Death, Taxes, and Imperfect Software: Surviving the Inevitable[1]

Crispin Cowan and Calton Pu
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
and Heather Hinton
Electrical and Computer Engineering, Ryerson Polytechnic University
(crispin@cse.ogi.edu)
http://www.cse.ogi.edu/DISC/projects/immunix

Abstract

A security system is only as strong as its weakest link. This observation lead to security architectures that use a *small* trusted computing base (TCB) to minimize the number of "links" in the system. A small TCB both reduces the chance of a bug occurring by reducing the volume of software that may contain a bug, and also makes formal verification of the correctness of the TCB feasible. Unfortunately, for a variety of reasons, the commercial marketplace of popular operating systems has chosen to ignore this line of reasoning. The "trusted computing base" (system components embodied with significant amounts of trust) is not small, is not formally verified, and consequently is neither correct nor secure. We conclude that it is *inevitable* that commodity systems software will have flawed security.

Techniques have developed to allow systems to cope with potential security flaws, which we call *security bug tolerance*. Security bug tolerance enhances the survivability of a flawed system by *post hoc* dealing with the system's security flaws. This paper presents a categorization scheme for security bug tolerance techniques, and populates it with techniques of our own and from the literature. The categorization allows the reader to analyze various techniques to discover their similarities and differences, enabling the reader to compare relatively diverse tools on their merits.

1 Introduction

A security system is only as strong as its weakest link. The recommended "technique" for achieving strong security is to place all security-relevant functionality in a small, verifiable *trusted computing base* (TCB). Formal verification techniques are then used to assure that the TCB is worthy of trust. Unfortunately, a TCB that is small enough to verify is, in practice, too small to contain all

1.This work supported in part by DARPA grant F30602-96-0331.

the required security functions. Thus TCBs are continuously expanded with "necessary extensions". Conversely, a TCB that is large enough to be useful seems to be far too large to verify in practice, without enormous expense. This difficulty has had the unfortunate effect of causing commercial OS vendors to choose to *not use* TCB ideas, or at best pay lip service to them by deploying "trusted" systems that are too large to formally verify. This has generated enough controversy to be the subject of a formal debate at the 1997 IEEE Symposium on Security and Privacy [3, 15,21].

If formal verification is not feasible, developers that care *somewhat* about security strive to minimize security bugs with a variety of debugging and bug minimization techniques, such as strict coding practices, "red teaming," and fault-injection. Resorting to debugging techniques does not enhance security, but does reduce the cost of developing a system.

Unfortunately, while these techniques are cheaper than full formal verification, they are still expensive to deploy in terms of human resources. For reasons we detail in Section 1.1, the economic motives to produce secure, correct software are remarkably weak compared to other needs, such as rapid development, rich feature lists, and other marketing needs. Bug minimization and elimination techniques are only applied when safety is of paramount importance, and cost is a minor consideration, such as nuclear reactor control [20]. Thus debugging techniques will not be used *enough* to make systems actually secure, and it is likely that the wide-spread use of systems that have security bugs will be a persistent state for some time to come.

This paper proposes an alternative to holding our collective breath until vendors produce bug-free software: *security bug tolerance*. Security bug tolerance combines the techniques of security and fault tolerance to produce systems that *survive* attacks despite having security bugs. We introduce a classification of bug-tolerant techniques, describe examples of each technique, and discuss how these techniques are relevant to the securing of an actual system. In the remainder of this section, Section 1.1 elaborates on why bugs continue to happen. We describe the consequences of choosing to live with buggy software in Section 1.2, and Section 1.3 defines what we mean by "security bug tolerance".

1.1 Bugs Happen

Commercial software chronically has bugs, many with security vulnerability implications. Tempting as it may be to hypothesize that this is because the vendors are lazy or stupid, this is not the case. Commercial software chronically has bugs for the dual-reason that correctness is hard, and correctness does not sell software.

That correctness is difficult hardly needs re-stating; proving correctness is nearly intractable for non-toy programs, complete testing is similarly intractable, and even rigorous testing is expensive, especially given a constant stream of patches and updates. Developing software on "Internet time" aggravates this problem, by presenting vendors with a very tight schedule to meet, squeezing out time allocated for testing and auditing the impact of new code.

That correctness does not sell software may be somewhat less intuitive, especially in the security community where correctness is highly valued, but on a global scale, software customers appear to be highly insensitive to the correctness of the software that they're buying. New features seem to matter more than correctness. Consider the calls to Microsoft's support line [4,17, 25]:

- Most calls request help on how to do some particular task

- 5% of calls request some new feature
- Less than 1% of calls are to report bugs
- Discussion at the NSPW 1998 Workshop tend to confirm this view[2]

Advertisements, where software vendors promote their products, compare features, and occasionally speed, but rarely correctness. Even ads for *security* products rarely mention correctness[3]

Since correctness is very hard to achieve, and seems to offer little competitive advantage in the market place, it seems very unlikely that we will see correct software in large volume any time soon. As long as feature lists, time-to-market, and a little bit of performance matter more to customers than correctness, bug-ridden systems will be the *normal* state of affairs.

1.2 So You Have to Live with Bugs

Given that bugs are normal, users, and to some extent system architects, have developed techniques to survive the consequences. . Exactly which technique is used depends on the application domain:

Video Games: Just live with it. The game-playing experience is often more important than the machine state it produces, so the occasional failure is not very important. It is just "ok" for a game to crash from time to time, if it is rare enough to not cause excess frustration.[4]

Text Editors: Frequent state saves. Early computer users learned to manually save their work frequently, lest the computer or editor crash, taking the work with it. "Using a computer is like going to summer camp; write soon, write often." More advanced applications provide automatic periodic checkpointing, saving a temporary version of the document being edited so that it can be recovered if the application or the system crashes.

Operating Systems: System software such as the kernel and the window system ideally can operate indefinitely. Unfortunately, systems that have not emphasized correctness, such as some versions of Windows 95 and X Windows, suffer from memory leaks that causes their performance to degrade over time, and eventually crash. Such systems can be effectively "checkpointed" by periodically rebooting them, restoring them to a particular state that is relatively stable.

Hardware: Hardware fault tolerance is a different kind of a problem. Because of the difficulty in releasing "patches" to hardware instances, hardware vendors invest extensive resources in establishing the correctness of their products. But unlike software systems, *instances* of hardware systems are subject to occasional failure of the device. To protect against the failure of an instance, system architects may choose to *replicate* the instances. When an instance fails, a replacement instance is available to take over its workload.

These techniques apply well to "functionality" bugs. *Security* bugs present a different kind of problem, one that is not amenable to either ignoring the problem, checkpointing, or replication, because a potential attacker can exploit a security bug to induce damage far greater in scope than the failure of the buggy component:

2.Cristina Serban heard claims from SGI that are consistent with the calls to Microsoft's help line.

3.Marv Schafer speculates that there may be undesirable legal implications to acknowledging security faults.

4.Games are often about puzzle solving, so *excess* frustration might be difficult to detect :-)

Ignore it: Naturally, this technique is ineffective in defending against attack.

Checkpointing: Checkpointing can limit the damage an attacker can impose, but it is not completely effective. First, checkpointing fails to prevent the attacker gaining access to the system and stealing secrets. Second, an attacker may be stealthy enough to go unnoticed, causing data corrupted by the attacker to be propagated into the backup systems, and possibly eventually pushing all uncontaminated data out of the backup system. Third, an attacker that obtains a high degree of privilege on the system can also obtain control of the backup copies of the system, maliciously corrupting their contents.

Replication: Replication is effective against failures when the replicas are *independent* of the failure in question. A security bug is a property of the software, so replicating the software replicates the bug. All of the instances of a set of replicated systems will have *the same* set of vulnerabilities induced by the same set of security bugs. To be effective against security attacks, replicas must be independent of the security bugs they seek to protect.

Security bugs have the property that the potential damage is far out of proportion to the scope of the bug. Functionality bugs at worst affect only the data that they manipulate. A security bug, in contrast, can allow the attacker to take control of a host, or even an entire domain, allowing the attacker to corrupt all data on those machines. Thus security bugs can affect all of the data within the protection domain of the afflicted software.

Thus specialized techniques are needed to deal with security bugs that induce vulnerabilities. We call this *security bug tolerance.*

1.3 Security Bug Tolerance

Fault tolerance techniques either limit the damage that a fault can cause (checkpointing) or provide additional resources to mask component failures using other redundant components. Security bug tolerance can use the same techniques, but the containment and replication must occur on the same level as the security bugs, i.e applied to the bugs themselves. If the fault is in the software, then replicating the software will be ineffective because it will replicate the bug. If data corruption goes undetected, then checkpointing the data will succeed only in checkpointing corrupted data.

This paper presents a categorization of security bug tolerance techniques. A *security bug* is a flaw which permits any user to violate the security policy the system claims to enforce[5]. Security bug tolerance enhancse the *survivability* [23] of the system despite having the security policy violated by enhancing the integrity of the system itself. We propose that "survivability" is, in part, the study of how to effectively retrofit security to a system that was not designed with security as a goal. While we recognize that retrofitting security is not the first choice to achieve security, it seems to be the *only* choice for users of most systems, because most vendors do not view security as a high priority.

Section 2 presents the security bug tolerance categorization scheme, populating it with security bug tolerance techniques of our own, and from the literature. Section 3 and Section 4 describe these security bug tolerance techniques in detail. Section 5 discusses the implications of security bug tolerance, including how to effectively compare various security bug tolerance techniques. Section 6 presents our conclusions.

5.Definition due to Marv Shafer.

2 Security Bug Tolerance Techniques

Many techniques have emerged to contain damage induced by software bugs. We focus our attention on techniques that can be applied to *existing* software, and that assist in *tolerating* the existence of bugs in that software. We specifically exclude debugging techniques for *detecting* bugs; while debugging techniques have great merit, we argued in Section 1 that debugging techniques will not be applied enough to eliminate all bugs, and some security bugs will likely remain in released software. Furthermore, debugging techniques cannot detect security flaws inherent in a design.

Since security bug tolerance techniques have to deal with the problems of existing software, we view them as *adaptation* techniques. We then classify security bug tolerance techniques along two dimensions: *what* is adapted, and *how* it is adapted. "What" is either the program's interface or its implementation, and "how" is either a permutation or a restriction of the adapted piece of software, as shown in Table 1, along with examples of each technique.

Table 1: Security Bug Tolerance Techniques

	Interface	**Implementation**
Restrictions	• Firewalls • TCP Wrappers • Java security model • TCP SYN time out adjustment • Dynamically remove `.rhost` capability	• Firewalls • *Small* TCB • Dynamic type checking, array bounds checking • Stackguard defense against buffer overflow
Permutation	• Cryptographic session keys • Deception Tool Kit	• Random code or data layout • QoS change

We distinguish between interface and implementation adaptations because they affect inter-operability. Interface adaptations change the interface presented, so some or all of the software components or users that need to use the adapted software may also need to be adapted to continue interoperation. Implementation adaptations, however, do not affect the interface presented, and so normal interoperation with legitimate clients should continue.

We distinguish between restrictions and permutations to classify the technique used to contain the potential damage posed by security bugs. A restriction is an "architectural" technique; the software will no longer do that particular kind of operation, no matter who you are or what you know. A permutation is a re-arrangement of the interface or the implementation that hides or re-arranges flaws in the program, making it more difficult for an attacker to find and exploit the flaws. Section 3 describes interface and implementation restrictions, expanding on the concepts and providing examples. Section 4 does the same for interface and implementation permutations.

3 Interface and Implementation Restrictions

A restriction is some mechanism to prevent a buggy program from performing some particular class of operations. Any software bugs that would manifest themselves by trying one of these operations is thus prevented, containing the damage.

3.1 Interface Restrictions: Access Control

Interfaces exist to give principals access to objects. The security problem is that all principals should not have access to all objects. An interface restriction is thus a form of access control. 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.

Interface restrictions are further classified as *static* and *dynamic* interface restrictions. A static interface restriction is precisely a classic access control mechanism: a restriction on the precise circumstances under which some set of principals may access some set of objects. A dynamic interface restriction *change* who may access what, depending on the degree of threat perceived by the security system. Section 3.1.1 describes several access control methods that illustrate static interface restrictions, and Section 3.1.2 describes some examples of dynamic interface restrictions.

3.1.1 Static Interface Restrictions

Here we describe several forms of static interface restriction:file system access controls, firewalls, wrappers, and the Java security model. Table 2 summarizes these interface restrictions, describing the interface affected, and how it is restricted, which we detail here.

There are *many* forms of file system access controls, e.g. classic UNIX **rwx** mode bits, access control lists, etc. The file system is the repository for persistent data, so it is both important to control access, and complex to specify who has access to what give the number of principals and objects.

Firewalls provide a powerful form of access control based on interface restriction and network topology. The firewall effectively restricts the interface between machines on the "outside" Internet and machines on the "inside" protected network to a particular set of ports and protocols, as shown in Figure 1. All other attempts at access are blocked by the firewall. Firewalls are effective in protecting weak configurations on many hosts in an entire subnet, but they do so at the cost of coarse granularity in the access control specification. Firewalls can also be used to provide protection in-

Table 2: Static Interface Restrictions

System	Interface	Restriction
File System Access Controls	Access to files	Only specified users can access specific files
Firewalls	Access to machines from outside the physical domain	Outside machines can only access particular ports, via particular network protocols
Wrappers: TCP	Access to ports on a machine	Remote hosts can only access particular ports, via particular network protocols
Wrappers: `SUID` programs	Invoking a privileged program	Users can only invoke a privileged program with a limited argument syntax
Java: Type Safety	Access to a variable	Programs must treat their data values as consistent types
Java: Bytecode Verifier	Access to a variable	Programs must treat their data values as consistent types
Java: Sandbox	Access to files and network hosts	Programs loaded from the network may not access local files, and may only access the network host they were loaded from

ternal to a domain, but this depends on appropriate layout of the domain's LAN, because firewalls depend on physically restricting access to network connectivity.

Wrappers are another form of interface restriction. 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. TCP Wrappers [26] 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 [2, 27].

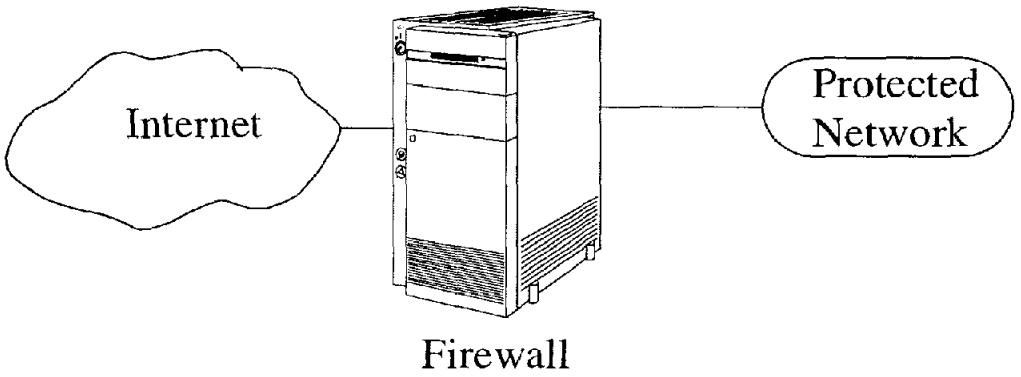

Figure 1 Firewall: Restricts interface from Internet to protected network to selected ports and protocols

Java's security depends on a "three-pronged" security model: the Java compiler's type safety, the Java bytecode verifier, and the JVM "sandbox." Each of these elements can be viewed as an interface restriction, as follows:

Type Safety: Type safety distinguishes between programs that have a consistent view of the types of their data values, and those that do not, e.g. it is inconsistent to view a single value as both an integer and as a memory reference. The type checking component of the Java compiler enforces an interface restriction by only compiling programs that are type safe. Proof-carrying code [18] is a special case of type-checking, where the code provided carries a proof that the code will not perform some class of "bad" operations.

Bytecode Verifier: The bytecode verifier enforces similar type safety restrictions to the Java compiler, and thus provides a similar interface restriction. The difference is that it does so at a later stage in the program's life, just prior to execution instead of during compilation.

JVM 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"). Applets are given access to a restricted subset of the resources that Java applications can access.

Whether these are interface or implementation restrictions is essentially a relativistic view of what constitutes the "normal" interface. If the "normal" interface does not have these restrictions (say, compared with the interface provided to C programs), then these are interface restrictions. However, if these restrictions are considered a "normal" part of the interface, then these are actually implementation restrictions that ensure that Java programs conform to the specified interface.

3.1.2 Dynamic Interface Restrictions

Dynamic interface restrictions can balance the convenience of a loose access control policy with the security of a tighter access control policy. Some interfaces provide facilities that are convenient, but not essential. When an attack is suspected (perhaps as indicated by an Intrusion Detection System) then "merely convenient" interfaces can be disabled. If vulnerabilities due to bugs are randomly distributed throughout the system, dynamically restricting the interface will reduce the probability that the attacker can actually reach the buggy component that they seek to exploit. Another example of a dynamic restriction is the BLP "*-property," which induces dynamic policy changes.

A simplistic example of a dynamic interface restriction is the use of `.rhost` files for access control. This form of access control is convenient for users, but highly insecure. A system supporting dynamic interface restriction could disable the honoring of `.rhost` specifications when the local intrusion detection system suspects intruders are present in the domain.

Schuba et al [22] present a more reasonable example of dynamic interface restriction. This work defends against the TCP SYN flood denial-of-service attack by dynamically adjusting the time-out window for TCP connection requests. When the system feels that it is under attack via SYN flooding, the 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.

An *Intrusion detection system* (IDS) *virtually* restricts interfaces by characterizing interface usage patterns as "legitimate" and "suspicious", raising alarms for suspicious behaviors, but not neces-

sarily taking any other pro-active actions. An IDS forms an important part of a dynamic interface restriction by providing a sophisticated notion of when the interface should be restricted. Intrusion detection systems come in a variety of forms [9, 24, 13, 14] but all share the basic property that they examine interface usage, and indicate suspicious usage, providing dynamically restrictable interfaces with an indication of when to be more restrictive.

3.2 Implementation Restrictions

Like interface restrictions, implementation restrictions contain the potential damage that can be inflicted by attackers exploiting security bugs. Unlike interface restrictions, implementation restrictions do *not* affect the explicit interface offered by the adapted component. Implementation restrictions take two forms. Section 3.2.1 describes *added code* restrictions, which do additional run-time checks to ensure correct behavior. Section 3.2.2 describes *removed code* restrictions, where functionality that is convenient but not essential is removed to reduce the probability of bugs occurring.

3.2.1 Added Code: Double-checking Correctness

Code added to restrict an implementation effectively double-checks the correctness of the implementation. Some notion of "correctness" is used to specify the correct behavior of the program, and code is injected to verify that the program conforms to this specification. The goal of this double-checking is to make the potentially buggy software *fail-stop* with respect to security faults: when an attacker attempts to exploit a vulnerability in a broken program, the program should fail and raise a security alert, rather than granting access to the attacker. If fail-stop security software can be achieved, it reduces the problem of security to the somewhat simpler problem of fault-tolerance.

The particular form of "correctness", and how it is specified, varies depending on the implementation restriction. Table 3 summarizes some added-code implementation restrictions, describing what additional correctness checks are provided by the added code in each case.

Table 3: Restricting Implementations with Added Code

System	Additional Check
Purify	Run-time verification of the correctness of memory references, e.g. ensure that a buffer has been `malloc`'d, and has not been `free`'d.
Non-executable Stack	Enhance the OS kernel's virtual memory model to make the stack segment non-executable, except under well-defined conditions.
StackGuard	Instrument stack activation records so that an integrity check can be done prior to each function call return, to prevent stack smashing exploits.
Array bounds checking	Instrument access to arrays to ensure that accesses do not occur outside of the legitmate span of the array.
Assertion checking	Check that assertion remains valid as program executes.

Purify: Purify is a debugging tool for C programs, focusing on memory problems [10]. It provides its own linker, which inserts integrity checking code around every memory reference to ensure that memory is being used in a consistent fashion. For instance, Purify ensures that dynamic memory is allocated before it is used, is freed before the program exits, is freed only

once, and is not used after it is freed. Violations of these rules produce run-time error reports to facilitate debugging, but these reports could become security alerts.

Non-executable Stack: Casper Dik and "Solar Designer" produced patches for Solaris [8] and Linux [7], respectively, that make the stack segment of the user's virtual address space non-executable. This protects programs from "stack smashing" attacks, which inject code onto the program's stack, and alter the return address to jump to that code. If the stack is not executable, this attack fails. These patches require added code to handle rare instances where the stack does need to be executable, and the kernel switches it back and forth as necessary.

StackGuard: Stackguard [6] is our compiler extension to protect programs against stack smashing attacks, similar to those considered for non-executable stacks above. StackGuard adds code to programs to do integrity checking on their activation records. When a function returns, it checks the integrity of its activation record, and if it has been altered, the program panics and issues a security alert, rather than give control to the attacker.

Array bounds checking: Numerous systems support bounds checking of array accesses. Most do so as part of the programming language semantics, which we view as an interface restriction as described in Section 3.1.1. However, array bounds checking has also been added to languages that do *not* support array bounds, such as C [12].

Assertion checking: Many languages support some form of "assertion checking", where in an assertion is embedded within the source code [1]. When the statements adjoining the assertion are executed, the asserted expression is checked for validity. If the assertion is not true, the program is presumed to have entered an invalid state, and it halts with an error message.

3.2.2 Removed Code: Minimizing Risk

The likelihood of a bug is proportionate to the size and complexity of the software in question, so the smaller a program is, the more likely it is to be correct. Here we describe techniques to *pare down* an existing software system to make it less vulnerable to unknown bugs. Paring down the complexity of a system does not automatically make the system completely trusted, but it does reduce its vulnerability to bugs to the marginal degree that it has been simplified.

Table 4: Restricting Implementations by Removing Code

System	Code Removed	Security Gained
Bastion hosts	Everything except proxy servers	Vulnerable programs are not present, and thus cannot be attacked.
"Crack a Mac" Challenge	Everything except HTTP server	MacOS has no native TCP servers, so the HTTP server is the *only* vulnerability on the host.

Table 4 shows some example implementation restrictions based on removing code. The canonical example is the bastion host in a firewall. It is essential that this host not be compromised, and so they are configured with the absolute minimum of services, minimizing potentially exploitable vulnerabilities. The "Crack a Mac" challenge [11] provides another example of this technique, where a Macintosh web server is set up as a public challenge to crackers. The platform is highly secure, because the MacOS has *no* native TCP server programs, so the installed web server is the only network vulnerability. The challenge eventually fell to an attacker [16], but the failure was a 3rd party

plug-in extension to the web server; had the plug-in code also been removed, the server would have been even more secure.

Like the interface restrictions described in Section 3.1, implementation restrictions can be applied dynamically. For instance, CGI scripts are well known to be a source of vulnerability to web servers. If an appropriate intrusion detection system is in place, a web server could choose to dynamically disable CGI extensions during times of attack, and then restore CGI extensions at a later time.

4 Interface and Implementation Permutations

Section 3 describes restrictions that can be imposed on a system to enhance its security. If the system can be modified by restricting it, what other changes can be made that might enhance security? One answer is to *permute* the system, so that it presents an unfamiliar environment to the attacker. *Static* permutations achieve approximately the same effect obtained by "security through obscurity," which tends to fail because the attackers eventually learn of the obscured system's architecture, and all the security benefits of the obscurity wear off as the attackers discover the details, while the costs remain.

Dynamic permutations provide the security benefits of security through obscurity *persistently*, because the permutations can be applied continuously, so that the attacker needs to learn each new configuration. Permutations can also be less costly than security through obscurity, if a degree of familiar structure can be preserved for the system administrator, so that it costs less than a deliberately obscured implementation. This section reviews some of the few systems that have used this technique. Similar to restrictions, permutations can be divided into interface permutations (Section 4.1) and implementation permutations (Section 4.2).

4.1 Interface Permutations

Consider an interface in object-oriented terms: a vector of methods. Now consider dynamically permuting the binding between the index number of each method and its meaning. One can only use this interface effectively if one knows the current configuration of the interface. The current interface configuration is a secret, to be distributed to the legitimate clients of the permuted interface, and kept from everyone else. Interface permutation is thus similar to encrypted communication, with the following differences:

Guessing: Key guessing must be done on the victim host; a permuted interface cannot be taken off line and analyzed by an arbitrarily fast computer. This makes guessing attempts highly obvious, and gives the defender an opportunity to arbitrarily slow down guessing attempts.

Mystery: If the attacker is only vaguely aware of the potential for interface permutation, they may become cautious when operations do not have the expected result, lest they risk detection. This is the dual of the above guess-detecting property.

Complexity: The complexity of the search space is very much smaller than for formal encryption algorithms. The complexity is the size of the permutation space of the number of possible configurations. This space can be artificially enlarged, but a determined attacker would likely notice that some parts of the space are not being used, and ignore them.

If the guess-detecting property is exploited, the defender has an opportunity to compensate for the relatively small size of the search space. Otherwise, the defender must depend on the mystery property to slow down the attacker's search of the permutation space.

There are few extant examples of interface permutations. One example is the Deception Tool Kit [5], which provides tools to spoof the existence of servers. Like the killdeer bird faking an injury to draw predators away from the nest, these faux servers are intended to draw the attacker's attention away from the machine that is running an actual server. To hold the attacker's attention, the faux servers even pretend to be vulnerable, yielding interesting results when presented with common attacks against known vulnerabilities for that server.

The DTK is an interface permutation with a small permutation space, and a relatively large amount of mystery. The permuted 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.

With additional effort, the DTK could be enhanced to allow dynamic permutation of the interface by migrating servers. When the server moves, it leaves behind a faux server for the attacker to toy with. The migrating server must, however, indicate its new location to all of its legitimate clients.

Wrappers are a successful form of interface restriction. A Class of wrappers could be developed that provides interface permutations. The DTK could be considered an example of this technique.

4.2 Implementation Permutations

Implementation permutations do not remove vulnerability to bugs, but rather seeks 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.The challenge of implementation permutations 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.

It is difficult to find such a permutation, because programs written in imperative languages are over-specified, and the attacker is exploiting a flaw in that specification. For instance, there are many vulnerabilities caused by sloppy creation of temporary files in the `/tmp` directory. Correcting these problems, even permuting their behavior, requires changing the program with respect to the specified meaning in the program's source code; the developer explicitly stated how the temporary file was to be created. An implementation permutation cannot be transparent to legitimate users unless it is faithful to the specification of the program.

The classic approach to this challenge is *N*-version programming, where a program specification is given to multiple teams to create independent implementations. This solves the problem of over-specification, but is not necessarily effective in permuting bugs. For instance, while the Windows NT and *BSD implementations of TCP/IP are completely independent, they were both vulnerable to the same set of network denial-of-service attacks that appeared in 1997, such as "teardrop" and "land". *N*-version programming also substantially increases development cost.

To generalize and automate the benefits of *N*-version programming, one could write programs in a higher level language that was less specific, giving the compiler the freedom to permute the implementation while remaining faithful to the specification. However, this will only be effective against attacks that exploit bugs that are independent of the specified program, i.e. the attacker is exploiting bugs in the virtual machine that executes the high level specification of the program, not the program itself.

To see this effect, consider Java (a slightly higher level language than C). One can permute the byte codes[6] used to implement the specified Java program, but these permutations will only defeat attacks against bugs in the Java compiler or JVM. Bugs that are present in the Java specification of the program will be faithfully reproduced by the compiler and JVM, and continue to be exploitable.

In this case, the implementation permutation *is* effective, but with respect to the implementation of a virtual machine environment for executing higher level specifications. The implementation permutation has not been applied *post hoc* to some implementation, but rather has been designed in to the virtual machine implementation.

Forrest provides an example of permutations in a virtual machine implementation [9] in the form of a modified C compiler that randomly permutes the size of stack frame activation records. Like StackGuard [6], this is designed to stop stack smashing attacks against the virtual machine implementation for C programs, which has the unfortunate property that some of the virtual machine's data structures such as the return address are accessible to the programs being executed. Forrest's compiler moves the return address around in memory, making it harder for attackers to hit.

5 Discussion

In the introduction we stated that one requirement of a secure system is that it has been designed as such, instead of retrofitted with security tools and techniques, and then pointed out that this is rarely the case for commercial software. The body of the paper describes and classifies techniques for retrofitting security onto insecure systems, which is always a second choice from the security perspective, but likely the *only* choice from a pragmatic perspective for many legacy systems. This section discusses how to use the security bug tolerance classification scheme to compare and select security bug tolerance techniques.

The classification scheme identifies what is protected, and how it is protected. This information can be used both to compare existing security bug tolerance techniques to decide which to use, and in suggesting how to devise *new* security bug tolerance techniques. In both cases, examining what needs to be protected, who it needs to be protected from, and how much it needs to be protected can indicate which form of security bug tolerance is appropriate.

5.1 Interface versus Implementation

As described in Section 3.1 and Section 4.1, interface adaptations largely amount to access control and authentication, respectively. In contrast, implementation adaptations are effective against *failures* in authentication and access control: when authentication and access control fail, implementation adaptations limit what *any* user can do, thus providing damage control. For example, a Stack-

6.Or native codes for JIT compilers.

guard protected program will prevent all types of buffer overflow attack, regardless of the motivation of the attacker (unintentional, intentional, malicious, accidental) or of the trustworthiness of the attacker (user). This is in direct contrast to a interface restriction approach, where once past a firewall (for example) a trusted user can still successfully implement a buffer overflow attack.

5.2 Restrictions versus Permutations

The dominant difference between restrictions and permutations is that restrictions reduce the amount of damage the attacker can impose, while permutations increase the cost of the attacker successfully exploiting a bug. Restrictions reduce potential damage either fractionally by disabling some operations, or probabilistically by disabling some principals from accessing the object in the hopes of containing the attacker. Permutations restrict nothing, and only make it more work for the attacker to find the bug they're looking for so they can exploit it.

Thus restrictions are suitable for applications where there are assets that require protection from potential attackers (as a subset of all potential users), while permutations are appropriate for service providers seeking high availability (for all potential users). Service providers can use machine replication and permutation to slow attacks to the point where human intervention occurs before the service is completely unavailable. Those wishing to protect assets can use restrictions to protect the assets, at the expense of the availability of service.

However, permutations are more difficult to make effective. An effective permutation 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 a permutation, that a more effective restriction could be deployed in its place:

Interface Permutations: Since interface permutations 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 authentication may be easier to deploy.

Implementation Permutations: Implementation permutations must be faithful to the program's specification, or else they become implementation restrictions. Relatively few implementation vulnerabilities are independent of the program's specification, so finding effective implementation permutations is difficult. We postulate that restricting virtual machine implementations is more effective and easier than implementation permutations.

Thus one must be cautious when presented with a permutation security bug tolerance technique. What is the permutation protecting, and how effective is it? For instance, the deception toolkit [5] is most effective at disguising a genuine service provider by surrounding it with faux service providers; how difficult is it for the attacker to identify the genuine service provider by other means? Permuting the size of stack frame activation records [9]successfully defeats existing stack smashing attacks because they depend on a static stack layout, but how difficult is it to construct stack smashing attacks that can adapt to a dynamic stack layout [6, 19]?

Implementation permutations 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 permutation, that this information could be used to more easily deploy either an interface permutation, or an implementation restriction, which we suspect would be easier to implement, more effective, or both. For instance, while Forrest's compiler makes certain stack smashing attacks more difficult to deploy, a similar implementation restriction in StackGuard makes an identical attack impossible.

As an added benefit, restrictions have permutation-like side effects. For instance, the StackGuard implementation [6] failed to detect some attacks, but these attacks were none the less stopped because of the permuted data layout induced by StackGuard.

5.3 What it All Means

All of the techniques presented are adaptations to enhance the security of a component of a system. The security maxim that you are only as strong as your weakest link still applies, so total security still depends on securing every interface that is exposed to potentially hostile principals. *However*, these techniques can also be applied to internal components, providing a degree of fault tolerance in the case where an attacker breaks through what should have been secured. The more components that have been made bug tolerant, the more likely the system will be able to stop an attacker at some point.

Each security bug tolerance adaptation has a cost to deploy, both in development time and in performance. The choice of which components to harden, and using which techniques, thus becomes a trade-off of how much one is willing to pay for a degree of security bug tolerance. The categorization that we have provided should help in measuring costs versus effectiveness.

6 Conclusions

Creating bug-free software is difficult and expensive, so vendors tend to release buggy software. Thus we view it as *normal* for software to have bugs, and hence security vulnerabilities induced by bugs. Therefore commercially relevant commodity systems will have to become *security bug tolerant* if they are to provide a reasonable degree of security. Security bug tolerance requires more than just a simple application of classic fault tolerance techniques, because security faults are Byzantine, and fault tolerance techniques largely assume that failures are fail-stop. We have presented a classification scheme for security bug tolerance techniques, populated it with examples from our own work and from the literature, and discussed the relative costs and benefits of these techniques to aid analysts in comparing security bug tolerance techniques.

References

[1] Anonymous. `assert(3)` C Library Function. section 3 of most UNIX Systems manuals.

[2] 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.

[3] 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.

[4] Klaus Brunnstein. Mr. Bill Gates: MS Software Essentially Bug-free. `comp.risks` 17.43,

October 1995. http://catless.ncl.ac.uk/Risks/17.43.html#subj5.

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

[6] 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*, San Antonio, TX, January 1998.

[7] "Solar Designer". Non-Executable User Stack. http://www.false.com/security/linux-stack/.

[8] 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.

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

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

[11] Joakim Jardenberg. Crack a Mac Contest. http://hacke.infinit.se/, February 1997.

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

[13] Gene H. Kim and E.H. Spafford. Writing, Supporting, and Evaluating Tripwire: A Publicly Available Security Tool. In *Proceedings of the USENIX UNIX Applications Development Symposium*, pages 88–107, Toronto, Canada, 1994.

[14] Calvin Ko, George Fink, and Karl Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL, December 1994.

[15] John McLean. Is the Trusted Computing Base Concept Fundamentally Flawed? In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[16] Martin Minow. "Crack a Mac" Server Cracked. comp.risks 19.31, August 1997.

[17] Nathan Myers. FOCUS Magazine Interview with Bill Gates: Microsoft Code Has No Bugs. http://www.cantrip.org/nobugs.html.

[18] 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. http://www.usenix.org/publications/library/proceedings/osdi96/necula.html.

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

[20] D.L. Parnas, S.P. Kwan, and J. van Schouwen. Evaluation Standards for Safety Critical

Software. In *Proceedings of the International Working Group on Nuclear Power Plant Control and Instrumentation, IAEA NPPCS Specialists' Meeting on Microprocessors in Systems Important to the Safety of Nuclear Power Plants*, London, UK, May 1988.

[21] William R. Schockley. Is the Reference Monitor Concept Fatally Flawed? The Case for the Negative. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

[22] 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.

[23] Howie Shrobe. ARPATech '96 Information Survivability Briefing. `http://www.darpa.mil/ito/ARPATech96_Briefs/survivability/survive_brief.html`, May 1996.

[24] Internet Security Systems. Real-time Attack Recognition and Response: A Solution for Tightening Network Security. Report, Internet Security Systems, 1997.

[25] Unknown. "Interview with Bill Gates". *FOCUS*, (43):206–212, October 23 1995.

[26] 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`.

[27] 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/`.