N-Variant Systems
A Secretless Framework for Security through Diversity

Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill,
Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser
University of Virginia, Department of Computer Science
http://www.nvariant.org

Abstract

We present an architectural framework for systematically using automated diversity to provide high assurance detection and disruption for large classes of attacks. The framework executes a set of automatically diversified variants on the same inputs, and monitors their behavior to detect divergences. The benefit of this approach is that it requires an attacker to simultaneously compromise all system variants with the same input. By constructing variants with disjoint exploitation sets, we can make it impossible to carry out large classes of important attacks. In contrast to previous approaches that use automated diversity for security, our approach does not rely on keeping any secrets. In this paper, we introduce the N-variant systems framework, present a model for analyzing security properties of N-variant systems, define variations that can be used to detect attacks that involve referencing absolute memory addresses and executing injected code, and describe and present performance results from a prototype implementation.

1. Introduction

Many security researchers have noted that the current computing monoculture leaves our infrastructure vulnerable to a massive, rapid attack [70, 29, 59]. One mitigation strategy that has been proposed is to increase software diversity. By making systems appear different to attackers, diversity makes it more difficult to construct exploits and limits an attack's ability to propagate. Several techniques for automatically producing diversity have been developed including rearranging memory [8, 26, 25, 69] and randomizing the instruction set [6, 35]. All these techniques depend on keeping certain properties of the running execution secret from the attacker. Typically, these properties are determined by a secret key used to control the randomization. If the secret used to produce a given variant is compromised, an attack can be constructed that successfully attacks that variant. Pointer obfuscation techniques, memory address space randomization, and instruction set randomization have all been demonstrated to be vulnerable to remote attacks [55, 58, 64]. Further, the diversification secret may be compromised through side channels, insufficient entropy, or insider attacks.

Our work uses artificial diversity in a new way that does not depend on keeping secrets: instead of diversifying individual systems, we construct a single system containing multiple variants designed to have disjoint exploitation sets. Figure 1 illustrates our framework. We refer to the entire server as an N-variant system. The system shown is a 2-variant system, but our framework generalizes to any number of variants. The polygrapher takes input from the client and copies it to all the variants. The original server process P is replaced with the two variants, P_0 and P_1. The variants maintain the client-observable behavior of P on all normal inputs. They are, however, artificially diversified in a way that makes them behave differently on abnormal inputs that correspond to an attack of a certain class. The monitor observes the behavior of the variants to detect divergences which reveal attacks. When a divergence is detected, the monitor restarts the variants in known uncompromised states.

As a simple example, suppose P_0 and P_1 use disjoint memory spaces such that any absolute memory address that is valid in P_0 is invalid in P_1, and vice versa. Since the variants are transformed to provide the same semantics regardless of the memory space used, the behavior

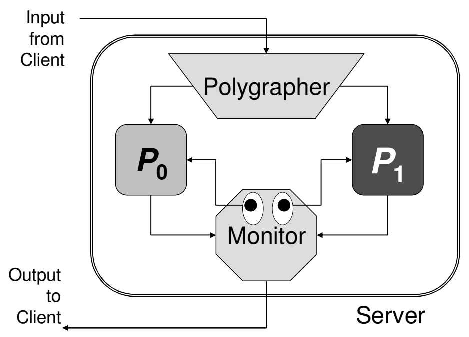

Figure 1. N-Variant System Framework.

on all normal inputs is identical (assuming deterministic behavior, which we address in Section 5). However, if an exploit uses an absolute memory address directly, it must be an invalid address on one of the two variants. The monitor can easily detect the illegal memory access on the other variant since it is detected automatically by the operating system. When monitoring is done at the system call level, as in our prototype implementation, the attack is detected before any external state is modified or output is returned to the attacker.

The key insight behind our approach is that in order for an attacker to exploit a vulnerability in *P*, a pathway must exist on one of the variants that exploits the vulnerability without producing detectably anomalous behavior on any of the other variants. If no such pathway exists, there is no way for the attacker to construct a successful attack, even if the attacker has complete knowledge of the variants. Removing the need to keep secrets means we do not need to be concerned with probing or guessing attacks, or even with attacks that take advantage of insider information.

Our key contributions are:

1. Introducing the N-variant systems framework that uses automated diversity techniques to provide high assurance security properties without needing to keep any secrets.
2. Developing a model for reasoning about N-variant systems including the definition of the normal equivalence and detection properties used to prove security properties of an ideal N-variant system (Section 3).
3. Identifying two example techniques for providing variation in N-variant systems: the memory address partitioning technique (introduced above) that detects attacks that involve absolute memory references and the instruction tagging technique that detects attempts to execute injected code (Section 4).
4. Describing a Linux kernel system implementation and analyzing its performance (Section 5).

In this paper we do not address recovery but consider it to be a successful outcome when our system transforms an attack that could compromise privacy and integrity into an attack that at worst causes a service shutdown that denies service to legitimate users. It has not escaped our attention, however, that examining differences between the states of the two variants at the point when an attack is detected provides some intriguing recovery possibilities. Section 6 speculates on these opportunities and other possible extensions to our work.

2. Related Work

There has been extensive work done on eliminating security vulnerabilities and mitigating attacks. Here, we briefly describe previous work on other types of defenses and automated diversity, and summarize related work on redundant processing and design diversity frameworks.

Other defenses. Many of the specific vulnerabilities we address have well known elimination, mitigation and disruption techniques. Buffer overflows have been widely studied and numerous defenses have been developed including static analysis to detect and eliminate the vulnerabilities [66, 67, 39, 23], program transformation and dynamic detection techniques [19, 5, 30, 45, 49, 57] and hardware modifications [38, 40, 41, 64]. There have also been several defenses proposed for string format vulnerabilities [56, 20, 63, 47]. Some of these techniques can mitigate specific classes of vulnerabilities with less expense and performance overhead than is required for our approach. Specific defenses, however, only prevent a limited class of specific vulnerabilities. Our approach is more general; it can mitigate all attacks that depend on particular functionality such as injecting code or accessing absolute addresses.

More general defenses have been proposed for some attack classes. For example, no execute pages (as provided by OpenBSD's W^X and Windows XP Service Pack 2) prevent many code injection attacks [2], dynamic taint analysis tracks information flow to identify memory corruption attacks [43], and control-flow integrity can detect attacks that corrupt an application to follow invalid execution paths [1]. Although these are promising approaches, they are limited to particular attack classes. Our framework is more general in the sense that we can construct defense against any attacker capability that can be varied across variants in an N-variant system.

Automated diversity. Automated diversity applies transformations to software to increase the difficulty an attacker will face in exploiting a security vulnerability in that software. Numerous transformation techniques have been proposed including rearranging memory [26, 8, 69, 25], randomizing system calls [17], and randomizing the instruction set [6, 35]. Our work is complementary to work on producing diversity; we can incorporate many different sources of variation as long as variants are constructed carefully to ensure the disjointedness required by our framework. A major advantage of the N-variant systems approach is that we do not rely on secrets for our security properties. This means we can employ diversification techniques with low entropy, so long as the transformations are able to produce variants with disjoint exploitation sets. Holland, Lim, and Seltzer propose many low entropy diversification techniques including number representations, register sets, stack direction, and memory layout [31]. In addition, our approach is not vulnerable to the type of secret-breaking attacks that have been demonstrated against secret-based diversity defenses [55, 58, 64].

O'Donnell and Sethu studied techniques for distributing diversity at the level of different software packages in a network to mitigate spreading attacks [44]. This can limit the ability of a worm exploiting a vulnerability present in only one of the software packages to spread on a network. Unlike our approach, however, even at the network level an attacker who discovers vulnerabilities in more than one of the software packages can exploit each of them independently.

Redundant execution. The idea of using redundant program executions for various purposes is not a new one. Architectures involving replicated processes have been proposed as a means to aid debugging, to provide fault tolerance, to improve dependability, and more recently, to harden vulnerable services against attacks.

The earliest work to consider running multiple variants of a process of which we are aware is Knowlton's 1968 paper [37] on a variant technique for detecting and localizing programming errors. It proposed simultaneously executing two programs which were logically equivalent but assembled differently by breaking the code into fragments, and then reordering the code fragments and data segments with appropriate jump instructions inserted between code fragments to preserve the original program semantics. The CPU could run in a checking mode that would execute both programs in parallel and verify that they execute semantically equivalent instructions. The variants they used did not provide any guarantees, but provided a high probability of detecting many programming errors such as out-of-range control transfers and wild memory fetches.

More recently, 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]. Since there is no guarantee that there will not be references at the same absolute locations, or reachable through the same relative offsets, their approach can provide only probabilistic expectations that a memory corruption will be detected by producing noticeably different behavior on the variants. Their goals were to enhance reliability and availability, rather than to detect and resist attacks. Consequently, when variations diverge in their framework, they allow the agreeing replicas to continue based on the assumption that the cause of the divergence in the other replicas was due a memory flaw rather than a successful attack. Their replication framework only handles processes whose I/O is through standard in/out, and only a limited number of system calls are caught in user space to ensure all replicas see the same values. Since monitoring is only on the standard output, a compromised replica could be successfully performing an attack and, as long as it does not fill up its standard out buffer, the monitor would not notice. The key difference between their approach and ours, is that their approach is probabilistic whereas our variants are constructed to guarantee disjointedness with respect to some property, and thereby can provide guarantees of invulnerability to particular attack classes. A possible extension to our work would consider variations providing probabilistic protection, such as the heap randomization technique they used, to deal with attack classes for which disjointedness is infeasible.

Redundant processing of the same instruction stream by multiple processors has been used as a way to provide fault-tolerance by Stratus [68] and Tandem [32] computers. For example, Integrity S2 used triple redundancy in hardware with three synchronized identical processors executing the same instructions [32]. A majority voter selects the majority output from the three processors, and a vote analyzer compares the outputs to activate a failure mode when a divergence is detected. This type of redundancy provides resilience to hardware faults, but no protection against malicious attacks that exploit vulnerabilities in the software, which is identical on all three processors. Slipstream processors are an interesting variation of this, where two redundant versions of the instruction stream execute, but instructions that are dynamically determined to be likely to be unnecessary are removed from the first stream which executes speculatively [60]. The second stream executes behind the first stream, and the processor detects inconsistencies between the two executions. These deviations either indicate false predications about unnecessary computations (such as a mispredicted branch) or hardware faults.

The distributed systems community has used active replication to achieve fault tolerance [9, 10, 16, 18, 50]. With active replication, all replicas are running the same software and process the same requests. Unlike our approach, however, active replication does nothing to hide design flaws in the software since all replicas are running the same software. To mitigate this problem, Schneider and Zhou have suggested proactive diversity, a technique for periodically randomizing replicas to justify the assumption that server replicas fail independently and to limit the window of vulnerability in which replicas are susceptible to the same exploit [51]. Active replication and N-variant systems are complementary approaches. Combining them can provide the benefits of both approaches with the overhead and costs associated with either approach independently.

Design diversity frameworks. The name *N-variant systems* is inspired by, but fundamentally different from, the technique known as *N-version programming* [3, 14]. The N-version programming method uses several independent development groups to develop different implementations of the same specification with the hope that different development groups will produce versions without common faults. The use of N-version programming to help with system security was proposed by Joseph [33]. He analyzed design diversity as manifest in N-version programming to see whether it could defeat certain attacks and developed an analogy between faults in computing systems that might affect reliability and vulnerabilities in computer systems that might affect security. He argued that N-version programming techniques might allow vulnerabilities to be masked. However, N-version programming provides no guarantee that the versions produced by different teams will not have common flaws. Indeed, experiments have shown that common flaws in implementations do occur [36]. In our work, program variants are created by mechanical transformations engineered specifically to differ in particular ways that enable attack detection. In addition, our variants are produced mechanically, so the cost of multiple development teams is avoided.

Three recent projects [46, 62, 28] have explored using design diversity in architectures similar to the one we propose here in which the outputs or behaviors of two diverse implementations of the same service (e.g., HTTP servers Apache on Linux and IIS on Windows) are compared and differences above a set threshold indicate a likely attack. The key difference between those projects and our work is that whereas they use diverse available implementations of the same service, we use techniques to artificially produce specific kinds of variation. The HACQIT project [34, 46] deployed two COTS web servers (IIS running on Windows and Apache running on Linux) in an architecture where a third computer forwarded all requests to both servers and compared their responses. A divergence was detected when the HTTP status code differed, hence divergences that caused the servers to modify external state differently or produce different output pages would not be detected. The system described by Totel, Majorczyk, and Mé extended this idea to compare the actual web page responses of the two servers [62]. Since different servers do not produce exactly the same output on all non-attack requests because of nondeterminism, design differences in the servers, and host-specific properties, they developed an algorithm that compares a set of server responses to determine which divergences are likely to correspond to attacks and which are benign. The system proposed by Gao, Reiter, and Song [28] deployed multiple servers in a similar way, but monitored their behavior using a distance metric that examined the sequence of system calls each server made to determine when the server behaviors diverged beyond a threshold amount.

All of these systems use multiple available implementations of the same service running on isolated machines and compare the output or aspects of the behavior to notice when the servers diverged. They differ in their system architectures and in how divergences are recognized. The primary advantage of our work over these approaches is the level of assurance automated diversity and monitoring can provide over design diversity. Because our system takes advantage of knowing exactly how the variants differ, we can make security claims about large attack classes. With design diversity, security claims depend on the implementations being sufficiently different to diverge noticeably on the attack (and functionality claims depend on the behaviors being sufficiently similar not exceed the divergence threshold on non-attack inputs). In addition, these approaches can be used only when diverse implementations of the same service are available. For HTTP servers, this is the case, but for custom servers the costs of producing a diverse implementation are prohibitive in most cases. Further, even though many HTTP servers exist, most advanced websites take advantages of server-specific functionality (such as server-side includes provided by Apache), so would not work on an alternate server. Design diversity approaches offer the advantage that they may be able to detect attacks that are at the level of application semantics rather than low-level memory corruption or code injection attacks that are better detected by artificial diversity. In Section 6, we consider possible extensions to our work that would combine both approaches to provide defenses against both types of attacks.

3. Model

Our goal is to show that for all attacks in a particular attack class, if one variant is compromised by a given attack, another variant must exhibit divergent behavior that is detected by the monitor. To show this, we develop a model of execution for an N-variant system and define two properties the variant processes must maintain to provide a detection guarantee.

We can view an execution as a possibly infinite sequence of states: $[S_0, S_1, \ldots]$. In an N-variant system, the state of the system can be represented using a tuple of the states of the variants (for simplicity, this argument assumes the polygrapher and monitor are stateless; in our implementation, they do maintain some state but we ignore that in this presentation). Hence, an execution of an N-variant system is a sequence of state-tuples where $S_{t,v}$ represents the state of variant v at step t: $[<S_{0,0}, S_{0,1}, \ldots S_{0,N-1}>, <S_{1,0}, S_{1,1}, \ldots S_{1,N-1}>, \ldots]$.

Because of the artificial variation, the concrete state of each variant differs. Each variant has a *canonicalization function*, C_v, that maps its state to a canonical state that matches the corresponding state for the original process. For example, if the variation alters memory addresses, the mapping function would need to map the variant's altered addresses to canonical addresses. Under normal execution, at every execution step the canonicalized states of all variants are identical to the original program state:

$$\forall t \geq 0, 0 \leq v < N, 0 \leq w < N: \\ C_v\,(S_{t,\,v}) = C_w\,\,(S_{t,\,w}) = S_t.$$

Each variant has a *transition function*, T_v, that takes a state and an input and produces the next state. The original program, P, also has a transition function, T. The set of possible transitions can be partitioned into *consistent transitions* and *aberrant transitions*. Consistent transitions take the system from one normal state to another normal state; aberrant transitions take the system from a normal state to a compromised state. An attack is successful if it produces an aberrant transition without detection. Our goal is to detect all aberrant transitions.

We partition possible variant states into three sets: *normal*, *compromised*, and *alarm*. A variant in a normal state is behaving as intended. A variant in a compromised state has been successfully compromised by a malicious attack. A variant in an alarm state is anomalous in a way that is detectable by the monitor. We aim to guarantee that the N-variant system never enters a state-tuple that contains one or more variants in compromised states without any variants in alarm states. To establish this we need two properties: *normal equivalence* and *detection*.

Normal equivalence. The normal equivalence property is satisfied if the N-variant system synchronizes the states of all variants. That is, whenever all variants are in normal states, they must be in states that correspond to the same canonical state. For security, it is sufficient to show the variants remain in equivalent states. For correctness, we would also like to know the canonical state of each of the variants is equivalent to the state of the original process.

We can prove the normal equivalence property statically using induction:

1. Show that initially all variants are in the same canonical state: $\forall\ 0 \leq v < N$: $\mathrm{C}_i\,(S_{0,\,v}) = S_0$.
2. Show that every normal transition preserves the equivalence when the system is initially in a normal state:

$$\forall S \in \mathit{Normal}, 0 \leq v < N, S_v \\ \text{where } C_v\,(S_v) = S, p \in \mathit{Inputs}: \\ C_v\,(T_v\,(S_v, p)) = T\,(S, p).$$

Alternatively, we can establish it dynamically by examining the states of the variants and using the canonicalization function to check the variants are in equivalent states after every step. In practice, neither a full static proof nor a complete dynamic comparison is likely to be feasible for real systems. Instead, we argue that our implementation provides a limited form of normal equivalence using a combination of static argument and limited dynamic comparison, as we discuss in Section 5.

Detection. The detection property guarantees that all attacks in a certain class will be detected by the N-variant system as long as the normal equivalence property is satisfied. To establish the detection property, we need to know that any input that causes one variant to enter a compromised state must also cause some other variant to enter an alarm state. Because of the normal equivalence property, we can assume the variants all are in equivalent states before processing this input. Thus, we need to show:

$$\forall S \in \mathit{Normal}, 0\ \leq v < N,\ S_v \text{ where } C_v\,(S_v) = S, \\ \forall p \in \mathit{Inputs}: \\ T_v\,(S_v, p) \in \mathit{Compromised} \\ \exists w \text{ such that } T_w\,(S_w, p) \in \mathit{Alarm} \text{ and } C_w\,(S_w) = S$$

If the detection property is established, we know that whenever one of the variants enters a compromised state, one of the variants must enter an alarm state. An ideal monitor would instantly detect the alarm state and prevent all the other variants from continuing. This would guarantee that the system never operates in a state in which any variant is compromised.

In practice, building such a monitor is impossible since we cannot keep the variants perfectly synchronized or detect alarm states instantly. However, we can approximate this behavior by delaying any external effects (including responses to the client) until all variants have passed a critical point. This keeps the variants loosely synchronized, and approximates the behavior of instantly terminating all other variants when one variant encounters an alarm state. It leaves open the possibility that a compromised variant could corrupt the state of other parts of the system (including the monitor and other variants) before the alarm state is detected. An implementation must use isolation mechanisms to limit this possibility.

4. Variations

Our framework works with any diversification technique that produces variants different enough to provide detection of a class of attack but similar enough to establish a normal equivalence property. The variation used to diversify the variants determines the attack class the N-variant system can detect. The detection property is defined by the class of attack we detect, so we will consider attack classes, such as attacks that involve executing injected instructions, rather than vulnerability classes such as buffer overflow vulnerabilities.

Next, we describe two variations we have implemented: address space partitioning and instruction set tagging. We argue (informally) that they satisfy both the normal equivalence property and the detection condition for important classes of attacks. The framework is general enough to support many other possible variations, which we plan to explore in future work. Other possible variations that could provide useful security properties include varying memory organization, file naming, scheduling, system calls, calling conventions, configuration properties, and the root user id.

4.1 Address Space Partitioning

The Introduction described an example variation where the address space is partitioned between two variants to disrupt attacks that rely on absolute addresses. This simple variation does not prevent all memory corruption attacks since some attacks depend only on relative addressing, but it does prevent all memory corruption attacks that involve direct references to absolute addresses. Several common vulnerabilities including format string [56, 54], integer overflow, and double-free [24] may allow an attacker to overwrite an absolute location in the target's address space. This opportunity can be exploited to give an attacker control of a process, for example, by modifying the Global Offset Table [24] or the `.dtors` segment of an ELF executable [48]. Regardless of the vulnerability exploited and the targeted data structure, if the attack depends on loading or storing to an absolute address it will be detected by our partitioning variants. Since the variation alters absolute addresses, it is necessary that the original program does not depend on actual memory addresses (for example, using the value of a pointer directly in a decision). Although it is easy to construct programs that do not satisfy this property, most sensible programs should not depend on actual memory addresses.

Detection. Suppose P_0 only uses addresses whose high bit is 0 and P_1 only uses addresses whose high bit is 1. We can map the normal state of P_0 and P_1 to equivalent states using the identity function for C_0 and a function that flips the high bit of all memory addresses for C_1 (to map onto the actual addresses used by P, more complex mapping functions may be needed). The transition functions, T_0 and T_1 are identical; the generated code is what makes things different since a different address will be referenced in the generated code for any absolute address reference. If an attack involves referencing an absolute address, the attacker must choose an address whose high bit is either a 0 or 1. If it is a 0, then P_0 may transition to a compromised state, but P_1 will transition to an alarm state when it attempts to access a memory address outside P_1's address space. In Unix systems, this alarm state is detected by the operating system as a segmentation fault. Conversely, if the attacker chooses an address whose high bit is 1, P_1 may be compromised but P_0 must enter an alarm state. In either case, the monitor detects the compromise and prevents any external state modifications including output transmission to the client.

Our detection argument relies on the assumption that the attacker must construct the entire address directly. For most scenarios, this assumption is likely to be valid. For certain vulnerabilities on platforms that are not byte-aligned, however, it may not be. If the attacker is able to overwrite an existing address in the program without overwriting the high bit, the attacker may be able to construct an address that is valid in both variants. Similarly, if an attacker can corrupt a value that is subsequently used with a transformed absolute address in an address calculation, the detection property is violated. As with relative attacks, this indirect memory attacks would not be detected by this variation.

Normal equivalence. We have two options for establishing the normal equivalence property: we can check it dynamically using the monitor, or we can prove it statically by analyzing the variants. A pure dynamic approach is attractive for security assurance because of its simplicity but impractical for performance-critical servers. The monitor would need to implement C_0 and C_1 and compute the canonical states of each variant at the end of each instruction execution. If the states match, normal equivalence is satisfied. In practice, however, this approach is likely to be prohibitively expensive. We can optimize the check by limiting the comparison to the subset of the execution state that may have changed and only checking the state after particular instructions, but the overhead of checking the states of the variants after every step will still be unacceptable for most services.

The static approach requires proving that for every possible normal state, all normal transitions result in equivalent states on the two variants. This property requires that no instruction in P can distinguish between the two variants. For example, if there were a conditional jump in P that depended on the high bit of the address of some variable, P_0 and P_1 would end up in different states after executing that instruction. An attacker could take advantage of such an opportunity to get the variants in different states such that an input that transitions P_0 to a compromised state does not cause P_1 to reach an alarm state. For example, if the divergence is used to put P_0 in a state where the next client input will be passed to a vulnerable string format call, but the next client input to P_1 is processed harmlessly by some other code, an attacker may be able to successfully compromise the N-variant system. A divergence could also occur if some part of the system is nondeterministic, and the operating environment does not eliminate this nondeterminism (see Section 5). Finally, if P is vulnerable to some other class of attack, such as code injection, an attacker may be able to alter the transition functions T_0 and T_1 in a way that allows the memory corruption attack to be exploited differently on the two variants to avoid detection (of course, an attacker who can inject code can already compromise the system in arbitrary ways).

In practice, it will not usually be possible to completely establish normal equivalence statically for real systems but rather we will use a combination of static and dynamic arguments, along with assumptions about the target service. A combination of static and dynamic techniques for checking equivalence may be able to provide higher assurance without the overhead necessary for full dynamic equivalence checking. Our prototype implementation checks equivalence dynamically at the level of system calls, but relies on informal static arguments to establish equivalence between them.

Implementation. To partition the address space, we vary the location of the application data and code segments. The memory addresses used by P_0 and P_1 are disjoint: any data address that is valid for P_0 is invalid for P_1, and vice versa. We use a linker script to create the two variants. Each variant loads both the code and data segments of the variants at different starting addresses from the other variant. To ensure that their sets of valid data memory addresses are disjoint, we use ulimit to limit the size of P_0's data segment so it cannot grow to overlap P_1's address space.

4.2 Instruction Set Tagging

Whereas partitioning the memory address space disrupts a class of memory corruption attacks, partitioning the instruction set disrupts code injection attacks. There are several possible ways to partition the instruction set.

One possibility would be to execute the variants on different processors, for example one variant could run on an x86 and the other on a PowerPC. Establishing the security of such an approach would be very difficult, however. To obtain the normal equivalence property we would need a way of mapping the concrete states of the different machines to a common state. Worse, to obtain the detection property, we would need to prove that no string of bits that corresponds to a successful malicious attack on one instruction set and a valid instruction sequence on the other instruction set. Although it is likely that most sequences of malicious x86 instructions contain an invalid PowerPC instruction, it is certainly possible for attackers to design instruction sequences that are valid on both platforms (although we are not aware of any programs that do this for the x86 and PowerPC, Sjoerd Mullender and Robbert van Renesse won the 1984 International Obfuscated C Code Contest with an entry that replaced main with an array of bytes that was valid machine code for both the Vax and PDP-11 but executed differently on each platform [35]).

Instead, we use a single instruction set but prepend a variant-specific tag to all instructions. The diversification transformation takes P and inserts the appropriate tag bit before each instruction to produce each variant.

Detection. The variation detects any attack that involves executing injected code, as long as the mechanism used to inject code involves injecting complete instructions. If memory is bit-addressable, an attacker could overwrite just the part of the instruction after the tag bit, thereby changing an existing instruction while preserving the original tag bit. If the attacker can inject the intended code in memory, and then have the program execute code already in the executable that transforms the injected memory (for example, by XORing each byte with a constant that is different in the two variants), then it is conceivable that an attacker could execute an indirect code injection attack where the code is transformed differently on the two variants before executing to evade the detection property. For all known realistic code injection attacks, neither of these is considered a serious risk.

Normal equivalence. The only difference between the two variants is the instruction tag, which has no effect on instruction execution. The variants could diverge, however, if the program examines its own instructions and makes decisions that depend on the tag. It is unlikely that a non-malicious program would do this. As with the memory partitioning, if the instruction tags are visible to the executing process an attacker might be able to make them execute code that depends on the instruction tags to cause the variants to diverge before launching the code injection attack on one of the variants. To prevent this, we need to store the tagged instructions in memory that is not readable to the executing process and remove the tags before those instructions reach the processor.

Implementation. To implement instruction set tagging, we use a combination of binary rewriting before execution and software dynamic translation during execution. We use Diablo [61, 22], a retargetable binary rewriting framework, to insert the tags. Diablo provides mechanisms for modifying an x86 binary in ELF format. We use these to insert the appropriate variant-specific tag before every instruction. For simplicity, we use a full byte tag even though a single bit would suffice for two variants. There is no need to keep the tags secret, just that they are different; we use 10101010 and 01010101 for the *A* and *B* variant tags.

At run-time, the tags are checked and removed before instructions reach the processor. This is done using Strata, a software dynamic translation tool [52, 53]. Strata and other software dynamic translators [4, 11] have demonstrated that it is possible to implement software dynamic translation without unreasonable performance penalty. In our experiments (Section 5), Strata's overhead is only a few percent. The Strata VM mediates application execution by examining and translating instructions before they execute on the host CPU. Translated instructions are placed in the fragment cache and then executed directly on the host CPU. Before switching to the application code, the Strata VM uses `mprotect` to protect critical data structures including the fragment cache from being overwritten by the application. At the end of a translated block, Strata appends trampoline code that will switch execution back to the Strata VM, passing in the next application PC so that the next fragment can be translated and execution will continue. We implement the instruction set tagging by extending Strata's instruction fetch module. The modified instruction fetch module checks that the fetched instruction has the correct tag for this variant; if it does not, a security violation is detected and execution terminates. Otherwise, it removes the instruction tag before placing the actual instruction in the fragment cache. The code executing on the host processor contains no tags and can execute normally.

5. Framework Implementation

Implementing an N-variant system involves generating variants such as those described in Section 4 as well as implementing the polygrapher and monitor. The trusted computing base comprises the polygrapher, monitor and mechanisms used to produce the variants, as well as any operating system functionality that is common across the variants. An overriding constraint on our design is that it be fully automated. Any technique that requires manual modification of the server to create variants or application-specific monitoring would impose too large a deployment burden to be used widely. To enable rapid development, our implementations are entirely in software. Hardware implementations would have security and performance advantages, especially in monitoring the instruction tags. Furthermore, placing monitoring as close as possible to the processor eliminates the risk that an attacker can exploit a vulnerability in the monitoring mechanism to inject instructions between the enforcement mechanism and the processor.

The design space for N-variant systems implementations presents a challenging trade-off between isolation of the variants, polygrapher, and monitor and the need to keep the variant processes synchronized enough to establish the normal equivalence property. The other main design decision is the granularity of the monitoring. Ideally, the complete state of each variant would be inspected after each instruction. For performance reasons, however, we can only observe aspects of the state at key execution points. Incomplete monitoring means that an attacker may be able to exploit a different vulnerability in the server to violate the normal equivalence property, thereby enabling an attack that would have otherwise been detected to be carried out without detection. For example, an attacker could exploit a race condition in the server to make the variants diverge in ways that are not detected by the monitor. Once the variants have diverged, the attacker can construct an input that exploits the vulnerability in one variant, but does not produce the detected alarm state on the other variants because they started from different states.

In our first proof-of-concept implementation, described in Section 5.1, we emphasized isolation and executed the variants on separate machines. This meant that any nondeterminism in the server program or aspects of the host state visible to the server program that differed between the machines could be exploited by an attacker to cause the processes to diverge and then allow a successful attack. It also meant the monitor only observed the outputs produced by the two variants that would be sent over the network. This enabled certain attacks to be detected, but meant a motivated attacker could cause the states to diverge in ways that were not visible from the output (such as corrupting server data) but still achieved the attacker's goals.

Our experience with this implementation led us to conclude that a general N-variant systems framework needed closer integration of the variant processes to prevent arbitrary divergences. We developed such a framework as a kernel modification that allows multiple variants to run on the same platform and normal equivalence to be established at system call granularity. This eliminates most causes of nondeterminism and improves the performance of the overall system. Section 5.2 describes our Linux kernel implementation, and Section 5.3 presents performance results running Apache variants on our system.

5.1 Proof-of-Concept Implementation

In our proof-of-concept implementation, the variants are isolated on separate machines and the polygrapher and monitor are both implemented by the nvd process running on its own machine. We used our implementation to protect both a toy server we constructed and Apache. In order for our approach to work in practice it is essential that no manual modification to the server source code is necessary. Hence, each server variant must execute in a context where it appears to be interacting normally with the client. We accomplish this by using divert sockets to give each variant the illusion that it is interacting directly with a normal client. To implement the polygrapher we use ipfw, a firewall implementation for FreeBSD [27] with a rule that redirects packets on port 80 (HTTP server) to our nvd process which adjusts the TCP sequence numbers to be consistent with the variant's numbering. Instead of sending responses directly to the client, the variant's responses are diverted back to nvd, which buffers the responses from all of the variants. The responses from P_0 are transmitted back to the client only if a comparably long response is also received from the other variants. Hence, if any variant crashes on a client input, the response is never sent to the client and nvd restarts the server in a known uncompromised state.

We tested our system by using it to protect a toy server we constructed with a simple vulnerability and Apache, and attempted to compromise those servers using previously known exploits as well as constructed exploits designed to attack a particular variant. Exploit testing does not provide any guarantees of the security of our system, of course, but it does demonstrate that the correct behavior happens under the tested conditions to increase our confidence in our approach and implementation. Our toy server contained a contrived format string vulnerability, and we developed an exploit that used that vulnerability to write to an arbitrary memory address. The exploit could be customized to work against either variation, but against the N-variant system both versions would lead to one of the variants crashing. The monitor detects the crash and prevents compromised outputs from reaching the client. We also tested an Apache server containing a vulnerable OpenSSL implementation (before 0.9.6e) that contained a buffer overflow vulnerability that a remote attacker could exploit to inject code [13]. When instruction set tagging is used, the exploit is disrupted since it does not contain the proper instruction tags in the injected code.

We also conducted some performance measurements on our 2-variant system with memory address partitioning. The average response latency for HTTP requests increased from 0.2ms for the unmodified server to 2.9ms for the 2-variant system.

The proof-of-concept implementation validated the N-variant systems framework concept, but did not provide a practical or secure implementation for realistic services. Due to isolation of the variants, various non-attack inputs could lead to divergences between the variants caused by differences between the hosts. For example, if the output web page includes a time stamp or host IP address, these would differ between the variants. This means false positives could occur when the monitor observes differences between the outputs for normal requests. Furthermore, a motivated attacker could take advantage of any of these differences to construct an attack that would compromise one of the variants without leading to a detected divergence.

5.2 Kernel Implementation

The difficulties in eliminating nondeterminism and providing finer grain monitoring with the isolated implementation, as well as its performance results, convinced us to develop a kernel implementation of the framework by modifying the Linux 2.6.11 kernel. In this implementation, all the variants run on the same platform, along with the polygrapher and monitor. We rely on existing operating system mechanisms to provide isolation between the variants, which execute as separate processes.

We modified the kernel data structures to keep track of variant processes and implemented wrappers around system calls. These wrappers implement the polygraphing functionality by wrapping input system calls so that when both variants make the same input system call, the actual input operation is performed once and the same data is sent to all variants. They provide the monitoring functionality by checking that all variants make the same call with equivalent arguments before making the actual system call.

This system call sharing approach removes nearly all of the causes of nondeterminism that were problematic in the proof-of-concept implementation. By wrapping the system calls, we ensure that variants receive identical results from all system calls. The remaining cause of nondeterminism is due to scheduling differences, in particular in handling signals. We discuss these limitations in Section 6.

In order to bring an N-variant system into execution we created two new system calls: n_variant_fork, and n_variant_execve. The program uses these system calls similarly to the way a shell uses fork/execve to bring processes into execution. The n_variant_fork system call forks off the variants, however instead of creating a single child process it creates one process per variant. The variants then proceed to call n_variant_execve, which will cause each of the variants to execute their own diversified binary of the server. Note that our approach requires no modification of an existing binary to execute it within an N-variant system; we simply invoke a shell command that takes the pathnames of variant binaries as parameters and executes n_variant_execve.

Next, we provide details on the system call wrappers that implement the polygraphing and monitoring. The Linux 2.6.11 kernel provides 267 system calls. We generalize them into three categories based on the type of wrapper they need: shared system calls, reflective system calls, and dangerous system calls.

Shared System Calls. For system calls that interact with external state, including I/O system calls, the wrapper checks that all variants make equivalent calls, makes the actual call once, and sends the output to all variants, copying data into each of the variants address space if necessary. Figure 2 shows pseudocode for a shared call, in this case the read system call. The actual wrappers are generated using a set of preprocessor macros we developed to avoid duplicating code. The first if statement checks whether this process is part of an N-variant system. If not, the system call proceeds normally. Hence, a single platform can run both normal and

```
ssize_t sys_read(int fd, const void *buf, size_t count) {
  if (!hasSibling (current)) { make system call normally } // not a variant process
  else {
    record that this variant process entered call
    if (!inSystemCall (current->sibling)) { // this variant is first
      save parameters
      sleep // sibling will wake us up
      get result and copy *buf data back into address space
      return result;
    } else if (currentSystemCall (current->sibling) == SYS_READ) { // this variant is second, sibling waiting
      if (parameters match) { // what it means to “match” depends on variation and system call
        perform system call
        save result and data in kernel buffer
        wake up sibling
        return result;
      } else { DIVERGENCE ERROR! } // sibling used different parameters
    } else { DIVERGENCE ERROR! } } } // sibling is in a different system call
```

Figure 2. Typical shared system call wrapper.

N-variant processes. If the process is a variant process, it records that it has entered this system call and checks if its sibling variant has already entered a system call. If it has not, it saves the parameters and sleeps until the other variant wakes it up. Otherwise, it checks that the system call and its parameters match those used by the first variant to make the system call. If they match, the actual system call is made. The result is copied into a kernel buffer, and the sibling variant process (which reached this system call first and went to sleep) is awoken. The sibling process copies the result from the kernel buffer back into its address space and continues execution.

Reflective System Calls. We consider any system call that observes or modifies properties of the process itself a *reflective* system call. For these calls, we need to ensure that all observations always return the same value regardless of which variant reaches the call first, and that all modifications to process properties are done equivalently on all variants. For observation-only reflective calls, such as getpid, we check that all variants make the same call, and then just make the call once for variant 0 and send the same result to all variants. This is done using wrappers similar to those for shared system calls, except instead of just allowing the last variant that reaches the call to make the actual system call we need to make sure that each time a reflective call is reached, it is executed for the same process.

Another issue is raised by the system calls that create child processes (sys_fork, sys_vfork, and sys_clone). The wrappers for these calls must coordinate each variant's fork and set up all the child processes as a child N-variant system before any of the children are placed on the run queue. These system calls return the child process' PID. We ensure that all the parents in the N-variant system get the same PID (the PID of variant 0's child), as with the process observation system calls.

The other type of reflective system call acts on the process itself. These system calls often take parameters given by the reflective observation system calls. In this case, we make sure they make the same call with the same parameters, but alter the parameters accordingly for each variant. For example, sys_wait4 takes a PID as an input. Each of the variants will call sys_wait4 with the same PID because they were all given the same child PID when they called sys_fork (as was required to maintain normal equivalence). However, each variant needs to clean up its corresponding child process within the child system. The wrapper for sys_wait4 modifies the PID value passed in and makes the appropriate call for each variant with its corresponding child PID. Similar issues arise with sys_kill, sys_tkill, and sys_waitpid.

Finally, we have to deal with two system calls that terminate a process: sys_exit and sys_exit_group. A terminating process does not necessarily go through these system calls, since it may terminate by crashing. To ensure that we capture all process termination events in an N-variant system we added a monitor inside the do_exit function within the kernel which is the last function all terminating processes execute. This way, if a process receives a signal and exits without going through a system call, we will still observe this and can terminate the other variants.

Dangerous System Calls. Certain calls would allow processes to break assumptions on which we rely. For example, if the process uses the execve system to run a new executable, this will escape the N-variant protections unless we can ensure that each variant executes a different executable that is diversified appropriately. Since it is unlikely we can establish this property, the execve wrapper just disables the system call and returns an error code. This did not pose problems for Apache, but might for other applications.

Other examples of dangerous system calls are those for memory mapping (old_mmap, sys_mmap2) which map a portion of a file into a process' address space. After a file is mapped into an address space, memory reads and writes are analogous to reads and writes from the file. This would allow an attacker to compromise one variant, and then use the compromised variant to alter the state of the uncompromised variants through the shared memory without detection, since no system call is necessary. Since many server applications (including Apache) use memory mapping, simply blocking these system calls is not an option. Instead, we place restrictions on them to allow only the MAP_ANONYMOUS and MAP_PRIVATE options with all permissions and to permit MAP_SHARED mappings as long as write permissions are not requested. This eliminates the communication channel between the variants, allowing memory mapping to be used safely by the variants. Apache runs even with these restrictions since it does not use other forms of memory mapping, but other solutions would be needed to support all services.

5.3 Performance

Table 1 summarizes our performance results. We measured the throughput and latency of our system using WebBench 5.0 [65], a web server benchmark using a variety of static web page requests. We ran two sets of

Configuration		1	2	3	4	5	6
Description		Unmodified Apache, unmodified kernel	Unmodified Apache, N-variant kernel	2-variant system, address partitioning	Apache running under Strata	Apache with instruction tags	2-variant system, instruction tags
Unsaturated	Throughput (MB/s)	2.36	2.32	2.04	2.27	2.25	1.80
	Latency (ms)	2.35	2.40	2.77	2.42	2.46	3.02
Saturated	Throughput (MB/s)	9.70	9.59	5.06	8.54	8.30	3.55
	Latency (ms)	17.65	17.80	34.20	20.30	20.58	48.30

Table 1. Performance Results.

experiments measuring the performance of our Apache server under unsaturated and saturated load conditions. In both sets, there was a single 2.2GHz Pentium 4 server machine with 1GB RAM running Fedora Core 3 (2.6.11 kernel) in the six different configurations shown in Table 1. For the first set of experiences, we used a single client machine running one WebBench client engine. For the load experiments, we saturated our server using six clients each running five WebBench client engines connected to the same networks switch as the server.

Configuration 1 is the baseline configuration: regular apache running on an unmodified kernel. Configuration 2 shows the overhead of the N-variant kernel on a normal process. In our experiments, it was negligible; this is unsurprising since the overhead is only a simple comparison at the beginning of each wrapped system call. Configuration 3 is a 2-variant system running in our N-variant framework where the two variants differ in the address spaces according to the partitioning scheme described in Section 4.1. For the unloaded server, the latency observed by the client increases by 17.6%. For the loaded server, the throughput decreases by 48% and the latency nearly doubles compared to the baseline configuration. 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 the 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 approximately halving of throughput reflects the redundant computation required to run two variants.

The instruction tagging variation is more expensive because of the added cost of removing and checking the instruction tags. Configuration 4 shows the performance of Apache running on the normal kernel under Strata with no transformation. The overhead imposed by Strata reduces throughput by about 10%. The Strata overhead is relatively low because once a code fragment is in the fragment cache it does not need to be translated again the next time it executes. Adding the instruction tagging (Configuration 5) has minimal impact on throughput and latency. Configuration 6 shows the performance of a 2-variant system where the variants are running under Strata with instruction tag variation. The performance impact is more than it was in Configuration 3 because of the additional CPU workload imposed by the instruction tags. For the unloaded server, the latency increases 28% over the baseline configuration; for the saturated server, the throughput is 37% of the unmodified server's throughput.

Our results indicate that for I/O bound services, N-variant systems where the variation can be achieved with reasonable performance overhead, especially for variations such as the address space partitioning where little additional work is needed at run-time. We anticipate there being many other interesting variations of this type, such as file renaming, local memory rearrangement, system call number diversity, and user id diversity. For CPU-bound services, the overhead of our approach will remain relatively high since all computation needs 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). Fortunately, many important services are largely I/O-bound today and trends in processor and disk performance make this increasingly likely in the future.

6. Discussion

Our prototype implementation illustrates the potential for N-variant systems to protect vulnerable servers from important classes of attacks. Many other issues remain to be explored, including how our approach can be applied to other services, what variations can be created to detect other classes of attacks, how an N-variant system can recover from a detected attack, and how compositions of design and artificially diversified variants can provide additional security properties.

Applicability. Our prototype kernel implementation demonstrated the effectiveness of our approach using Apache as a target application. Although Apache is a representative server, there are a number of things other servers might do that would cause problems for our implementation. The version of Apache used in our experiments on uses the fork system call to create separate processes to handle requests. Each child process is run as an independent N-variant system. Some servers use user-level threading libraries where there are multiple threads within a single process invisible to our kernel monitor. This causes problems in an N-variant system, since the threads in the variants may interleave differently to produce different sequences of system calls (resulting in a false detection), or worse, interleave in a way that allows an attacker to exploit a race condition to carry out a successful attack without detection. One possible solution to this problem is to modify the thread scheduler to ensure that threads in the variants are scheduled identically to preserve synchronization between the variants.

The asynchronous property of process signals makes it difficult to ensure that all variants receive a signal at the exact same point in each of their executions. Although we can ensure that a signal is sent to all the variants at the same time, we cannot ensure that all the variants are exactly at the same point within their program at that time. As a result, the timing of a particular signal could cause divergent behavior in the variants if the code behaves differently depending on the exact point when the signal is received. This might cause the variants to diverge even though they are not under attack, leading to a false positive detection. As with user-level threads, if we modify the kernel to provide more control of the scheduler we could ensure that variants receive signals at the same execution points.

Another issue that limits application of our approach is the use of system calls we classified as dangerous such as execve or unrestricted use of mmap. With our current wrappers, a process that uses these calls is terminated since we cannot handle them safely in the N-variant framework. In some cases, more precise wrappers may allow these dangerous calls to be used safely in an N-variant system. Some calls, however, are inherently dangerous since they either break isolation between the variants or allow them to escape the framework. In these situations, either some loss of security would need to be accepted, or the application would need to be modified to avoid the dangerous system calls before it could be run as an N-variant system.

Other variations. The variations we have implemented only thwart attacks that require accessing absolute memory addresses or injecting code. For example, our current instruction tagging variation does not disrupt a *return-to-libc* attack (since it does not involve injecting code), and our address space partitioning variation provides no protection against memory corruption attacks that only use relative addressing. One goal for our future work is to devise variations that enable detection of larger classes of attack within the framework we have developed. We believe there are rich opportunities for incorporating different kinds of variation in our framework, although the variants must be designed carefully to ensure the detection and normal equivalence properties are satisfied. Possibilities include variations involving memory layout to prevent classes of relative addressing attacks, file system paths to disrupt attacks that depend on file names, scheduling to thwart race condition attacks, and data structure parameters to disrupt algorithmic complexity attacks [21].

Composition. Because of the need to satisfy the normal equivalence property, we cannot simply combine multiple variations into two variants to detect the union of their attack classes. In fact, such a combination risks compromising the security properties each variation would provide by itself. By combining variations more carefully, however, we can compose variants in a way that maintains the properties of the independent variations. To do this securely, we must ensure that, for each attack class we wish to detect, there is a pair of variants in the system that differs only in the transformation used to detect that attack class. This is necessary to ensure that for each variation, there is a pair of variants that satisfy the normal equivalence property for that variation but differ in the varied property. This approach can generalize to compose n binary variations using $n + 1$ variants. More clever approaches may be able to establish the orthogonality of certain variations to allow fewer variants without sacrificing normal equivalence.

Another promising direction is to combine our approach with design diversity approaches [46, 28, 62]. We could create a 3-variant system where two variants are Apache processes running on Linux hosts with controlled address space partitioning variation, and the third variant is a Windows machine running IIS. This would provide guaranteed detection of a class of low-level memory attacks through the two controlled variants, as well as probabilistic detection of attacks that exploit high-level application semantics through the design variants.

Recovery. Our modified kernel detects an attack when the system calls made by the variants diverge. At this point, one variant is in an alarm state (e.g., crashed), and the other variant is in a possibly compromised state. After detecting the attack, the monitor needs to restart the service in an uncompromised state. Note that the attack is always detected before any system call is executed for a compromised process; this means no external state has been corrupted. For a stateless server, the monitor can just restart all of the variants. For a stateful server, recovery is more difficult. One interesting approach is to compare the states of the variants after the attack is detected to determine the valid state. Depending on the variation used, it may be possible to recover a known uncompromised state from the state of the alarm variant, as well as to deduce an attack signature from the differences between the two variants' states. Another approach involves adding an extra *recovery variant* that maintains a known uncompromised state and can be used to restart the other variants after an attack is detected. The recovery variant could be the original *P*, except it would be kept behind the normal variants. The polygrapher would delay sending input to the recovery variant until all of the regular variants process it successfully. This complicates the wrappers substantially, however, and raises difficult questions about how far behind the recovery variant should be.

7. Conclusion

Although the cryptography community has developed techniques for proving security properties of cryptographic protocols, similar levels of assurance for system security properties remains an elusive goal. System software is typically too complex to prove it has no vulnerabilities, even for small, well-defined classes of vulnerabilities such as buffer overflows. Previous techniques for thwarting exploits of vulnerabilities have used ad hoc arguments and tests to support claimed security properties. Motivated attackers, however, regularly find ways to successfully attack systems protected using these techniques [12, 55, 58, 64].

Although many defenses are available for the particular attacks we address in this paper, the N-variant systems approach offers the promise of a more formal security argument against large attack classes and correspondingly higher levels of assurance. If we can prove that the automated diversity produces variants that satisfy both the normal equivalence and detection properties against a particular attack class, we can have a high degree of confidence that attacks in that class will be detected. The soundness of the argument depends on correct behavior of the polygrapher, monitor, variant generator and any common resources.

Our framework opens up exciting new opportunities for diversification approaches, since it eliminates the need for high entropy variations. By removing the reliance on keeping secrets and providing an architectural and associated proof framework for establishing security properties, N-variant systems offer potentially substantial gains in security for high assurance services.

Availability

Our implementation is available as source code from http://www.nvariant.org. This website also provides details on the different system call wrappers.

Acknowledgments

We thank Anil Somayaji for extensive comments and suggestions; Lorenzo Cavallaro for help with the memory partitioning scripts; Jessica Greer for assistance setting up our experimental infrastructure; Caroline Cox, Karsten Nohl, Nate Paul, Jeff Shirley, Nora Sovarel, Sean Talts, and Jinlin Yang for comments on the work and writing. This work was supported in part by grants from the DARPA Self-Regenerative Systems Program (FA8750-04-2-0246) and the National Science Foundation through NSF Cybertrust (CNS-0524432).

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. *CCS* 2005.

[2] Starr Andersen. Changes to Functionality in Microsoft Windows XP Service Pack 2: Part 3: Memory Protection Technologies. *Microsoft TechNet*. August 2004.

[3] Algirdas 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.

[4] Vasanth Bala, E. Duesterwald, S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *ACM Programming Language Design and Implementation* (PLDI). 2000.

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

[6] Elena Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, D. Zovi. Intrusion Detection: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *CCS* 2003.

[7] Emery Berger and Benjamin Zorn. DieHard: Probabilistic Memory Safety for Unsafe Lan-

guages. *ACM Programming Language Design and Implementation* (PLDI), June 2006.

[8] Sandeep Bhatkar, Daniel DuVarney, and R. Sekar. Address Ofuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. *USENIX Security* 2003.

[9] Kenneth Birman. Replication and Fault Tolerance in the ISIS System. *10th ACM Symposium on Operating Systems Principles,* 1985.

[10] K. Birman, *Building Secure and Reliable Network Applications*, Manning Publications, 1996.

[11] Derek Bruening, Timothy Garnett, Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. *International Symposium on Code Generation and Optimization.* 2003.

[12] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*. Vol 0xa Issue 0x38. May 2000. http://www.phrack.org/phrack/56/p56-0x05

[13] CERT. *OpenSSL Servers Contain a Buffer Overflow During the SSL2 Handshake Process*. CERT Advisory CA-2002-23. July 2002.

[14] L. Chen and Algirdas Avizienis. N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation. *8th International Symposium on Fault-Tolerant Computing*. 1978.

[15] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. *USENIX Security* 2005.

[16] Marc Chérèque, David Powell, Philippe Reynier, Jean-Luc Richier, and Jacques Voiron. Active Replication in Delta-4. *22nd International Symposium on Fault-Tolerant Computing*. July 1992.

[17] Monica Chew and Dawn Song. *Mitigating Buffer Overflows by Operating System Randomization*. Tech Report CMU-CS-02-197. December 2002.

[18] George Coulouris, Jean Dollimore and Tim Kindberg. *Distributed Systems: Concepts and Design* (Third Edition). Addison-Wesley. 2001.

[19] Crispin Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *USENIX Security* 1998.

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

[21] Scott Crosby and Dan Wallach. Denial of Service via Algorithmic Complexity Attacks. *USENIX Security* 2003.

[22] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, D. Chanet, K. De Bosschere. Link-time Optimization of ARM Binaries. Language. *Compiler and Tool Support for Embedded Systems*. 2004.

[23] Nurit Dor, M. Rodeh, M. Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. *ACM Programming Language Design and Implementation*. June 2003.

[24] Jon Erickson**.** *Hacking: The Art of Exploitation*. No Starch Press. November 2003,

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

[26] Stephanie Forrest, Anil Somayaji, David Ackley. Building diverse computer systems. *6th Workshop on Hot Topics in Operating Systems*. 1997.

[27] The FreeBSD Documentation Project. *FreeBSD Handbook*, Chapter 24. 2005.

[28] Debin Gao, Michael Reiter, Dawn Song**.** Behavioral Distance for Intrusion Detection. *8th International Symposium on Recent Advances in Intrusion Detection*. September 2005.

[29] Daniel Geer, C. Pfleeger, B. Schneier, J. Quarterman, P. Metzger, R. Bace, P. Gutmann. *Cyberinsecurity: The Cost of Monopoly*. CCIA Technical Report, 2003.

[30] Eric Haugh and Matt Bishop. Testing C programs for buffer overflow vulnerabilities. *NDSS* 2003.

[31] David Holland, Ada Lim, and Margo Seltzer. An Architecture A Day Keeps the Hacker Away. *Workshop on Architectural Support for Security and Anti-Virus*. April 2004.

[32] D. Jewett. Integrity S2: A Fault-Tolerant Unix Platform. *17th International Symposium on Fault-Tolerant Computing Systems*. June 1991.

[33] Mark K. Joseph. *Architectural Issues in Fault-Tolerant, Secure Computing Systems*. Ph.D. Dissertation. UCLA Department of Computer Science, 1988.

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

[35] Gaurav Kc, A. Keromytis, V. Prevelakis. Countering Code-injection Attacks with Instruction Set Randomization. *CCS* 2003.

[36] John 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.

[37] Ken Knowlton. A Combination Hardware-Software Debugging System. *IEEE Transactions on Computers*. Vol 17, No 1. January 1968.

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

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

[40] Ruby Lee, D. Karig, J. McGregor, and Z. Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. *International Conference on Security in Pervasive Computing*. March 2003.

[41] John McGregor, David Karig, Zhijie Shi, and Ruby Lee. A Processor Architecture Defense against Buffer Overflow Attacks. *IEEE International Conference on Information Technology: Research and Education*. August 2003.

[42] Sjoerd Mullender and Robbert van Renesse. The International Obfuscated C Code Contest Entry. 1984. http://www1.us.ioccc.org/1984/mullender.c

[43] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. *NDSS* 2005.

[44] Adam J. O'Donnell and H. Sethu. On Achieving Software Diversity for Improved Network Security using Distributed Coloring Algorithms. *CCS* 2004.

[45] Manish Prasad and T. Chiueh. A Binary Rewriting Defense against Stack-Based Buffer Overflow Attacks. *USENIX Technical Conference*. June 2003.

[46] James 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.

[47] Michael Ringenburg and Dan Grossman. Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking. *CCS* 2005.

[48] Juan Rivas. *Overwriting the .dtors Section*. Dec 2000. http://synnergy.net/downloads/papers/dtors.txt

[49] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. *NDSS* 2004.

[50] Fred Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*. Dec 1990.

[51] Fred Schneider and L. Zhou. *Distributed Trust: Supporting Fault-Tolerance and Attack-Tolerance*, Cornell TR 2004-1924, January 2004.

[52] Kevin Scott and Jack W. Davidson. Safe Virtual Execution Using Software Dynamic Translation. *ACSAC*. December 2002.

[53] Kevin Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, M. L. Soffa. Retargetable and Reconfigurable Software Dynamic Translation. *International Symposium on Code Generation and Optimization*. March 2003.

[54] Scut / team teso. *Exploiting Format String Vulnerabilities*. March 2001.

[55] Hovav Shacham, M. Page, B. Pfaff, Eu-Jin Goh, N. Modadugu, Dan Boneh. On the effectiveness of address-space randomization. *CCS* 2004.

[56] Umesh Shankar, K. Talwar, J. Foster, D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. *USENIX Security* 2001.

[57] Stelios Sidiroglou, G. Giovanidis, A. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. *8th Information Security Conference*. September 2005.

[58] Ana Nora Sovarel, David Evans, Nathanael Paul. Where's the FEEB?: The Effectiveness of Instruction Set Randomization. *USENIX Security* 2005.

[59] Mark Stamp. Risks of Monoculture. *Communications of the ACM*. Vol 47, Number 3. March 2004.

[60] Karthik Sundaramoorthy, Z. Purser, E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *Architectural Support for Programming Languages and Operating Systems* (ASPLOS). Nov 2000.

[61] Bjorn De Sutter and Koen De Bosschere. Introduction: Software techniques for Program Compaction. *Communications of the ACM*. Vol 46, No 8. Aug 2003.

[62] Eric Totel, Frédéric Majorczyk, Ludovic Mé. COTS Diversity Intrusion Detection and Application to Web Servers. *Recent Advances in Intrusion Detection*. September 2005.

[63] Timothy Tsai and Navjot Singh. *Libsafe 2.0: Detection of Format String Vulnerability Exploits*. Avaya Labs White Paper. February 2001.

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

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

[66] John Viega, J. Bloch, T. Kohno, Gary McGraw. ITS4 : A Static Vulnerability Scanner for C and C++ Code. *ACSAC*. Dec 2000.

[67] David Wagner, J. Foster, E. Brewer, A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. *NDSS* 2000.

[68] D. Wilson. The STRATUS Computer System. *Resilient Computer Systems: Volume 1*. John Wiley and Sons, 1986. p. 208-231.

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

[70] Yongguang Zhang, H. Vin, L. Alvisi, W. Lee, S. Dao. Heterogeneous Networking: a New Survivability Paradigm. *New Security Paradigms Workshop* 2001.