In the 38th *IEEE/IFPF International Conference on Dependable Systems and Networks*, Dependable Computing and Communications Symposium. Anchorage, June 2008.

Security through Redundant Data Diversity

Anh Nguyen-Tuong, David Evans, John C. Knight, Benjamin Cox, Jack W. Davidson
University of Virginia, Department of Computer Science
{*nguyen* | *evans* | *knight* | *btc4w* | *davidson*}*@cs.virginia.edu*

Abstract

Unlike other diversity-based approaches, N-variant systems thwart attacks without requiring secrets. Instead, they use redundancy (to require an attacker to simultaneously compromise multiple variants with the same input) and tailored diversity (to make it impossible to compromise all the variants with the same input for given attack classes). In this work, we develop a method for using data diversity in N-variant systems to provide high-assurance arguments against a class of data corruption attacks. Data is transformed in the variants so identical concrete data values have different interpretations. In order to corrupt the data without detection, an attacker would need to alter the corresponding data in each variant in a different way while sending the same inputs to all variants. We demonstrate our approach with a case study using that thwarts attacks that corrupt UID values.

1. Introduction

Distributed computing relies upon networked services that are exposed to malicious adversaries. These adversaries, posing as legitimate clients, attack the services with which they interact, doing so by exploiting vulnerabilities in the service software. Despite much effort, it has proven difficult to build services that do not contain security vulnerabilities.

The N-variant systems approach makes use of redundancy, using an architecture that combines tailored program diversity and execution monitoring to provide strong security guarantees that do not rely on assumptions about keeping secrets. The transformations used to generate variants can be simple and the keys used to generate the variants can be openly published. The N-variant architecture enables high-assurance arguments to be made with respect to specific attack classes, regardless of the vulnerability exploited.

A simple example is *address space partitioning*, in which a program *P* is replaced with two variants P_0 and P_1 (Figure 1). The variants are constructed to behave identically to *P* on normal inputs, but use disjoint memory regions: P_0 uses addresses that start with a 0 bit while addresses for P_1 start with a 1 bit. All inputs are replicated and sent to both variants. A monitor observes both variants and reports an attack if their behaviors diverge. An attack that involves accessing a specific absolute memory address (e.g., typical format string, stack and heap smashing, and return-to-libc attacks) may be constructed to succeed against *either* P_0 or P_1, but if that same input is run on the other variant it is guaranteed produce a memory access error which will be detected by the monitor. Thus, an attack that relies on directly inserting an absolute address is *impossible* (assuming the framework replicates inputs correctly and the monitor observes both variants behavior with sufficient granularity) since the high bit cannot be 0 and 1 at the same time.

Our earlier work introduced N-variant systems and demonstrated address space partitioning as well as another instance of the approach for defeating code injection by tagging instructions in different variants with different values and checking and removing the tags before execution [16]. Other researchers have developed other variations within similar frameworks: Bruschi et al. created a variation to thwart partial memory overwrites [9], and Franz created a variation using reverse stack ordering that provides probabilistic protection against certain relative memory corruption attacks [20]. All of these variations alter some low-level, program-wide property such as the format of

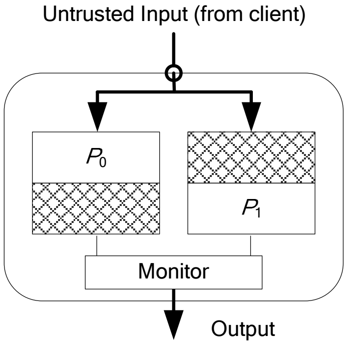

Figure 1. Two-variant address partitioning.

instructions or the address space.

Such variations are promising for thwarting large general attack classes, but provide only a glimpse of the opportunity N-variant frameworks provide, namely, the ability to deploy any diversity technique, including low-entropy variations, in a way that provides high assurance security against a particular attack class.

In this paper, we develop a general method for employing *data diversity* in N-variant frameworks. Data diversity is a general approach to software fault tolerance in which identical copies of a program are executed with different data, and their outputs are subject to a vote [1]. The different versions of the data are obtained from the original input by a process known as *reexpression*, and the reexpression function is chosen so that either the results of the program are unaffected or the effects of reexpression on the outputs can be reversed easily. In traditional data diversity, the goal of reexpression is to avoid the regions of the input space for which the program fails. Since these regions are unknown in general, traditional data diversity offers only probabilistic guarantees of tolerating software faults. Our work is focused on security against malicious attacks, so instead of using a majority vote we interpret any divergence in behavior as a security violation. To achieve high assurance security properties, our goal is to find reexpression functions that are disjoint, so that any data corruption attack will be detected as a divergence.

Unlike previous diversity techniques that are applied universally to a process' address space, data diversity techniques depend on understanding the underlying semantics of program data. The data and program must be transformed in a way that preserves the original program semantics while allowing the program to operate on a different concrete data representation. Different data diversity techniques could be employed for different types of program data. As an example of our technique, we develop a variation that diversifies user IDs to thwart a class of data corruption attacks where user identification data is corrupted to gain root privileges or masquerade as an arbitrary user.

The primary contribution of this paper is the development of a method for designing, implementing, and reasoning about N-variant systems that employ data diversity. Section 2 presents a model for data variation and explains how previous work on N-variant systems fits into our model. Section 3 demonstrates our approach using a data diversification that thwarts attacks that target corrupting UID values. Section 4 reports on a case study implementation of our technique for the Apache web server. Section 5 discusses general lessons learned from our experience designing and implementing the UID variation. We present related work (which, surprisingly, extends to the 18th century) in Section 6 and conclude in Section 7.

2. Model

In previous work, we reasoned about N-variant systems by considering the sequence of program states in each variant [16]. Obtaining the desired detection and correctness properties required establishing two properties:

1. *Normal equivalence*: When executing on normal (non-malicious) inputs, the variants remain in semantically equivalent states. To establish normal equivalence, a canonicalization function is used to map the states of all variants onto a canonical state. (For the address partitioning example, the canonicalization function maps the address spaces into the same space.)

2. *Detection* – when executing on abnormal (attack) inputs, the variants diverge in a way that is detectable by the monitor. Typically, this occurs when one of the variants enters an alarm state. (For the address partitioning example, the detection property occurs when an attack injects an absolute address, which causes one of the variants to segmentation fault.)

This model provides a general framework for reasoning about N-variant systems, but does not provide much insight for designing or reasoning about how to use data diversity effectively. The difficulty is this model relies on reasoning about the entire program state.

2.1 Interpreters Model

To reason about data diversity variations, we prefer a model that allows us to reason more directly about how data transformations preserve the necessary normal equivalence and detection properties. We consider an application as being composed of a series of interpreters, typically organized hierarchically. Each interpreter processes a particular type of data. For example, a web application depends on interpreters for handling the network protocol, the HTTP protocol, interpreting scripts that implement application logic, executing database queries, accessing operating system services, and executing machine instructions.

To carry out a successful attack an attacker needs to break through several layers of interpretation and control inputs to a specific target interpreter. For example, if the malicious payload consists of x86 machine instructions, the targeted interpreter is the machine hardware itself. If the attack payload opens a shell (e.g., by executing `/bin/sh` on Unix systems), then one targeted interpreter might be the filesystem. A single exploit may target many different interpreters.

The reason why an attacker is able to send malicious data to a targeted interpreter is that higher-level interpreters contain vulnerabilities. Software is often deployed with many residual faults, some of which turn out to be severe security vulnerabilities.

Figure 2 illustrates an N-variant system with two variants using different interpreters for some data type, but otherwise implementing the same program. The attacker is constrained to use the same communication channel as regular user input (External Input), and will attempt to craft input that compromises the application. This external input, including its embedded malicious payload, will be interpreted by a series of interpreters in the application, abstracted in the figure by a single interpreter, App Interpreter. By exploiting a path through App Interpreter containing a vulnerability, the embedded malicious data reaches the target interpreter.

In general, diversity techniques attempt to thwart attacks by changing the interfaces between interpreters. If an attacker does not know this interface, the attacker will have difficultly guessing an input that has the desired effect on the target interpreter.

With data diversity, the variations are created by using different data reexpression functions. If there is a large space of possible reexpression functions and associated secrets, it may be possible to provide a high degree of security with a single variant. To inject specific malicious data, the attacker needs to know the particular inverse reexpression function that is used. This configuration corresponds to the use of synthetic diversity techniques such as address space [8][42] and instruction set randomization for disrupting attacks [6] [25][28]. Security arguments for such techniques are based on the claim that it is difficult for an attacker to guess the randomization key. In practice, keeping randomization keys secret has proven difficult, as demonstrated by attacks on address space randomization [37] and instruction set randomization [38] techniques that exploit the limited actual entropy available for randomizations and probing opportunities.

The N-variant framework obviates the need for secrets and high entropy. The reexpression functions are designed so that *any* concrete data that is valid for one variant is invalid for the other. The target interpreters are not designed to attempt to distinguish between malicious and normal data directly. Instead, they rely on the fact that the same malicious data will be sent to both target interpreters, whereas normal application data will have been reexpressed.

As shown in Figure 2, each variant has a different reexpression function (R_0, R_1), and hence will operate on different data. Trusted data embedded in P is transformed using these functions in the corresponding variants. To preserve program semantics, the target interpreters are preceded by the corresponding inverse reexpression functions, $R^{-1}{}_0$ and $R^{-1}{}_1$. This establishes a different data interpretation between the application and target interpreters.

2.2 Normal Equivalence

Consider a data variation for a target type T and a given program P. To establish the normal equivalence property for each variant P_i we need to show:

(1) All trusted data of type T used by P is transformed using the reexpression function R_i.

(2) All instructions in P_i that operate on T values directly (that is, without sending them to the target interpreter) are transformed to preserve the original semantics when operating on reexpressed data.

In addition, we need to show the reexpression function and its inverse are indeed inverses:

(3) $\forall x$: T, $R^{-1}{}_i(R_i(T)) \equiv T$. (*inverse property*)

Showing the necessary inverse property holds is usually straightforward since the reexpression function is designed to have this property.

Establishing the first two properties requires reasoning about a program transformation (and possibly also about transformation of other external data as seen in Section 3.4). Transforming trusted program data requires identifying the constant data of the target type in P, and applying R_i to it to produce P_i. If the target data type is well defined, this should be fairly straightforward. Preserving the semantics is a more challenging problem. At worst, the inverse reexpression function can be embedded in the program to preserve the semantics of the original code.

2.3 Detection

The detection property states that if one variant is

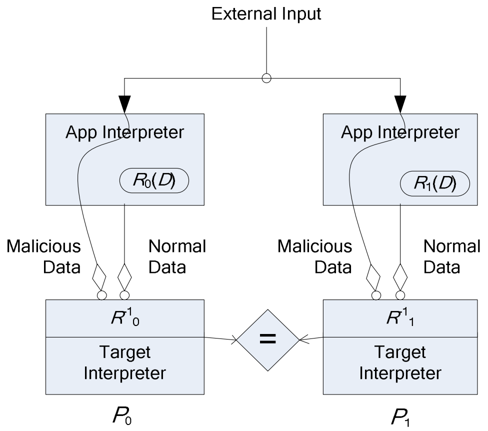

Figure 2. N-Variant Systems with Data Diversity.

Variation	Target Type	Reexpression Functions	Inverse Functions
Address Space Partitioning [16]	Address	$R_0(a) = a$ $R_1(a) = a + \text{0x80000000}$	$R^{-1}{}_0(a) = a$ $R^{-1}{}_1(a) = a - \text{0x80000000}$
Extended Address Space Partitioning [9]	Address	$R_0(a) = a$ $R_1(a) = a + \text{0x80000000} + \text{offset}$	$R^{-1}{}_0(a) = a$ $R^{-1}{}_1(a) = a - \text{0x80000000} - \text{offset}$
Instruction Set Tagging [16]	Instruction	$R_0(inst)= 0 \,\|\, inst$ $R_1(inst)= 1 \,\|\, inst$	$R^{-1}{}_0(0 \,\|\, inst) = inst$ $R^{-1}{}_1(1 \,\|\, inst) = inst$
UID Variation (this paper)	UID	$R_0(u)= u$ $R_1(u)= u \oplus \text{0x7FFFFFFF}$	$R^{-1}{}_0(u)= u$ $R^{-1}{}_1(u)= u \oplus \text{0x7FFFFFFF}$

Table 1. Reexpression Functions.

compromised, the other must be in a state that indicates an attack. This requires that any injected data of the target type will be detected when the target interpreters compare their input data. This is achieved if the inverse reexpression functions are disjoint:

$$\forall x \colon R^{-1}{}_0(x) \neq R^{-1}{}_1\,(x)\ (\textit{disjointedness property}).$$

Hence, any time an identical value is sent to both interpreters an alarm is raised, since the inverted values must be different.

Detection is only guaranteed by this property if all transformations that the application interpreter performs on input data are identical in the two variants. Otherwise, an attacker may be able to craft an input Z that is transformed by P_0 into Z' and P_1 into Z'' (where $Z' \neq Z''$) before it is sent to the target interpreter.

The detection property also requires that an attack must inject complete values of the targeted type. For example, address space partitioning provides protection only against attacks that inject complete addresses. It is vulnerable to an attack that can corrupt just the three low-order bytes of an address, leaving the high-order byte unchanged. The extended version of address space partitioning is (probabilistically) resilient to a byte-overwriting attack since the low order bytes will also differ between variants.

2.4 Examples

Table 1 summarizes four variations using our model. The first three variations were developed in previous papers; we introduce the fourth variation in the next section. For the previous variations, the target type is broad: for the first two, it is all addresses, and for instruction set tagging it is all instructions. Hence, creating the variations to satisfy the needed normal equivalence property is fairly straightforward and requires no analysis of the program.

3. UID Data Variation

We now examine a data diversification designed to thwart attacks that corrupt user ID data. This is a type of non-control data attack as described by Chen et al. in which an attacker corrupts a data value that causes the original program to execute maliciously [12].

We focus on the corruption of user and group identification data (UID/GID), although data diversity techniques could be designed to provide protection against other data attacks. In the rest of the paper, we use the term UID to denote both UID and GID values.

To test the idea of data variation, we implemented a UID variation on the Apache web server [2]. A common pattern for servers is to drop their privileges when handling client requests. However, there will be instances when accessing critical system resources that require the escalation of privileges to the root account. If an attacker can corrupt the UID value used to drop or escalate privileges, then the attacker can masquerade as root (or any other user) in the system. Chen et al. describe one example of such an attack [12].

3.1 N-Variant Framework

Before describing our variation strategy and its implementation, we review the existing N-variant framework prototype [16]. Our implementation is a Linux kernel modified to execute the variants using system call boundaries for both synchronization and monitoring purposes. To run a program as an N-variant system, the variant executables are created. Then, a script is used to launch the N-variant system with the selected variants, e.g., `nvexec prog1 prog2`.

We updated kernel data structures to keep track of variant processes and implemented wrappers around system calls. System calls are used as synchronization points: once one variant makes a system call, it will not proceed until all other variants make the same system call. We wrap input system calls so that the actual input operation is only performed once and the same data is sent to all variants.

This removes most sources of non-determinism since each variant receives the same result for system calls. However, our implementation does not yet handle issues involving scheduling divergences that can be caused by signals and threading [16]. For example, if a signal is delivered to variants at different points in their execution, their behaviors may diverge. This leads to a false attack detection. Bruschi et. al. have developed a different implementation of a similar redundant execution framework that provides some steps towards simultaneous signal delivery [9].

The wrappers also act as monitors and check for divergent behavior by making sure that all system calls receive equivalent arguments before allowing the actual system call to proceed. For output related system calls, we also check that the variants are making equivalent system calls, and issue the actual call only once.

3.2 Reexpression Functions

To defend against this attack class, we adopted a reexpression function that is resilient to partial data value corruptions. For P_0, the reexpression function (and its inverse) is the identity function. Hence, UID = 0 corresponds to root as normal. For P_1, we use:

$$R_1(u) = u \oplus \mathrm{0x7FFFFFFF}$$
$$R^{-1}{}_1(u) = u \oplus \mathrm{0x7FFFFFFF}$$

Hence, 0x7FFFFFFF represents root. The reexpression functions satisfy both the inverse property (the XORs cancel out) and the disjointedness property (flipping bits always changes the value).

This reexpression function is susceptible to a high bit overwrite, since the high bit is not flipped. Ideally we would have used a reexpression function that flips all bits in the data value (XOR with 0xFFFFFFFF). This causes some implementation difficulties. Although the UID datatype is normally unsigned. The kernel internally treats negative UID values as special cases so flipping the high bit (sign) would cause difficulties.

Although individual bit attacks are certainly possible in theory, the lowest level of granularity reported for partial memory overwriting attacks under a remote attacker threat model is at the byte-level so we do not consider this a likely threat. While bit flips have been reported for other threat models, e.g., the heat lamp attack on the Java virtual machine [3], no known realistic attack allows an attacker to reliably target a specific bit to flip.

3.3 Applying Reexpression Functions

To create the variants we must transform the program to incorporate our reexpression function. Since the reexpression function for P_0 is the identity function, the original program can be used unchanged for the first variant. To create the second variant, we perform a source-to-source program transformation. For our case study, the transformation was done manually, but in a way that could be readily automated (as discussed in Section 5). To apply the transformations, our transformer must be able to determine which values in a program are UID values. For a well-typed C program, all values used as UIDs are typed `uid_t`, and the `uid_t` type is never used to hold non-UID values.

For the second variant, we need to establish the first two properties required for normal equivalence from Section 2.2: (1) all UID values in P_1 must be transformed using $R_{1;}$ and (2) all instructions in P_1 that operate directly on UID values must be transformed to preserve the original semantics when operating on reexpressed values.

For the first property, we identify all UID constants using the C data type, and replace these values with the result of applying R_1 to them. In some situations, constants are used implicitly. For example, an if statement such as `if(!getuid())` contains an implied comparison to the constant `0`. The statement is replaced with `if(getuid()==0)`. This is to have the UID constant explicitly stated, after which the constant value is transformed.

The second property requires modifying code that manipulates UID values. We assume that only assignment and comparison operations are applied to UID values. Programs do not typically perform other operations on UID values, but if a program uses other operations on UID values additional transformations would be needed. Handling assignments and equality comparisons requires no code changes; if the operation involves a constant value, it was already transformed by the data transformation. Inequality comparisons must be logically reversed, however, to preserve the original semantics on transformed values (where all bits except the high bit have been flipped).

3.4 Support for External Data

Our data variation requires that *all* trusted data used by the variants is transformed using the reexpression function. Otherwise, untransformed data will have the wrong representation when it reaches the target interpreter. The transformations in the previous section transform data in the program itself, but many servers also rely on external data such as configuration files for their proper operations. For example, Apache uses UID values in the `/etc/passwd` and `/etc/group` files.

We thus needed to develop a mechanism for the two variants to receive varied data originating from trusted external sources. One approach would be to apply the reexpression functions as data is read from external sources. This seems risky, however, since an attacker may be able to corrupt data by using this same path. The alternative is to provide two versions of the trusted files and extend the framework to support file variants for the program variants. This approach is more general, and opens other interesting possibilities based on diversity of data in configuration files.

To enable this, we created the notion of *unshared files*. Previously, all files were shared since all variants operated on identical data. I/O system calls were performed once and the result was passed to all variants. Now, when the variants make a request to open an unshared file, the kernel opens a different file for each variant that contains data specific to that variant. For example, when both variants request that `/etc/passwd` be opened, P_0 will actually open `/etc/passwd-0` and P_1 will open `/etc/passwd-1`. The diversified password files are identical except the UID values are transformed using the appropriate reexpression function. When the variants then perform an operation such as a read on an unshared file, each will do it on its separate file, while shared files will behave the same as before, having one variant perform the system call and giving all variants the same result.

We modified the kernel so that each variant keeps its own file table data structure where information about the processes' open files resides. We keep this data-structure synchronized between the variants so that the n^{th} slot in P_0's data structure corresponds to the n^{th} slot in P_1's data structure. When a file is opened, the kernel creates an entry for that file in each variants' file table. If the file being opened is shared (the normal case), the kernel marks the bit in the shared files data-structure to indicate it, otherwise it will clear that bit. When subsequent system calls are made that use a file descriptor, the kernel accesses the shared files bitmap and determine if the files are shared or unshared. If they are shared, the kernel will have P_0 perform the system call and give the result to all variants. If the file is unshared, each variant will perform the system call reading or writing data to their own diversified file. When the files are closed the kernel will clear the entry in all variants' file tables.

3.5 System Calls

The kernel calls that take UID parameters are the target interface for the data variation. Hence, the implementations of these calls should incorporate the inverse data transformation. We also use the system calls to check that the variants have not diverged. They should operate identically on the same data (after it has been transformed using the appropriate inverse reexpression function).

We modified the wrappers of all system calls that involve UID parameters. For calls that take UID parameters such as `long setuid(uid_t)`, the wrapper applies the inverse reexpression function. It also checks that the same actual (post-inverse transformation) values are passed into the call by all variants. For the system calls that return a UID value such as `uid_t getuid()`, the wrapper applies the re-expression transformation on the result (which is trusted), giving each variant its own varied UID value.

Function Signature	Description
`uid_t uid_value(uid_t)`	Compares parameter value (across variants) and returns passed value.
`bool cond_chk(bool)`	Checks conditional value given between variants is the same.

`bool cc_eq(uid_t, uid_t)`
`bool cc_neq(uid_t, uid_t)`
`bool cc_lt(uid_t, uid_t)`
`bool cc_leq(uid_t, uid_t)`
`bool cc_gt(uid_t, uid_t)`
`bool cc_geq(uid_t, uid_t)`
	Compares parameters and returns the truth value for comparison.

Table 2. Detection System Calls.

We are also concerned with attacks where a UID value is corrupted in a way that leads to other behaviors before one of the system calls involves a UID parameter directly. Ideally, the monitor would observe and check the variants to be in normally equivalent states after each transition. This is impractical, so our current implementation approximates this by observing the system call made by the variants and ensures that they are equivalent. To ensure detection, we transform the program to expose UID uses to the monitor with newly created system calls. This ensures that the monitor observes any UID divergence before the corrupted UID value is used.

Table 2 summarizes the newly created system calls. The `uid_value(uid_t)` function passes the UID value to the kernel which compares the values across the variants and ensures they have equivalent meanings (i.e., they are identical after applying the appropriate inverse reexpression functions). The function returns the same value that was passed in. An example where this is used is in `getpwname(uid_t)`:

```
pw = getpwname(uid);
```

becomes

```
pw = getpwname(uid_value(uid));
```

The `cond_chk(bool)` function checks a condition code, which UID values may directly or indirectly affect. It is passed in the result of a conditional expression and ensures that both variants take the same path. For example, `(pw == NULL)` would be replaced by `(cond_chk(pw == NULL))`.

Configuration		1	2	3	4
Description		Unmodified Apache	Transformed Apache	2-Variant Address Space	2-Variant UID
Unsaturated	Throughput (KB/s)	1010	973	887	877
	Latency (ms)	5.81	5.81	6.56	6.65
Saturated	Throughput (KB/s)	5420	5372	2369	2262
	Latency (ms)	16.32	16.24	37.36	38.49

Table 3. Performance Results.

The other system calls are used when directly comparing two UID values (=, ≠, <. ≤, >, ≥). They could be written using the `cond_chk` call, but providing these additional calls offers two advantages: (1) it reduces the number of system calls needed to perform the check since both UID values are checked with one system call, and (2) the variants' instruction streams remain identical, while if the comparison were done in user space, P_1's operators would need to get switched (≤ becomes ≥) due to the data variation. For example, `(uid == VARIANT_ROOT)` is replaced by `(cc_eq(uid, VARIANT_ROOT))`.

4. Apache Case Study

To evaluate our variation, we conducted a case study on the Apache web server. To create Apache variants we needed to make a total of 73 changes to the source code. Fifteen of the changes involved applying the reexpression function to constant UID values in the source code. We needed 16 changes to introduce the new system calls to expose single UID value usages to the monitor, 22 changes to expose conditional statements that compared UID values, and 20 changes to check conditional statements.

Constructing variants by hand is tedious and error prone. Without any automation, this variation would not likely be practical. There are two main parts of this transformation. First, identifying the variables that contain UID values. If the programmer uses the `uid_t` and `gid_t` data types strictly, then it would only require identifying which constant values were assigned or compared to those variables and changing them according to the variation. If the programmer did not use `uid_t` data type to declare the variables, they could be inferred using dataflow analysis by seeing which variables stored the result of functions returning a known uid value (e.g., `getuid`) or were passed as a parameter to a function expecting a user id (e.g., `setuid`). Several static analysis tools, including Splint [31], are available that already do this analysis. Using this simple analysis technique would have identified all instances of UIDs in the Apache Web Server.

Once all the UID values were identified and changed accordingly, we exposed the uses of UID variables to the monitor using the newly developed system calls (Table 2).

Apache only had one complicating factor. If Apache encountered an error related to the UID, it would write an error message including the UID to a log file. If these output statements were left unmodified, it would result in a divergence since the UID values are different. However, modifying the statements so P_1 converts the UID value would open a potential security vulnerability. We worked around this problem simply by removing the user id value from the log output.

Table 3 summarizes our performance results. We measured the throughput and latency of our system using WebBench 5.0 [41], a web server benchmark that serves a variety of static web page requests. We ran two sets of experiments measuring the performance of our Apache server under unsaturated and saturated load conditions. For the first set of experiments, we used a single client machine running one WebBench client engine. For the load experiments, we saturated our server using 3 clients each running five WebBench clients connected to the same networks switch as the server. In both sets, a single 1.4 GHz Pentium 4 server machine with 384 MB RAM ran Fedora Core 5 (2.6.16 kernel) using 4 different configurations.

Configuration 1 is the baseline configuration: unmodified apache running on our kernel. Note that in general an unmodified program running under our modified kernel incurs practically no overhead. The only overhead would be the addition of an extra check (an if statement to determine if a process is participating in N-variant system) per system call.

Configuration 2 shows the overhead of the UID code transformations made to Apache. In our experiments, it was negligible; this is unsurprising since most of the UID operations are done when the server initializes. The additional overhead is one system call per request to compare two UID values.

Configuration 3 is a 2-variant system where the two variants differ in the address spaces with the kernel configured to support unshared files. This configuration provides a baseline case when running two variants and can be used to measure the overhead of any additional variations. For the unloaded server, this resulted in a throughput decrease of 12.2% and a latency increase of 12.9% from the baseline configuration. For the loaded server, throughput decreases by 56% while latency increases by 129%.

Since the N-variant system executes all computation twice, but all I/O system calls only once, the overhead incurred reflects the cost of duplicating computation, as well as the checking done by the wrappers. The overhead measured for the unloaded server is fairly low, since the process is primarily I/O bound. For the loaded server, the process becomes more compute-bound, and the approximate halving of throughput reflects the redundant computation required from running 2 variants.

Configuration 4 is a 2-variant system running the UID variation described in Section 4. We present overhead relative to Configuration 3 to measure the added overhead of our variation. For the unloaded server, throughput decreased by 1%, while latency increased by 1.4%. For the loaded server, throughput decreased by 4.5%, while latency increased by 3%.

These results are encouraging in that although the overall overhead is high because of the redundant computation, additional variations may be performed at relatively low cost. This opens up the practical possibility of combining variations to achieve broader coverage of attack classes. However, variation composition must be done carefully to ensure that variations still satisfy the required normal equivalence properties when they are composed [16].

In general, our results indicate that for I/O bound services, N-variant systems with the UID variation can be done with performance overhead that would be acceptable for many deployments. For CPU-bound services, the overhead of our approach is high since all computations need to be performed twice. Multiprocessors may alleviate some of the problem (in cases where there is not enough load to keep the other processors busy normally) [20].

5. Discussion

Designing data variations for non-control data attacks is more difficult than we had anticipated. In particular, applications such as Apache rely on external configuration files such as `/etc/passwd` and `/etc/group` to map user names to UIDs. We wanted to avoid embedding the reexpression functions directly inside the web server itself since this would have opened up a potential path by which an attacker could bypass detection by reusing the reexpression functions. Our solution was to provide support for the concept of unshared files, in which the variants read from their respective reexpressed files (e.g., `/etc/passwd-0` for variant 0 and `/etc/passwd-1` for variant 1). Although we have not yet explored other applications of unshared files, they provide other exciting opportunities for diversity. For example, web server variants could be run with different directory structures and different configuration files to thwart attacks on file paths.

For detection, we defined new system calls to synchronize and check for the validity of UID values at the point of use. This design choice was motivated by our desire to make strong arguments regarding (nearly) immediate detection of corrupted UID values. Another possibility is to rely on the already existing monitoring mechanism for checking divergence at system call boundaries at the cost of detection precision. From our performance results, the costs of these extra system calls appear to be minor.

Varying UIDs as a reexpression strategy required making strong assumption about their uses being limited to assignments and comparisons. This assumption turns out to be warranted for a simple data type like UIDs, but UIDs are only one type of security critical data identified by Chen et al. [12]. Our next step is to investigate data variations for other types of security-critical data such as configuration data and decision-making data. In the general case, data operations can be much more complex, e.g., functions that manipulate strings such as regular expression matchers. More complex data types pose more challenges in diversification while preserving semantics, but also opportunities to thwart larger attack classes. If data types are properly encapsulated, perhaps via C++ classes, we could safely maintain program semantics while varying data representations provided the class interface did not leak internal implementation details.

6. Related Work

The first use of data diversity of which we are aware was by British Astronomer Royal, Nevil Maskelyne, who employed data diversity techniques using human computers to improve the reliability of astronomical tables published in the 1767 *Nautical Almanac* [17][23]. For the lunar tables, Maskelyne would assign one (human) computer the task of calculating the moon's position at noon for each day of the month, and another computer (known as the *anticomputer*) the task of calculating the moon's position at midnight. A third person known as a *comparer* was responsible for merging and checking the computers' results.

We discussed the most closely related recent work on N-variant systems in the introduction. Next, we consider other defenses suggested by the interpreter model, and other work on redundant execution.

Other Defenses. An orthogonal strategy is to eliminate vulnerabilities altogether so that malicious data cannot reach the target interpreter. An example of this strategy is to use type-safe languages to eliminate memory vulnerabilities or using point defenses against specific vulnerabilities [14][18][36]. Another strategy is to seek ways to distinguish trusted and untrusted data. An example would be taint analysis techniques to track the flow of information from untrusted sources and prevent their use in security-critical functions [24][33][34][43].

Diversity Techniques. Numerous diversity techniques have been proposed for increasing the difficulty of exploiting vulnerabilities, including randomizing instructions [6][28], memory layout [8][42], compiler layout [1][19], encrypting pointers [15][40], and operating system interface [13]. Unlike the N-variant systems approach, all of these works rely on attackers' inability to guess a secret key for security.

Redundant Computation. *N-version programming* [4][11][26] (from which we adopted the name *N-variant systems*) uses multiple independent teams to produce the software intended to implement the same requirements. It is based on design diversity, in the hope of avoiding common faults between versions. However, Knight and Leveson have shown experimentally that even separate teams are likely to make similar mistakes [29]. Furthermore, N-version programming is resource-intensive, and thus typically applied to critical systems only. Littlewood et. al present a recent overview of design and data diversity, and their application to security [32].

For popular servers, such as web servers, multiple implementations of the same protocol may be available. The HACQIT project [27][35] deployed two web servers (IIS running on Windows and Apache on Linux) and checked HTTP status code to indicate divergence. Totel, Majorczyk and Mé extended this idea and compared the actual web page responses of the servers [39]. The challenge in this approach is to distinguish benign differences in the output arising because of design difference in the servers or host specific properties, from differences that indicate an attack. Gao, Reiter and Song correlate system calls between web servers to identify attacks [21][22]. Of these, the first two approaches would not have detected a UID exploit provided the attack did not perturb the output web pages. Gao et al.'s system may potentially detect such an attack if it results in sufficiently non-correlated system calls. In contrast, using our approach we can make strong guarantees about detecting all attacks in a particular attack class.

Berger and Zorn proposed a redundant execution framework with multiple replicas each with a different randomized layout of objects within the heap to provide probabilistic memory safety [7]. Their replication framework only handles processes whose I/O is through standard in/out, and only a limited number of system calls are caught to ensure all replicas see the same values. Their goals were to enhance reliability and availability, rather than to detect and resist attacks. An extension would be to combine the fine-grained monitoring capabilities of N-variant systems with probabilistic variations such as theirs.

7. Conclusion

The N-variant systems approach to security holds the promise for building systems whose security properties with respect to particular attack classes can be assured with high confidence. Furthermore, these properties can be achieved without relying on secrets, and using low-entropy transformations.

In this paper, we developed a general approach to data diversity for N-variant systems and demonstrated this approach with a data variation for combating attacks that involve corruption of UID values. Although this particular problem can be more easily combated in other ways, the approach described is promising in demonstrating how low-entropy data diversity can be used to provide high assurance security against particular attack classes. In future work we plan to investigate the addition and composition of further data diversity techniques.

Acknowledgements

We gratefully acknowledge support from the National Science Foundation through awards CNS-0524432 and CNS-0627523.

References

[1] P. E. Amman and J. C. Knight. Data Diversity: an Approach to Software Fault Tolerance. *IEEE Trans. On Computers*, 37 (4), pp. 418-25, 1988.

[2] Apache Software Foundation. Apache HTTP Server project. http://httpd.apache.org.

[3] A. Appel and S. Govindavajhala. Using Memory Errors to Attack a Virtual Machine. *IEEE Symp. On Security and Privacy*. 2003.

[4] A. Avizienis and L. Chen. On the Implementation of N-version Programming for Software Fault-Tolerance During Program Execution. *International Computer Software and Applications Conference*. 1977.

[5] A. Baratloo, N. Singh, T. Tsai. Transparent Run-Time Defense against Stack Smashing Attacks. *USENIX Technical Conference*. 2000.

[6] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *ACM Computer and Communications Security*. 2003.

[7] E. Berger and B. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. *Programming Language Design and Implementation* (PLDI). June 2006.

[8] S. Bhatkar, D. DuVarney, and R. Sekar. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. *Usenix Security.* 2005.

[9] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified Process Replicae for Defeating Memory Error Exploits. *3rd Intl. Workshop on Information Assurance*. 2007.

[10] W. Cheng, Q. Zhao, B. Yu, S. Hiroshige. TaintTrace: Efficient Flow Tracking with Dynamic Binary Rewriting. *Computers and Communications*. 2006.

[11] L. Chen and Algirdas Avizienis. N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation. *Fault Tolerant Computing Symposium*. 1978.

[12] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. *USENIX Security.* 2005.

[13] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. *Tech Report CMU-CS-02-197*. December 2002.

[14] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. *USENIX Security.* 2001.

[15] C. Cowan, S. Beattie, J. Johansen, P. Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. *USENIX Security.* 2003.

[16] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, J. Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. *15th USENIX Security.* August 2006.

[17] M. Croarken. Tabulating the Heavens: Computing the Nautical Alamanac in 18th-Century England. *IEEE Annals of the History of Computing*. 2003.

[18] H. Etoh. *GCC extension for protecting applications from stack-smashing attacks.* IBM. 2004. http://www.trl.ibm.com/projects/security/ssp

[19] S. Forrest, A. Somayaji, D. Ackley. Building diverse computer systems. *6th Workshop on Hot Topics in Operating Systems*. 1997.

[20] M. Franz. Understanding and Countering Insider Threats in Software Development. *UC Irvine Technical Report ICS-TR-07-09*. 2007.

[21] D. Gao, M. Reiter, D. Song**.** Behavioral Distance for Intrusion Detection. *Recent Advances Intrusion Detection.* 2005.

[22] D. Gao, M. K. Reiter, D. Song. Beyond Output Voting: Detecting Compromised Replicas using Behavioral Distance. *Tech Report, CMU-CYLAB-06-019*. 2006.

[23] D. A. Grier. *When Computers Were Human*. Princeton University Press. 2005.

[24] V. Haldar, D. Chandra, M. Franz. Dynamic Taint Propagation for Java. *Annual Computer Security Applications Conference*. 2005.

[25] W. Hu, J. Hiser, D. Williams, A. Filipi, J. Davidson, D. Evans, J. Knight, A. Nguyen-Tuong, J. Rowanhill. Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation. *Virtual Execution Environments.* 2006.

[26] M. K. Joseph. Architectural Issues in Fault-Tolerant, Secure Computing Systems. *PhD Dissertation*. UCLA. 1988.

[27] J. Just, J. Reynolds, L. Clough, M. Danforth, K. Levitt, R. Maglich, J. Rowe. Learning Unknown Attacks – A Start. *Recent Advances Intrusion Detection.* 2002.

[28] G. Kc, A. Keromytis, V. Prevelakis. Countering Code-injection Attacks with Instruction Set Randomization. *ACM Computer and Communications Security*. 2003.

[29] J. C. Knight and N. Leveson. An Experimental Evaluation of the Assumption of Independence in Multi-version Programming. *IEEE Transactions on Software Engineering*, Vol 12, No 1. Jan 1986.

[30] B. Kuperman, C. Brodley, H. Ozdoganoglu, T. Vijaykumar, A. Jalote. Detection and Prevention of Stack Buffer Overflow Attacks. *Comm. of the ACM*, Nov 2005.

[31] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. *USENIX Security.* 2001.

[32] B. Littlewood, L. Strigini. Redundancy and Diversity in Security. *European Symp. on Research in Computer Security.* 2004.

[33] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *Network and Distributed System Security.* 2005.

[34] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, D. Evans. Automatically Hardening Web Applications Using Precise Tainting. *20th IFIP Information Security Conference*. 2005.

[35] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, K. Levitt. The Design and Implementation of an Intrusion Tolerant System. *Foundations of Intrusion Tolerant Systems* (OASIS). 2003.

[36] M. Ringenburg and D. Grossman. Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking. *ACM Comp. Comm. Security*. 2005.

[37] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, D. Boneh. On the Effectiveness of Address-Space Randomization. *ACM Computer and Communications Security*. 2004.

[38] A. N. Sovarel, D. Evans, N. Paul. Where's the FEEB?: The Effectiveness of Instruction Set Randomization. *USENIX Security* 2005.

[39] E. Totel, F. Majorczyk, L. Mé. COTS Diversity Intrusion Detection and Application to Web Servers. *Recent Advances in Intrusion Detection*. 2005.

[40] N. Tuck, B. Calder, and G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow**.** *Intl. Symposium on Microarchitecture*. Dec 2004.

[41] VeriTest Corporation. *WebBench 5.0.* http://www.veritest.com/benchmarks/webbench

[42] J. Xu, Z. Kalbarczyk, R. Iyer. Transparent Runtime Randomization for Security. *Symposium on Reliable and Distributed Systems*. 2003.

[43] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. *USENIX Security.* 2006.