to *bolster* the semantics of our interfaces. In particular, implementation adaptations provide protection against *failures* in access controls, especially due to errors in the implementation of the interface semantics.

Thus interface adaptations are the *primary* goal of a security enhancement. Implementation adaptations *complement* interfaces by acting to ensure that the interface's semantics will be enforced.

3.2 Restriction vs. Obscurity Adaptations

The dominant difference between restrictions and obfuscations is that restrictions reduce the amount of damage the attacker can impose, while obfuscations increase the cost of the attacker successfully exploiting a bug. Restrictions reduce potential damage either fractionally by disabling some operations, probabilisticly by disabling some principals from accessing the object in the hopes of containing the attacker, or completely by preventing *any* principal from performing operations that are always deemed to be "bad." Obfuscations restrict nothing, and only make it more work for the attacker to find the bug they're looking for so they can exploit it [25].

We also suspect that obfuscations are more difficult to make effective. An effective obfuscation must preserve many invariants in order to preserve interoperability with legitimate clients, while simultaneously breaking sufficient invariants to make attacks ineffective. Choosing effective invariants is made difficult by the fact that the invariants needed for legitimate interoperability are often subtle and implicit, and the invariants needed by attackers are completely unknown. We postulate that, given sufficient information to effectively deploy an obfuscation, that a more effective restriction could be deployed in its place:

Interface Obfuscations: Since interface obfuscations necessitate distribution of the current configuration to authorized clients, knowledge of the current configuration acts as an authentication token. An interface restriction using sufficiently strong cryptographic authentication is probably easier to deploy than an equivalent-strength interface obfuscation.[4]

4.If not necessarily easier to export; see "Chaffing & Winnowing" in Section 2.3.2.

Implementation Obfuscations: Implementation obfuscations must be faithful to the program's specification, or else they become implementation restrictions. Very few implementation vulnerabilities are independent of the program's specification[5], so finding effective implementation obfuscations depends on finding an obfuscation that works in the "gap" between the program's specification in the source code and the implementation in executable instructions, which is difficult. We postulate that restricting the implementation of the virtual machine that executes the instructions is more effective and easier than implementation obfuscation, e.g. the Java bytecode verifier, or StackGuard's restriction on corrupting activation records on running functions.

Thus one must be cautious when presented with a obfuscation security bug tolerance technique. What is the obscurity protecting, and how effective is it? For instance, the deception toolkit [3] is most effective at disguising a genuine service provider by surrounding it with faux service providers. However, it is not difficult for the attacker to identify the genuine service provider by other means, i.e. solicit an e-mail reply from a naive user and inspect the mail headers. Obscuring the size of stack frame activation records [12] successfully defeats simple stack smashing attacks because they depend on a static stack layout, but it is not difficult is it to construct stack smashing attacks that can adapt to a dynamic stack layout [8, 22].

Implementation obfuscations that are both transparent to the application programmer and effective in defeating or slowing attacks require a lot of information. We conjecture that if one has sufficient information for any given implementation obfuscation, that this information could be used to more easily deploy either an interface obfuscation, or an implementation restriction, which we suspect would be easier to implement, more effective, or both. For instance, while Forrest's compiler (an implementation obfuscation) makes certain stack smashing attacks more difficult to deploy, a similar implementation restriction in StackGuard (an implementation restriction) makes an identical attack impossible.

4 Evaluating the Model

Here we evaluate the predictions from Section 3 by directly comparing competing security bug tolerance techniques that reside in different cells in the quadrant, but that have similar goals. As in Section 3, we first compare interface vs. implementation techniques in Section 4.1, and then compare restriction vs. obscurity techniques in Section 4.2.

4.1 Interface vs. Implementation Techniques

The model does not make strong preferential predictions here. As discussed in Section 3.1, the interface must be sufficient to specify the operations that subjects may perform on objects. Interface adaptations that enhance the ability of the interface to specify who may do what to which are a primary means to enhancing security. In subtle contrast, implementation adaptations enhance the ability of implementations to *enforce* interface semantics. Thus interface and implementation techniques complement each other, as illustrated by these examples of using both:

Firewalls: Firewalls are a technique to enhance interface restrictions at the network level, and thus employ interface restrictions. Firewall employ implementation restrictions, primarily in the form of code minimization to minimize potential vulnerabilities in the firewall itself.

Java Security: Java security employs interface restrictions in the form of the "Java sandbox", which prevents mobile applets from accessing more than a trivial set of file system and network resources. These interface restrictions are enforced by the `security_monitor` object. Java security employs implementation restrictions in the form of static and dynamic type checking, to prevent hostile applets from corrupting the state of the `security_monitor` object.

Both of the above systems employ interface restrictions to enhance the expressiveness of the specification of "who" may do "what", and both employ implementation restrictions to ensure that the interface restrictions remain in force in the face of adversity. Thus interface and implementation adaptations seem to be complementary.

5.Because implementation vulnerabilities precisely exploit errors in the implementation, by definition they depend upon that implementation specification.

4.2 Restriction vs. Obscurity Techniques

Here we examine the relative strengths of restriction adaptations vs. obscurity adaptations. Section 4.2.1 compares interface restrictions vs. interface obfuscation, and Section 4.2.2 compares implementation restrictions vs. implementation obfuscations. In all of the cases we examined, we found restrictions to be the more effective technique, and in only a few cases did we find obfuscations adding any value beyond that provided by restrictions.

4.2.1 Interface Restriction vs. Interface Obfuscations

Here we compare competing techniques employing interface restrictions and interface obfuscations.

Morphing File System vs PACLs: The Morphing File System (MFS, part of the `Blinded` project) is a defense that we designed and implemented to protect sensitive files (e.g. /etc/passwd) from attack by *renaming* them to something obscure, and only "telling" the true name to programs that need to access these sensitive files. All other programs trying to find these sensitive files must search through a "forrest" of fake files, trying to determine which is the real one. MFS is an interface obfuscation in that the name of the object to be accessed has been obscured, and only clients knowing the true name can access the object.

While the MFS was "successful" in that it did make it hard for attacking programs to find the sensitive files, it was also complex and brittle. The MFS achieves *precisely* the semantics of Levitt's PACLs (Program Access Control Lists) [30]: disclosing the true name of an obscured file to a program is identical to adding the program to the ACL. PACLs are simpler to administer, and thus more likely to be administered correctly. PACLs can be implemented more simply, and thus are more likely to be implemented correctly. Thus for controlling program access to file system resources, interface restrictions seem to be more cost-effective than interface obfuscations.

Firewalls vs. DTK: Both firewalls and the DTK seek to protect vulnerable host systems from external network attack. Firewalls protect "inside" hosts by restricting the operations that external nodes may perform on internal nodes. DTK protects hosts by "chaffing" the LAN environment with numerous faux servers, making it more difficult for the attacker to locate the true server of a given service.

The problem with the DTK defense is that the search space (which machine is the real server?) is small, and the true server is easy to discover via other means (e.g. discover the mail server by inspecting mail headers). In contrast, the protection provided by firewalls is absolute: if the mail server is not to be accessed by outside hosts, then the firewall will not allow such packets to be delivered.

Chaffing & Winnowing vs. Encryption: While the chaffing & winnowing [24] interface obfuscation technique (see Section 2.3.2) does achieve effective confidentiality, it does so with very poor efficiency: the amount of raw data to be transmitted is very large relative to the payload. In contrast, the interface restriction technique of using classical encryption (see Section 2.1.6) achieves a similar degree of confidentiality with much greater efficiency.

Capability Enforcement: While the notion of enforcing non-forgeable capabilities using very large address spaces is intriguing (see Section 2.3.3) it has not yet been shown to be feasible. Memory-based capability enforcement imposes substantial implementation and performance overhead, but type-based enforcement seems to eliminate most of those costs. Obscure addresses in large address spaces would seem to impose performance costs due to the size of pointers, while not delivering any enforcement improvements relative to type- and memory-based enforcement mechanisms.

In all of these examples, we observe two patterns. First, in all cases the interface restriction techniques are *more cost effective* than the interface obfuscation techniques, in that restrictions either deliver more security for the same complexity (Firewalls vs. DTK) or deliver the same security with less complexity (MFS vs. PACLs and Chaffing & Winnowing vs. Classical Encryption). Second, the interface obfuscations *do* add security value relative to a system that has no corresponding adaptations applied.

Because *simplicity* is such a fundamental consideration in construction secure systems [25] it is important to choose the simplest means possible to achieve a given goal. Thus because interface restrictions seem to achieve the same results as interface obfuscations via simpler means, or greater security via similarly complex means, the suggestion for practitioners is to apply interface restrictions *first* to achieve the desired level

of security. Only if all practical interface restrictions have been applied and security is still not sufficient should interface obfuscations be considered.

For example, adding a firewall or strengthening the firewall's rules is likely more effective than deploying the DTK. But if the firewalls rules are already as strict and precise as possible, then deploying the DTK will further enhance security.

4.2.2 Implementation Restriction vs. Implementation Obfuscations

Here we compare competing techniques employing implementation restrictions and implementation obfuscations.

Memory Restrictions vs. Memory Obfuscations: Here we compare an assortment of memory access restriction techniques vs. memory layout obfuscation techniques. Memory access restriction techniques include generic techniques such as array bounds checking [15, 4] and debugging memory monitors [13] as well as security-specific techniques such as StackGuard [8] and non-executable segments [9, 10] (see Section 2.2.3) all of which act to restrict memory usage patterns that are considered "bad." Memory obfuscation techniques, in contrast, randomize some aspect of the layout of data or code in memory to make it difficult for buffer overflow attacks to accurately target critical pieces of program state (see Section 2.4.2).

The problem with memory obfuscation techniques is that attackers have developed *adaptive* attack methods that do not need to know the precise layout of memory [22, 11, 19, 27] limiting the effectiveness of this technique. For instance, an attacker does not need to know the precise offset of a buffer containing attack code; the attacker can prepend a string of NOP instructions in front of the attack code, and "lob" flow control into the midst of this field of NOPs.

In our own work, we investigated an enhancement to the StackGuard "terminator canary" mechanism [6]. A vulnerability arose, allowing attackers to undetectably corrupt stack frames protected with the "terminator" style of StackGuard protection. We investigated adding "jitter" to the position of the terminator canary, but found this to be difficult because the gcc compiler assumes a *fixed* offset between the stack frame and the local automatic variables. Use of StackGuard's "random canary" was found to be both more effective and more efficient.

Generalizing on this point, it is often said that the art of system design is the continuous insertion and removal of levels of indirection. Memory diversity defenses usually introduce an additional level of indirection for every object to be moved about. Memory obfuscation defenses that do *not* need to add a level of indirection often are ineffective, because it is precisely those techniques that attack programs can adapt to. Thus memory obfuscation will likely often introduce performance overhead and implementation complexity.

Cryptographic Authentication vs. Random TCP Initial Sequence Numbers: The random TCP initial sequence number implementation obfuscation technique described in Section 2.4.1 is effective at defeating some of the methods for hijacking TCP connections. However, an interface restriction technique of using cryptographically strong authentication and session encryption (see Section 2.1.6) is much more effective at stopping most kinds of TCP/IP session hijacking. The major advantage to the random initial sequence number is precisely that it is an *implementation* adaptation, and does not require modified client software, while cryptographic protocols require protocol-compliant peers or clients.

Random QoS Changes vs. Isolation: The implementation obfuscation technique of random QoS changes is effective at limiting the bandwidth timing covert channels, but does not eliminate them [21]. Extensive research into covert channel analysis has shown that it is very difficult to eliminate timing covert channels. Given the prevalence of cheap commodity computers, a brute force interface restriction approach of "put different security levels on different machines" seems to be easier and more effective than elaborate implementation obfuscations, which only limit covert channel bandwidth.

As in interface adaptations (see Section 4.2.1) we find that restrictions are more cost-effective than obfuscations. This finding is supported by Koopman and DeVale's finding [16] that natural diversity is of limited value in enhancing operating system robustness against simple error handling. However, the case is weaker than in interface adaptations: there are instances, such as hardening TCP, where implementation obscurity offers a benefit similar to an *interface* restriction. In a widely networked setting, backward software com-

patibility is at a premium, and so a technique that allows us to use an implementation adaptation rather than an interface adaptation may be much easier to deploy. Thus our recommendation to practitioners is to use implementation restrictions where possible, and to deploy implementation obfuscations in cases where *interface* restrictions would otherwise be required but are not viable.

5 Discussion

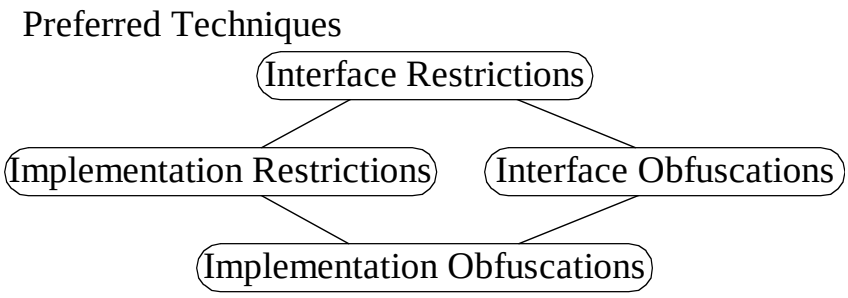

Figure 1 Partial Order of Security Bug Tolerance Techniques

Section 4.1 argues that interface adaptations and implementation adaptations are complementary, but that interface adaptations are more fundamental to security. Section 4.2 presents numerous examples of competing restriction and obfuscation adaptations that show that restrictions are more cost-effective than obfuscations. These two conclusions induce the partial order of security bug tolerance techniques shown in Figure 1.

These results may be controversial. On one hand, some may argue that lending *any* credibility to obscurity techniques is questionable. On the other hand, some might suggest that our preference for restrictions over obfuscations is too hasty. Here we seek to address both concerns.

Concerning the basic criticism of "security through obscurity" the various obfuscation techniques presented here clearly show that a carefully crafted obfuscation *does* add security value to a system. In some selected cases (Chaffing & Winnowing in Section 2.3.2, and random TCP initial sequence number in Section 2.4.1) obfuscation delivers a security property that is otherwise unobtainable for non-technical reasons (export control legality and backward compatibility, respectively). The crucial issue is that the obscurity must be *dynamic*, so that the security provided by the obscurity is not destroyed as soon as attackers learn of the obscurity technique.

Concerning the criticism that our preference for restrictions is hasty, we have several responses. First, it has been known for a long time that *simplicity* is critical to the implementation of secure systems [25]. All of the examples we have studied clearly show that restriction techniques are simpler to implement than obfuscation techniques, and thus are more likely to be implemented correctly.

Second, we want to emphasize that we are *not* claiming that obscurity techniques have *no* value, only that obscurity is less cost-effective than restriction. Obscurity advocates often employ analogies to *biodiversity* as a defense against disease in the organic world. We observe that in the organic world, *restrictions* such as skin, nose hair, eyelashes, mucus membranes, and antibodies form the *primary* defense against disease. Biodiversity is effective, but only at preventing complete catastrophe; restrictions also play an essential role in defending organisms against attack.

6 Conclusions

We have examined the difficult problem of *post hoc* enhancement of a system's security. We categorized *post hoc* security enhancements according to what they adapt (*interfaces* and *implementations*) and how they adapt it (*restrictions* and *obfuscations*). This model predicts a *preference order* of these techniques. Our model, populated with numerous example techniques of our own and from the literature, show that while *obfuscations* do add some security value, *restrictions* are more cost-effective than *obfuscations*. Our analysis provides a guide to security practitioners of when it is appropriate to develop and deploy a given kind of *post hoc* security adaptation.

References

[1] AUSCERT. `overflow_wrapper.c` – Wrap Programs to Prevent Command Line Argument Buffer Overrun Vulnerabilities. `ftp://ftp.auscert.org.au/pub/auscert/tools/overflow_wrapper`, May 1997.

[2] B. Blakley and D.M. Kienzle. Some Weaknesses of the TCB Model. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[3] Fred Cohen. The Deception Toolkit. `comp.risks` 19.62, March 1998. `http://all.net/dtk.html`.

[4] Compaq. `ccc` C Compiler for Linux. `http://www.unix.digital.com/linux/compaq_c/`, 1999.

[5] Charles Consel and Francois Noël. A General Approach to Run-time Specialization and its Application to C. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburgh Beach, FL, January 1996.

[6] Crispin Cowan, Tim Chen, Calton Pu, and Perry Wagle. StackGuard 1.1: Stack Smashing Protection for Shared Libraries. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998. Brief presentation and poster session.

[7] Crispin Cowan, Calton Pu, and Heather Hinton. Death, Taxes, and Imperfect Software: Surviving the Inevitable. In *Proceedings of the New Security Paradigms Workshop*, Charlottesville, VA, September 1998.

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Conference*, pages 63–77, San Antonio, TX, January 1998.

[9] "Solar Designer". Non-Executable User Stack. `http://www.openwall.com/linux/`.

[10] Casper Dik. Non-Executable Stack for Solaris. Posting to `comp.security.unix`, `http://x10.dejanews.com/getdoc.xp?AN=207344316&CONTEXT=890082637.1567359211&% hitnum=69&AH=1`, January 2 1997.

[11] "DilDog". The Tao of Windows Buffer Overflow. `http://www.cultdeadcow.com/cDc_files/cDc-351/`, April 1998.

[12] Stephanie Forrest, Anil Somayaji, and David. H. Ackley. Building Diverse Computer Systems . In *HotOS-VI*, May 1997.

[13] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, 1992. Also available at `http://www.rational.com/support/techpapers/fast_detection/`.

[14] L.T. Heberlein and M. Bishop. Attack Class: Address Spoofing. In *Proceedings of the 19th National Information Systems Security Conference*, pages 371–377, October 1996.

[15] Richard Jones and Paul Kelly. Bounds Checking for C. `http://www-ala.doc.ic.ac.uk/ phjk/BoundsChecking.html`, July 1995.

[16] Philip Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems (FTCS-29)*, Madison, WI, June 1999.

[17] Peter A. Loscocco, Stephen Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrel. The Inevitability of Failure: The Flawed Assumptions of Security in Modern Computing Environments. In *Proceedings of the 21st National Computer Security Conference*, Washington, DC, October 1998.

[18] Memco. STOP: Stack Overflow Protection. `http://www.platinum.com/products/sysman/security/sec_faq.htm`, 1999. A component of the Memco "Secured" product.

[19] "Mudge". How to Write Buffer Overflows. `http://l0pht.com/advisories/bufero.html`, 1997.

[20] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the USENIX 2nd Symposium on OS Design and Implementation (OSDI'96)*, 1996. Also available at `http://www.usenix.org/publications/library/proceedings/osdi96/necula.html`.

[21] Nick Ogurstov, Hilarie Orman, Richard Schroeppel, Sean O'Malley, and Oliver Spatscheck. Experimental Resoults of Covert Channel Elimination in One-Way Communication Systems. In *NDSS (Network and Distributed System Security)*, San Diego, CA, February 1997.

[22] "Aleph One". Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[23] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[24] Ronald L. Rivest. Chaffing and Winnowing: Confidentiality Without Encryption. *CryptoBytes (RSA Laboratories)*, 4(1):12–17, 1998. `http://theory.lcs.mit.edu/ rivest/chaffing-980701.txt`.

[25] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), November 1975.

[26] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, and Diego Zamboni.

Analysis of a Denial of Service Attack on TCP. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[27] Nathan P. Smith. Stack Smashing vulnerabilities in the UNIX Operating System. `http://millcomm.com/ nate/machines/security/stack-smashing/nate-buffer.ps`, 1997.

[28] Wietse Venema. TCP WRAPPER: Network Monitoring, Access Control, and Booby Traps. In *Proceedings of the Third Usenix UNIX Security Symposium*, pages 85–92, Baltimore, MD, September 1992. `ftp://ftp.win.tue.nl/pub/security/tcp_wrapper.ps.Z`.

[29] Eugen-Nicolae Volanschi, Gilles Muller, and Charles Consel. Safe Operating system Specialization: The RPC Case Study. In *Proceedings of the First Annual Workshop on Compiler Support for System Software*, Tuscon, AZ, February 1996.

[30] D.R. Wichers, D.M. Cook, R.A. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo. PACL's: An Access Control List Approach to Anti-viral Security. In *Proceedings of the 13th National Computer Security Conference*, pages 340–349, Washington, DC, October 1-4 1990.

[31] Joe Zbiciak. `wrapper.c` Generic Wrapper to Prevent Exploitation of `suid/sgid` Programs. Bugtraq mailing list, `http://geek-girl.com/bugtraq/`, May 19 1997. `http://cegt201.bradley.edu/ im14u2c/wrapper/`.