# Software Self-Healing Using Collaborative Application Communities

Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis
*Network Security Lab,* Department of Computer Science, Columbia University
{*locasto, stelios, angelos*}*@cs.columbia.edu*

## Abstract

*Software monocultures are usually considered dangerous because their size and uniformity represent the potential for costly and widespread damage. The emerging concept of collaborative security provides the opportunity to re-examine the utility of software monoculture by exploiting the homogeneity and scale that typically define large software monocultures. Monoculture can be leveraged to improve an application's overall security and reliability. We introduce and explore the concept of* Application Communities*: collections of large numbers of independent instances of the same application. Members of an application community share the burden of monitoring for flaws and attacks, and notify the rest of the community when such are detected. Appropriate mitigation mechanisms are then deployed against the newly discovered fault. We explore the concept of an application community and determine its feasibility through analytical modeling and a prototype implementation focusing on software faults and vulnerabilities.*

*Specifically, we identify a set of parameters that define application communities and explore the tradeoffs between the minimal size of an application community, the marginal overhead imposed on each member, and the speed with which new faults are detected and isolated. We demonstrate the feasibility of the scheme using Selective Transactional EMulation (STEM) as both the monitoring and remediation mechanism for low-level software faults, and provide some preliminary experimental results using the Apache web server as the protected application. Our experiments show that ACs are practical and feasible for current applications: an AC of 15,000 members can collaboratively monitor Apache for new faults and immunize all members against them with only a 6% performance degradation for each member.*

## 1 Introduction

Software monocultures have been identified as a major source of problems in today's networked computing environments [30, 27, 52]. Monocultures act as force amplifiers for attackers, allowing them to exploit the same vulnerability across thousands or millions of instances of the same application across the network. Such attacks have the potential to rapidly cause widespread disruption, as evidenced by several incidents over the last few years [5, 3, 4, 2]. The severity of the problem has fueled research behind introducing diversity in software systems. However, creating a large number of different systems manually [11] not only presents certain practical challenges [31] but can result in systems that are not diverse enough [36, 16].

As a result, recent research has focused on creating artificial diversity, by introducing "controlled uncertainty" in one of the system parameters that the attacker must know (and control) in order to carry out a successful attack. Such parameters include, but are not limited to, the instruction set [33, 12, 15], the high-level implementation [48], the memory layout [13], the operating system interface [23] and others, with varying levels of success [50, 44]. However, running different systems in a network creates its own set of problems involving configuration, management, and certification of each new platform [56, 10]. In certain cases, running such multi-platform environments can decrease the overall security of the network [47].

Given the difficulties associated with artificial diversity and the pervasive nature of homogeneous software systems, can we identify a scenario in which a homogeneous software base can be used to improve security and reliability, relative to a single instance of an application? Specifically, given a large number of almost identical copies of the same application running autonomously, is it possible to employ a collaborative distributed scheme that improves the *overall* security of the group?

To answer this question, we introduce the concept of an Application Community[1] (AC), a collection of almost-identical instances of the same application running autonomously across a wide area network. Members of an AC collaborate in identifying previously unknown (zero day) flaws/attacks and exchange information so that such failures are prevented from re-occurring. Individual members may succumb to new flaws; however, over time the AC should converge to a state of immunity against that specific

[1] To our knowledge, the term first appeared in the title of the DARPA Application Communities Workshop, in October 2004. This paper expands on our short paper that introduced some of the concepts that we explore in depth here [41].

fault. The system learns new faults and adapts to them, exploiting the AC size to achieve both coverage (in detecting faults) and fairness (in distributing the monitoring task).

This definition raises several questions. First, is the approach feasible and, if so, for what types of faults? Second, how expensive can the monitoring, coordination, and reaction mechanisms be, and is it possible to share the burden equitably across the AC members? What is the performance impact of the additional computation on individual AC members? How small can an AC be to achieve coverage *and* share fairness at the same time? Finally, how can this scheme be achieved in the presence of mutually untrusted (or possibly subverted) participants?

We do not attempt to answer all of these questions in this paper, although we outline possible directions for future research. Instead, we provide a high-level analysis of the basic parameters that govern an Application Community. We then apply this analysis in a prototype AC that is targeted against remotely exploitable software vulnerabilities and input-data-driven faults. We use the Selective Transactional EMulator (STEM) technique from [51] both for fault-monitoring and immunization. Members of the AC emulate different "slices" of the application, monitoring for low-level failures (such as buffer overflows or illegal memory accesses). When a fault is detected by an member, the relevant information is broadcast to the rest of the AC. Members may verify the fault and use STEM on the identified vulnerable code slice, possibly combining this with input filtering. Our scheme also takes into consideration input from code analysis tools that identify specific code sections as potentially more vulnerable to attacks. Because of the use of STEM, it is possible to wrap the necessary functionality "around" existing applications, without requiring source code modifications.

Our analysis indicates that AC's are an achievable goal. Specifically, we analyze the effects of risk assessment and the impact of protection mechanisms on the overall workload for the AC. We determine that a reasonably-sized application (e.g., the Apache web server) requires an AC of about 17,000 members, assuming a normal (random) distribution of faults. Our experimental evaluation of Apache shows that an AC can be a practical method of protection; in the best case, an AC of size 15,000 can execute Apache with a performance degradation of only 6% at each member. A small AC of 15 hosts can execute Apache with a performance degradation of approximately 73%. This paper makes the following novel contributions:

- Introduce the concept of an Application Community as a way to exploit large-scale homogeneous software environments towards improving the security of the AC's members.
- Present the various parameters that define an Application Community and analytically explore the various tradeoffs among them.
- Illustrate the feasibility of the AC concept by implementing and experimenting with a prototype geared towards detecting and immunizing software against *previously unknown* general software failures and vulnerabilities.

**Paper Organization** We elaborate on the Application Community concept in more detail in Section 2, and explore the tradeoffs between the various parameters of such systems in Section 3. We discuss our preliminary experimental results in Section 4. Section 5 gives an overview of related work.

## 2 Application Communities

An Application Community (AC) is a collection of congruent instances of the same application running autonomously across a wide area network, whose members cooperate in identifying previously unknown flaws or vulnerabilities. By exchanging information, the AC members can prevent the failure from manifesting in the future. Although individual members may be susceptible to new failures, the AC should eventually converge on a state of immunity against a particular fault, adding a dimension of learning and adaptation to the system. The size of the AC impacts both coverage (in detecting faults) and fairness (in distributing the monitoring task). An AC is composed of three main mechanisms, for monitoring, communication, and defense, respectively.

The purpose of the monitoring mechanism is the detection of previously unknown (zero day) software failures. There exists a plethora of work in this area, namely, using the compiler to insert run-time safety checks [25], "sandboxing" [29], anomaly detection [8] and content-based filtering [1]. While shortcomings may be attributed to each of the approaches [18, 57, 53], when they are considered within the scope of an AC a different set of considerations need to be examined. Specifically, the significance of the security versus performance tradeoff is not as important as the the ability to employ the mechanism in a distributed fashion. The advantage of utilizing an AC is that the use of a fairly invasive mechanism (in terms of performance) may be acceptable, since the associated cost can be distributed to the participating members. By employing a more invasive instrumentation technique, the likelihood of detecting subversion and identifying the source of the vulnerability is increased. The monitoring mechanism in our prototype is an instruction-level emulator that can be selectively invoked for arbitrary segments of code, allowing us to mix emulated and non-emulated execution inside the same execution context [51].

Once a failure is detected by a member's monitoring component, the relevant information is distributed to the AC. Specifically, the purpose of the communication component is the dissemination of information pertaining to the discovery of new failures and the distribution of the work load within the AC. The choice of the communication model to be employed by an AC is subject to the characteristics of the collaborating community such as size and flexibility. The immediate trade-off associated with the communication model is the overhead in messages versus

the latency of the information in the AC. In the simplest case, a centralized approach is arguably the most efficient communication mechanism, however, there are a number of issues associated with this approach. If there is a fixed number of collaborating nodes, a secure structured overlay network [34, 21] can be employed with exemption from the problems associated with voluminous joins and leaves. If nodes enter and leave the AC at will, a decentralized approach may be more appropriate. Efficient dissemination of messages is outside the scope of this paper, but has been the topic of research, *e.g.*, [6].

The final component of our architecture is responsible for immunizing the AC against a specific failure. Ideally, upon receiving notification of an experienced failure, individual members independently confirm the validity of the reported weakness and create their own fix in a decentralized manner thus reducing issues regarding trust. At that point, each member in the AC decides autonomously which fix to apply in order to inoculate itself. As independent verification of an attack report may be impossible in some situations, a member's action may depend on predefined trust metrics. Depending on the level of trust among users, alternative mechanisms may be employed for the adoption of universal fixes and verification of attack reports. In the case of systems where there is minimal trust among members a voting system can be employed at the cost of an increased communication overhead. Finally, given that a fix could be universally adopted by the AC, special care must be placed in minimizing the performance implications of the immunization.

The inoculating approach that can be employed by the AC is contingent on the nature of the detection mechanism and the subsequent information provided on the specific failure. The type of protection can range from statistical blocking, behavioral or structural transformation. For example, IP address and content filtering [1], code randomization [33] and emulation [51] may be used for the protection of the AC. For the defense component in our experimental prototype, we use the STEM instruction-level emulator.

### 2.1 Selective Transactional EMulation

STEM is an *x86* instruction-level emulator that can be selectively invoked in the spatial or temporal domain during a program's execution. In other words, we can decide which routines to emulate during program execution (with the rest of the code running natively on the system CPU), and whether to emulate or run natively any specific routine each time it is invoked. Although STEM allows us to operate at the granularity of individual instructions, we confine ourselves to emulating (or not) whole routines because these represent convenient abstractions that aid in program recovery, as we shall see soon.

When a piece of code is being emulated, STEM checks the validity of every instruction's operands prior to emulating that instruction. For example, STEM can verify that the destination address of a memory-write operation is in a properly OS-mapped page, or that a memory-write to control information on the stack (*e.g.*, the return address) is using input-supplied data (which may indicate a buffer overrun), similar to TaintCheck [43] and Minos [26]. In the latter case, interesting future work includes identifying non-control hijacking overflow attacks [22], which is a non-trivial problem. In addition, STEM maintains a log of all memory changes done during emulation of the code.

Once a fault is detected, and it has not been seen before, the emulator enters an "error virtualization" phase. The goal is to determine how to modify program execution so that this fault (which the program is not designed to handle) is translated into an error that the existing program code can recover from. The intuition is that by simulating an error-return of the function within which the fault occurred, its caller may be able to handle the error appropriately (*i.e.*, through existing error-checking code that the programmer wrote).

STEM simulates an error-return by first taking a snapshot of the program state (memory and registers) at the time of the fault, putting an appropriate error code in the return value field on the stack, undoing all memory changes made during emulation using the log, and returning execution to the caller. Currently, STEM uses certain heuristics to determine the correct value to use as an error code, which require knowledge of the return type of each function. STEM uses both $-1$ and $0$ as return codes (testing these separately) for functions returning an integer or a long, and uses `NULL` as an error code for functions returning a pointer of any type. Of course, these are not guaranteed to be correct conventions, although their impact can be verified through testing. Static program analysis or programmer-supplied annotations (either in the code or out-of-band) may be used to achieve better accuracy, but this remains a topic for future work. In practice, these heuristics work in over $80\%$ of all cases examined (see Section 4).

If forcing an error-return of the top-most function on the stack does not work (*i.e.*, the program exhibits another fault shortly after simulating this error-return), STEM simulates error-return from the second top-most function on the stack. This process is repeated until the application does not terminate abnormally after STEM forces an error-return of a function call sequence. In the extreme case, the whole application could end up being emulated, at a significant performance cost. We give some indications on the effectiveness and performance impact of this approach in Section 4. For more details, see [51].

If the program does not crash after the forced return, STEM has found a "vaccine" for the fault, which it can use as a remediation technique if a fault is detected in the future. If the fault is not triggered during an emulated execution, emulation halts at the end of the code segment, and program execution reverts to the native CPU.

The overhead of emulation is incurred at all times (whether the fault is triggered or not). To minimize this cost, STEM must identify the smallest piece of code that it needs to emulate in order to catch and recover from the

fault. Using an application community to divide the emulation task across large numbers of application instances can minimize the performance cost on any specific instance. Once a fault is identified by one application, the relevant information (faulty function, recovery strategy) is propagated across the community. All AC members then begin emulating the problematic code and thus become immune to the specific failure.

# 3 Analysis

Here, we present an analysis of the properties that govern the AC. Subsection 3.1 explains the calculations that affect the size of the AC based on the parameters we list in Table 1. We consider the problem of distributing work to the AC members in Subsection 3.2 and present some simple approaches to addressing it. Subsection 3.2 also defines the general form of the work distribution problem, which we term the AC-CALLGRAPH-KNAPSACK problem. In addition, we outline a strategy for solving this problem that optionally takes into consideration member-local policy. Subsection 3.3 briefly discusses the probability of catching new faults by duplicating monitoring responsibilities. Subsection 3.4 presents the results of our analysis and simulations.

To make our analysis concrete, we consider an AC aimed at low-level software attacks and faults (*e.g.*, buffer overflows, illegal memory dereferences, exceptions arising from illegal instruction operands, and other faults that cause process termination). ACs protecting against different types of failures are possible; we do not consider them further in this paper, except to the extent that our analysis apply to such systems.

**Work Overview** We formalize the notion of total work in the AC, $W$, as a function of both the cost of the monitoring mechanisms and the perceived vulnerability of each function. The *actual* work done can be calculated by two runtime metrics: $(a)$ the number of machine instructions executed by the function during a request, and $(b)$ the amount of real time that a function takes to execute a request. Each metric has advantages and drawbacks. For example, while instruction count is an intuitive unit and is straightforward to measure, there is a clear difference in computation between 100 logical "AND" operations and 100 floating point "MUL" operations (everything else, like data dependencies and structural hazards, being equal). On the other hand, using only timing information can obscure the effects of nondeterminism or interaction with other systems even though it may provide a more realistic sense of system response or throughput.

Our main focus is on calculating the amount of work in the system and determining the level of resources needed to achieve both a *fair coverage* and a *full coverage*. That is, we wish to determine an assignment of monitoring tasks that dictates an equal amount of work for each member of the AC while simultaneously guaranteeing that all functions in an application are being monitored. If the size of the AC is already fixed, then $W$ dictates how much work each member should do. If it is not yet fixed, then $W$ serves as a lower bound on the size of the "optimally fair" AC. If the value of "fairness" is predetermined, falling below the minimum set of AC members means that we must either reduce coverage to maintain fairness or reduce fairness to maintain coverage. If fairness means that each node does an equal amount of work, the system can degrade gracefully.

## 3.1 Work Calculation

The cost, $c_i$, of executing each $f_i$ is a function of the amount of computation present in $f_i$ (we denote this computation as $x_i$) and the amount of risk present in $f_i$ (we denote this risk as $v_i$). All the information (an annotated call graph of a profiling run) needed to perform the analysis is present at each member of the AC. The calculations can be kept in a form similar to Table 2.

The calculation of $x_i$ can be driven by at least two different metrics: $o_i$, the raw number of machine instructions executed as part of $f_i$, or $t_i$, the amount of time spent executing $f_i$. Since the cost of certain functions (as noted above) may not be easy to extrapolate from total instructions executed, the experimental evaluation in Section 4 uses the running time of a function as a measure of $x_i$, but this analysis will assume either metric may be used. Both $o_i$ and $t_i$ can vary as a function of time or application workload according to the application's internal logic[2]. For example, an application may perform logging or cleanup duties after it passes a threshold number of requests. Code that normally lies dormant would then be executed. Future work will explore functions that approximate $x_i$'s value at a given time for either metric ($o_i$ or $t_i$), as either parameter may change during the lifetime of an AC (*e.g.*, due to hardware or software upgrades).

The risk factor is somewhat harder to characterize, as it is more likely to vary during runtime and it is not clear how to classify risk in terms of execution time or number of machine operations. We approximate the risk by a simple scaling factor $\alpha$ based on a statistical measure of vulnerability introduced by the CoSAK project[3]. Other measures (*e.g.*, static analysis tools) may be used; exploring the range of risk metrics is interesting future work.

Let $v_i$ represent a vulnerability (or risk) score for $f_i$. This $v_i$ may be the result of a complex function that calculates risk or may be a simple scalar factor $\alpha$. Its purpose is to weight a function such that more members monitor it. Let $T = \sum_{i=1}^{n} x_i$. If we express the relative cost of executing each $f_i$ as some cost function $c_i = C(f_i, x_i, v_i)$, then the total amount of work in the system can be represented by the equation: $W = \sum_{i=1}^{n} C(f_i, x_i, v_i)$.

[2] In order to gain confidence in the value of $x_i$, we determine $x_i$ over a range of requests to see if the application somehow varies the amount of instructions it executes based on the number of requests it has handled so far.

[3] `http://serg.cs.drexel.edu/projects/cosak/`

| Variable | Description | Variable | Description |
|---|---|---|---|
| $N$ | total AC members needed | $F$ | set of application functions |
| $n$ | the size of $F$ | $E$ | set of edges for $F$ |
| $G$ | directed call graph of $(F, E)$ | $W$ | the total amount of work |
| $Z$ | the base unit of work | $C$ | a cost function |
| $M$ | the set of AC members | $m_i$ | the $i^{th}$ member of $M$ |
| $f_i$ | the $i^{th}$ member of $F$ | $c_i$ | the total cost of executing $f_i$ |
| $x_i$ | the performance cost of $f_i$ | $v_i$ | the risk cost of executing $f_i$ |

**Table 1. Various parameters and data sets for an Application Community. The risk score and performance score for each function combine to define the amount of work in the system. To be fair to each member, an equivalent amount of resources must be allocated to the monitoring of some subset of functions.**

We provide a cost function in two phases. The first phase calculates the cost due to the amount of computation for each $f_i$. The second phase normalizes this cost and applies the risk factor $v_i$ to determine the final cost of each $f_i$ and the total amount of work in the system. If we let $C(f_i, x_i) = \frac{x_i}{T} * 100$, then we can normalize each cost by grouping a subset of $F$ to represent one unit of work. Membership in this subset can be arbitrary, but is meant to provide a flexible means of defining what a work unit translates to in terms of computational effort. A good heuristic is to group the $k$ lowest cost functions together and declare the sum of their work as the base work unit, $Z$. Every other function's cost is normalized to this work unit, and $r_i$ represents the relative weight of each $f_i$ with respect to $Z$. As a result, we know that $W = N_{base} = \sum_{i=1}^{n} r_i$ represents the total number of AC members needed to obtain full coverage of an application when we only consider performance.

However, we still have to account for the measure of a function's vulnerability (or alternatively, the risk level of executing the function). We can treat the vulnerability score of a function as a discrete variable with a value of $\alpha$ (where $\alpha$ can take on a range of values according to the amount of risk). Thus,

$$v_i = \begin{cases} \alpha & \text{if } f_i \text{ is vulnerable, } \alpha > 1; \\ 1 & \text{if } f_i \text{ is not vulnerable.} \end{cases} \quad (1)$$

Given the scaling factor $v_i$ for each function, we can determine the total amount of work in the system and the total number of members needed to monitor every function is $W = N_{vuln} = \sum_{i=1}^{n} v_i * r_i$

| $f_i$ | $x_i$ | $r_i$ | $v_i$ | $T$ | $C(f_i, x_i)$ | $r_i * v_i$ |
|---|---|---|---|---|---|---|
| a() | 100 | 1 | $\alpha_1$ | 600 | 16 | $\alpha_1$ |
| b() | 200 | 2 | $\alpha_2$ | 600 | 33 | $2\alpha_2$ |
| c() | 300 | 3 | $\alpha_3$ | 600 | 50 | $3\alpha_3$ |

**Table 2. An example of AC work calculation. Each member of the AC can calculate this table independently. Here, the AC is executing an application with three functions. The choice of $\alpha$ is somewhat arbitrary and can vary based on the context of a particular function.**

### 3.2 Work Distribution

After each AC member has a clear idea of the amount of work in the system, work units (slices) must be distributed to each member. In the simplest scenario, a central controller simply assigns approximately $\frac{W}{N}$ work units to each node. A more robust method of work distribution would be for each AC member to autonomously determine their work set. Each member can simply iterate through the list of work units, flipping a coin weighted with the value $v_i * r_i$. If the result of the flip is "true" then the member adds that work unit to its work set. A member stops when its total work reaches $\frac{W}{N}$. Such an approach offers statistical coverage of the application. A more elegant method of work distribution is possible; since a full treatment of it is beyond the scope of this paper, we only provide an overview of the approach.

**Distributed Bidding** The problem of assigning work to individual members in the AC can be seen as an instance of the general KNAPSACK problem. We call this problem the AC-CALLGRAPH-KNAPSACK problem. For the call graph $G$, each node has a particular weight ($v_i * r_i$ from above). The problem is then to assign some subset of the weighted nodes in $F$ to each member of $M$ such that each member does no more than $\frac{W}{N}$ work. We can relax the threshold constraint to be approximately $\frac{W}{N}$ within some tunable range $\epsilon$. Thus, $\epsilon$ is a measure of the fairness of the system. Once the globally fair amount of work $\frac{W}{N}$ is calculated, each AC member should be able to adjust their workload $\epsilon$ by bargaining with other AC members via a distributed bidding process.

Two additional considerations impact the assignment of work units to AC members. First, we would like to preferentially allocate work units with higher weights, as these work units likely have a heavier weight due to an increased vulnerability score. Even if the weight is derived solely from the measure of performance cost, assigning more members to it is beneficial because these members can round-robin the monitoring task so that any one member does not have to assume the cost alone. Second, in some situations, the value $v_i * r_i$ will be greater than the average amount of work $\frac{W}{N}$. Achieving fairness then means that the value $v_i * r_i$ defines the quantity of AC members that must be assigned to it, and the sum of all these quan-

tities defines the minimum number of members that must participate in an AC to achieve a fair and full coverage for a particular application.

Our algorithm works in two rounds. First, each member calculates a table similar to Table 2. Then, AC members enter into a distributed bidding phase to adjust their individual workload. The distributed algorithm uses tokens to bid; tokens map directly to the number of time quanta that an AC member is responsible for emulating the execution of a particular code slice. A node will accumulate tokens by taking on extra computation. The distributed algorithm makes sure that each node should not accumulate more than the total number of tokens allowed by the choice of $\epsilon$. Since we currently assume a collaborative AC, useful future work can analyze what can be done to protect the bidding process in the face of various threats (*e.g.*, insider accumulating tokens) and constraints (*e.g.*, anonymity for AC members).

### 3.3 Overlapping Coverage

While "full coverage" means that every work unit (or slice) of an application is being monitored for the given time unit, it does not mean that every AC member's individual application is being fully monitored. Consider the following situation: member $A$ is monitoring function $Z$, and member $B$ is monitoring function $Y$. If a fault is present in function $Z$, $B$ will miss it. Even though the *community* may catch the fault (by virtue of $A$'s willingness to monitor $Z$), there may exist individual servers that have not yet detected the fault (*e.g.*, $B$, or even $A$ if $A$ is executing another part of the application and not $Z$). There is a tradeoff between the amount of individual coverage and how quickly the AC can identify a new fault.

If AC members monitor more than their share (*e.g.*, $A$ also monitors $Y$ and $B$ also monitors $Z$, then we have increased coverage to 200% and made sure that the fault, if present, is detected as quickly as possible. A similar situation is presented in Table 3. Assuming a uniform random distribution of new faults across AC members, the probability of a fault happening at a particular member $k$ is: $P(\text{fault}) = \frac{1}{N}$. The probability of member $k$ detecting the error is a function of $k$'s individual coverage level. For Alice in Table 3, $P(\text{detection}) = \frac{1}{4}$. Thus, the probability of Alice detecting a new fault is the probability that the fault happens at Alice *and* that Alice detects the fault: $P(\text{fault at Alice} \wedge \text{detection}) = \frac{1}{N} * \frac{1}{4}$. Given that $N = 4$ for Alice's AC, the probability that Alice will detect a new fault is $\frac{1}{16}$. Similar calculations for each member shown in Table 3 show that the application has an overall new fault detection probability of $\frac{3}{8}$. If every AC member adds the missing functions to its auxiliary set, then each member has a $\frac{1}{4}$ chance of detecting the new fault: this probability is exactly $\frac{1}{N}$, their best possible chance (because the fault could happen to one of the other three). At the cost of 400% coverage, the AC has reached a probability of 1 for new fault detection. We can generalize this relationship:

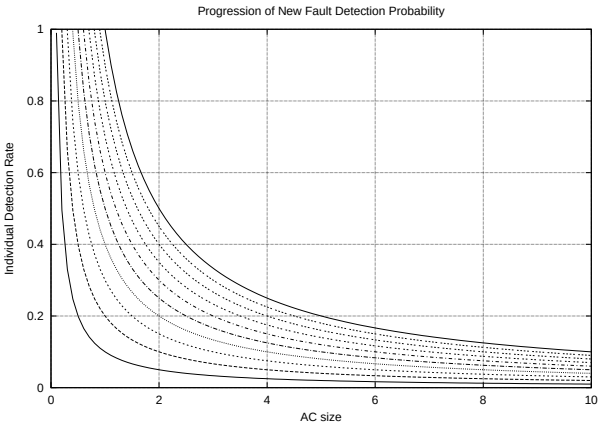


**Figure 1. Rate of detection varies with AC size. Each line represents a member-local coverage level in 10% increments, with 10% being the bottom curve. Note how detection degrades as the AC size grows: each member is only doing a constant amount of extra coverage. However, when every member performs 100% local coverage, they regain the best chance to detect the fault, and achieve a *community probability* of 1 that the fault is detected when it first occurs; *e.g.,* an AC of size 2 with each member doing 100% coverage gives each member a probability of $\frac{1}{2}$ in detecting the fault, *i.e.,* the probability that the fault is seen by that member.**

the probability of the AC detecting the fault is

$$P(\text{AC detect}) = \sum_{i=1}^{N} \frac{1}{N} * k_i \qquad (2)$$

where $k_i$ is the percentage of coverage at AC member $k$. Figure 1 shows how the AC's detection rate improves as individual member coverage tends towards 100%. As each $k_i$ goes to 100%, Equation 2 becomes $\sum_{i=1}^{N} \frac{1}{N}$, or $\frac{N}{N}$, a probability of 1 that the fault is detected when it first occurs. The worst case in terms of performance is the best case in terms of rapid detection and requires $N * 100\%$ coverage.

### 3.4 Analytical Results

Our simulations explore the influence of various parameters on the amount of work in the AC: $(a)$ the size of the application (number of functions it contains), $(b)$ the distribution of work between functions, $(c)$ the level of work present in each function, and $(d)$ the policy for determining the $\alpha$ score (and thus $v_i$) for each function.

We simulate an application with a small, medium, large, and massive (20, 200, 2000, and 20000 functions, respectively) size. Similarly, the level of work for each function is small, medium, large, and massive (50, 500, 5000, 50000, respectively) normalized work units. The work level is interpreted differently for each distribution scheme. We examine three types of distributions of $r_i$. The *even* distribution defines an equal work level for every function. The

| AC Member ID | Monitored Set | Auxiliary Set |
|---|---|---|
| Alice | {A, F} | {∅} |
| Bob | {B, C} | {A, F} |
| Carol | {D, E} | {G, H} |
| David | {G, H} | {∅} |

**Table 3. A distribution of work and overlapping monitoring. Here, Alice and David choose not to do extra monitoring. However, Bob and Carol are each monitoring two more functions than strictly necessary for "fairness" and 100% *application community* coverage. Bob and Carol have increased their individual coverage from 25% to 50%, and their overall chances of detecting a new fault from $\frac{1}{16}$ to $\frac{1}{8}$.**

*norm* distribution is an approximately "normal" distribution that is centered on an average value of the work level. The *skew* distribution sets the cost of most functions relatively low, but includes a few functions that account for a large part of the execution cost.

We determine $\alpha$ according to two policies: *exp* and *flat*. The *flat* policy applies a static factor of 10 for every function deemed vulnerable. The *exp* policy exponentially increases the value of $\alpha$ for "more vulnerable" functions. Every function is assigned a default $\alpha$ value of 1. For both policies, we determine if a function is vulnerable or not by examining the distance of the function (in the application call graph) from a `read()` system call, using the heuristics proposed by the COSAK project. For our simulations, we assume that the path length from each function $f_i$ to a `read()` system call is normally distributed around a mean of *log(n)*, where $n$ is the size of the call graph, leaving exploration of different distributions as future work. Thus, our simulation assigns a normally distributed distance about this mean to each function, representing the distance from a `read()` system call. If a program is heavily saturated with `read()`'s, our simulation underestimates the weight that should be assigned to each function. However, this is not a problem, as this situation can be easily detected from the application's call graph, and every function can be scaled accordingly. The behavior of the *flat* policy is seen in Figure 2. Figure 3 shows the relationship between a program's size and the workload $W$ of the AC. While the values for workload are quite large, they are based on a program where each function performs about 50000 work units. Our simulations for smaller workloads show the same relationship with lower total cost. We also consider a more realistic case (see Figure 2) for an Apache-like application: of medium size (200 functions), with a normal distribution of $x_i$ (cost) and a *flat* policy for determining $\alpha$.

## 4 Evaluation

In this section, we quantitatively measure the tradeoffs presented in Section 3, namely, the size of the an application community and the length of the work time quantum. Measurements are conducted using the Apache web server as the protected application and STEM as the monitoring and remediation component.

**Effectiveness of STEM** For our monitoring and remediation mechanism we use an instruction-level emulator, *STEM*, that can be selectively invoked for arbitrary segments of code, allowing the mix of emulated and non-emulated execution inside the same execution run. The emulator allows us to $(a)$ monitor for the specific type of failure prior to executing the instruction, $(b)$ undo any memory changes made by the code function inside which the fault occurred, by having the emulator record all memory modifications made during its execution, and $(c)$ simulate an error-return from said function. One of the key assumptions behind STEM is that we can create a mapping between the set of errors and exceptions that *could* occur during a program's execution and the limited set of errors that are explicitly handled by the program's code. We call this approach "error virtualization".

In a series of experiments using a number of open-source server applications including Apache, OpenSSH, and Bind, we showed that our "error virtualization" mapping assumption holds for more than 88% of the cases we examined. Testing with real attacks against Apache, OpenSSH, and Bind, we showed that this technique can be effective in quickly and automatically protecting against zero day attacks. Although full emulation is prohibitively expensive (30-fold slowdown), selective emulation imposes an overhead between 1.3 and 2, depending on the size of the emulated code segment, assuming the fault is localized within a small code region.

**Performance** In order to understand the performance implications of an AC, we run a set of performance benchmarks which we use to explore the tradeoffs presented by our system. We employ STEM on the Apache web server and measure the overhead of our protection mechanism in terms of coverage and fairness.

Before we explore the costs associated with using an AC, we examine the cost of protecting a single instance of Apache. We demonstrate that emulating the bulk of an application entails a significant performance impact. In particular, we emulated the main request processing loop for Apache (contained in *ap_process_http_connection()*) and compared our results against a non-emulated Apache instance. In this experiment, the emulator executed roughly 213,000 instructions. The impact on performance is clearly seen in Figure 4, which plots the performance of the fully emulated request-handling procedure.

To get a more complete sense of this performance im-

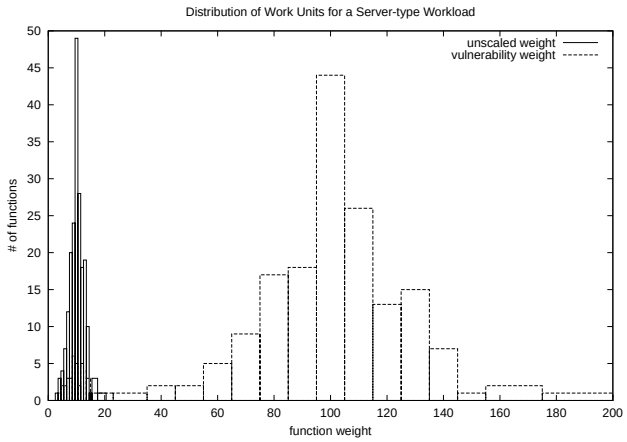


**Figure 2. Workload scaling for a realistic parameter set. For an application of about 200 functions in size, with each function's work normally distributed around a normalized $r_i$ of 10 and a *flat* policy for $\alpha$, the workload ($W$) scales from 2020 to 16897.**

| Slice size | Requests/sec | Number of servers |
|---|---|---|
| 10.34 | 148 (27%) | 15 |
| 5.24 | 333 (62%) | 30 |
| 0.25 | 380 (70%) | 635 |
| 0.14 | 497 (92%) | 1135 |
| 0.04 | 471 (87%) | 3973 |
| 0.01 | 506 (94%) | 15893 |

**Table 4. Work-time quantums and their effects on Apache performance and AC size.**

pact, we timed the execution of the request handling procedure for both the non-emulated and fully-emulated versions of Apache by embedding calls to *gettimeofday()* where the emulation functions were (or would be) invoked.

For our test machines and sample loads, Apache normally (*e.g.*, non-emulated) spent 6.3 milliseconds to perform the work in the *ap_process_http_connection()* function, as shown in Table 5. The fully instrumented loop running in the emulator spends an average of 278 milliseconds per request in that particular code section.

To calculate the amount of work in the system and determine the level of resources needed to achieve *fair coverage* and *full coverage* as explained in Section 3, we first need to get a detailed analysis of the run-time characteristics of the protected application. For this purpose, we ran a profiled version of Apache against a set of test suites and examined the subsequent call-graph generated by these tests with *gprof* and Valgrind [49]. The ensuing call trees were analyzed in order to extract the time spent doing work for each function. Using the corresponding costs, we evaluate the performance of Apache in requests per second, by employing STEM as the protection mechanism on different work time quantums to achieve full coverage.

We start with the examination of the performance of an unmodified Apache server using ApacheBench. We then

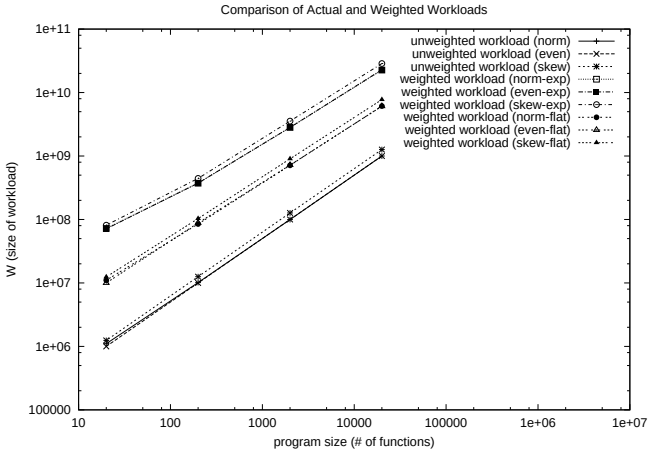


**Figure 3. A logscale comparison of workloads given a vulnerability policy. Note that the raw values are quite high, but are drawn from data that assumes a massive value (50000) for normalized workload. More important is how the relationship between the size of the program and the total workload is affected by the choice of vulnerability policy.**

| Apache | trials | Mean | Std. Dev. |
|---|---|---|---|
| Normal | 18 | 6314 | 847 |
| STEM | 18 | 277927 | 74488 |

**Table 5. Timing of main request processing loop. Times are in microseconds. This table shows the overhead of running the whole primary request handling mechanism inside the emulator. In each trial a user thread issued an HTTP GET request.**

proceed with the emulation of different functions representing varying work-time quantums and measure the performance overhead in terms of requests per second. Specifically, all functions invoked at least once per transaction are examined for their relative cost (time spent in function). Given the per function cost, we sample 6 functions that represent a characteristic distribution of work done per request. At that point, we wrap each function with STEM and measure the performance overhead imposed by the emulation.

The machine we chose to host Apache was a single Pentium IV at 3GHz with 1GB of memory running RedHat Linux with kernel 2.4.24. The client machine was a Pentium IV at 2 GHz with 1GB of memory running Debian Linux with kernel 2.6.8-1. For the performance evaluation of Apache, we use ApacheBench, a complete benchmarking and regression testing suite. Examination of application response is preferable to explicit measurements in the case of complex systems, as we seek to understand the effect on overall system performance. Specifically, we look at the requests per second served by Apache for 10000 requests at a concurrency of 5. We use the average of 100 runs omitting statistical outliers.

As illustrated in Figure 5 and Table 4, we examine the use of a variety of work-time quantums on raw Apache

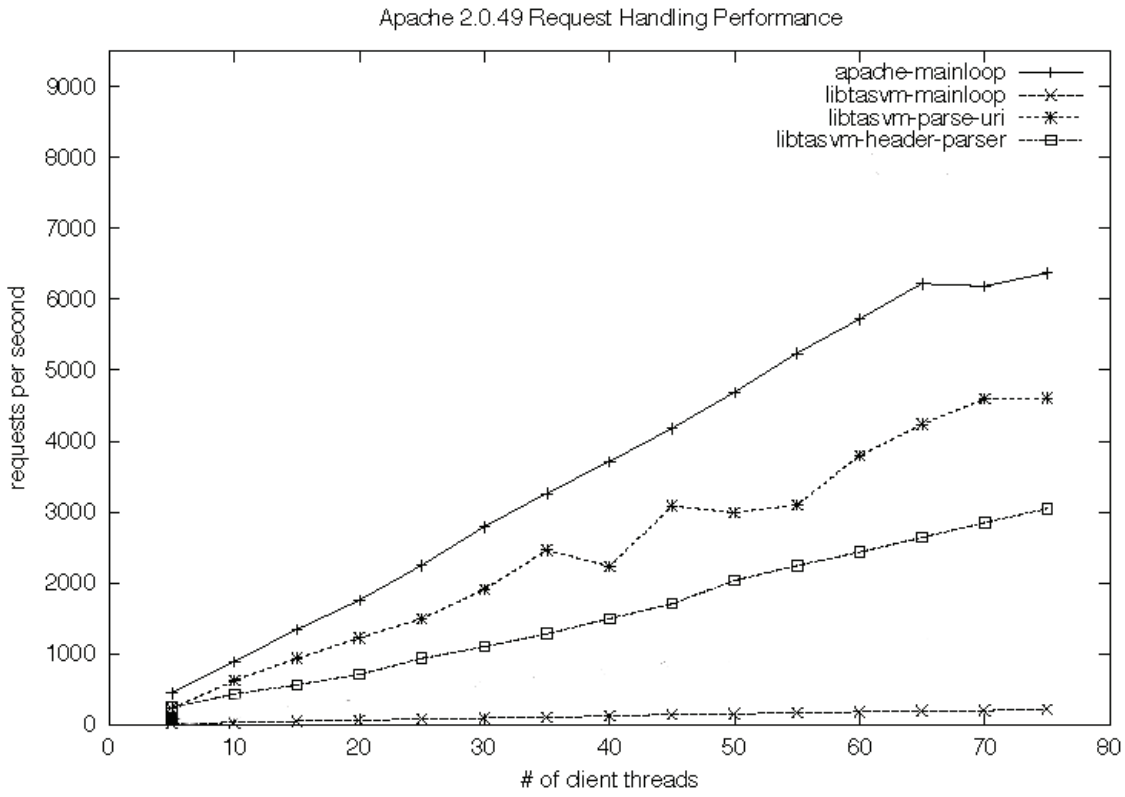


**Figure 4. Performance of the system under various levels of emulation. While full emulation is fairly expensive, selective emulation of input handling routines appears quite sustainable.**

performance and coverage. As expected, emulating large "slices" using STEM translates into lower performance for each participating member but requires the smallest community size for 100% coverage. Concretely, using the largest work-time quantum translates into a performance degradation of 73% per member and an AC size of 15 members. As the "slice" size is reduced (using a less expensive function as the base), the performance overhead per member is decreased at the cost of a larger community. For the smallest work-time quantum, a performance overhead of 6% is experienced per member whilst the size of the AC grows to 15893. These results are very encouraging and closely follow the intuition provided in Section 3

Figure 6 illustrates the effect of varying the vulnerability index on the size of the community for 100% coverage. In this example, we double the number of servers required to cover an $\alpha$ region. We start with the case where 25% of the code is considered potentially vulnerable and increment the $\alpha$ value until the entire code base is covered. As expected, when a higher percentage of the code base is deemed vulnerable, the community needs to be larger to realize fair coverage. Note that the effect on Apache performance is linear despite an aggressive protection policy. Our experiments demonstrate that the use of an AC can alleviate the problems associated with using an invasive protection mechanism by fairly distributing work to participating members. Furthermore, we show that the flexibility of our protection mechanism can facilitate the adjustment of parameters associated with the requirements of an AC.

## 5 Related Work

The synthesis of our system draws on work from many other areas. Most notably, the major themes of our system are distributed large-scale collaborative security and survivable computing. Traditional fault-tolerance techniques are a related area of work, although they are primarily intended to supply enough resources for a particular enclave to survive an attack by outlasting the resources of an attacker.

Secure survivable architectures are typically very application- or domain-specific. Ghosh, *et al.* [28] propose "fault injection analysis" applied to software, while Strunk, *et al.* [55] apply a low-level approach: they propose an intrusion detection and recovery model at the storage layer. Kreidl, *et al.* [37] propose a formalized feedback-driven model for individual COTS applications. SABER [35] is a generalized, application-neutral architecture that encompasses a broad array of tools. The APOD project [9] uses a combination of intrusion detection, firewalls, TCP stack probes, virtual private networks, bandwidth reservation, and traffic shaping mechanisms, to allow applications to detect attacks and contain the damage of successful intrusions by changing their behavior. They also discuss the use of randomizing techniques, such as changing the TCP ports applications listen to.

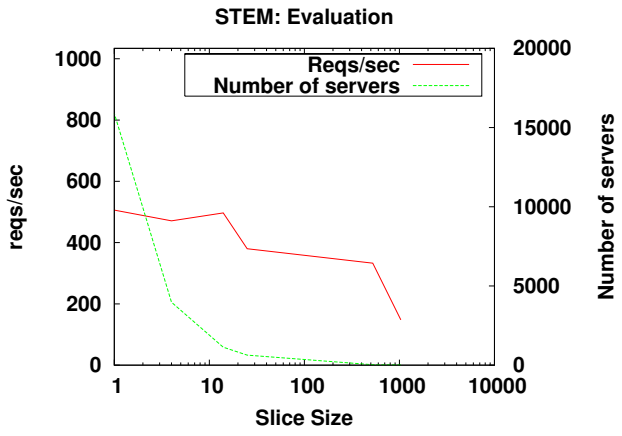


**Figure 5. The effect of different work-time quantums on request/sec for Apache and on the size of the AC.**

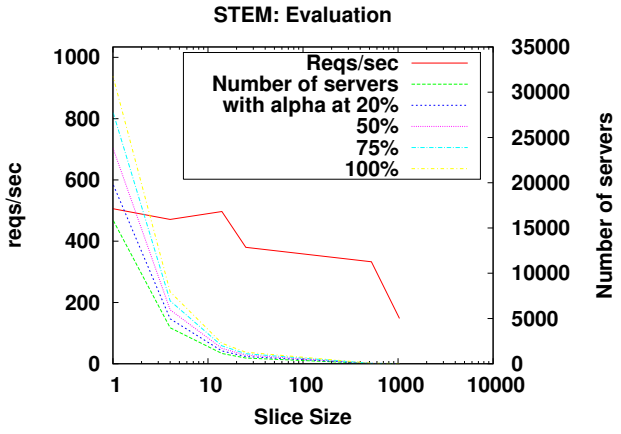


**Figure 6. The impact of the vulnerability index on the size of an AC.**

[54, 40] explore the notion of collaborative security with specific application to coordinating IDS alerts for worms and scanning attacks across administrative domains. Indra [32] is another scheme that provides a peer-to-peer approach to intrusion detection. In addition, a collaborative approach to containing the spread of worms has been the focus of current research [42, 46, 7]. Vigilante [24] proposes the concept of Self-Certifying Alerts, which are exchanged between hosts as a result of a newly detected attack. The recipient can verify the validity of the alert and use an appropriate protection mechanism. In Vigilante, every host checks for all attacks all the time, in contrast to our more general load-sharing-capable approach. Furthermore, we propose a software-based protection mechanism (as opposed to their use of filtering) that both protects against attacks and also maintains application availability, thus providing an element of real "software healing."

O'Donnell and Sethu [44] study algorithms for the assignment of distinct software packages (whether randomized or inherently different) to individual systems in a network, towards increasing the intrinsic value of available diversity. Their goal is to limit the ability of a malicious node to compromise a large number (or any) of its neighbors with a single attack. Unfortunately, their abstraction does not translate well to the end-to-end semantics of the Internet, where any host can contact another without (in most cases) needing to pass through a series of other hosts. Their work can be viewed as a situation where a community of nodes collaboratively diversifies, where our work seeks to collaboratively protect a homogeneous group of nodes.

DOMINO [58] is an overlay system for cooperative intrusion detection. The system is organized in two layers, with a small core of trusted nodes and a larger collection of nodes connected to the core. The experimental analysis demonstrates that a coordinated approach has the potential of providing early warning for large-scale attacks while reducing potential false alarms. A similar approach using a DHT-based overlay network to automatically correlate all relevant information is described in [19]. [59] describes an architecture and models for an early warning system, where the participating nodes/routers propagate alarm reports towards a centralized site for analysis. The question of how to respond to alerts is not addressed, and, similar to DOMINO, the use of a centralized collection and analysis facility is weak against worms attacking the early warning infrastructure.

Gamma [45, 14] is an architecture for instrumenting software such that information that can lead to future improvements of the code can be gathered in a central location, without imposing excessive overhead to any given code instance. Their technique, software tomography, is similar to our code-slicing approach, and has been combined with a dynamic software update mechanism that allows code producers to fix bugs as they are detected. Our work is different primarily in that $(a)$ we introduce *a fully automated* mechanism for software healing, $(b)$ which does not require merging of the monitoring information from the different software instances.

The Cooperative Bug Isolation project [38, 39] uses a sampling infrastructure to gather information from a program's execution and communicates its findings to a central database where the data is analyzed to extract debugging information automatically. In order to reduce the instrumentation cost they statistically spread the monitoring across an application and a large user base.

Finally, a number of efforts have been made to protect applications via the introduction of diversity [33, 12, 15, 48, 13, 23, 50, 44]. The ability to rollback [17] and cleanly restart [20] is critical to our system, and we expect to integrate such capabilities in our future work.

## 6 Conclusions

The growing concern about monocultures at all levels of computing systems has engendered a body of research that seeks to increase system diversity. Given the large legacy base and the continuing need for computing systems

to communicate and interoperate, introducing artificial diversity is no easy task, and it is often hampered by extra management complexity. While we support the notion of artificial diversity and actively explore its use, this paper introduces the concept of *Application Communities*: our contribution is a method for exploiting the resources available in large scale monocultures to provide protection to each member of the community. We postulate that systems that may not tolerate the introduction of artificial diversity or cannot easily take advantage of it may benefit from the use of an AC.

Our experimental and analytical results show that members of an application community can reasonably deploy our novel monitoring framework (*STEM*) and collaborate to share the overhead of its protection mechanisms. We validate our analysis of workload and fault discovery by experimenting with the Apache web server. Furthermore, AC members can employ *STEM* to automatically recover from attacks and preemptively notifiy other AC members of new faults, thus inoculating the community at the cost of a few failed members.

## References


[1] Using Network-Based Application Recognition and Access Control Lists for Blocking the "Code Red" Worm at Network Ingress Points. Technical report, Cisco Systems, Inc.

[2] CERT Advisory CA-2001-19: 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. `http://www.cert.org/advisories/CA-2001-19.html`, July 2001.

[3] CERT Advisory CA-2001-26: Nimda Worm. `http://www.cert.org/advisories/CA-2001-26.html`, September 2001.

[4] Cert Advisory CA-2003-04: MS-SQL Server Worm. `http://www.cert.org/advisories/CA-2003-04.html`, January 2003.

[5] CERT Advisory CA-2003-21: W32/Blaster Worm. `http://www.cert.org/advisories/CA-2003-20.html`, August 2003.

[6] D. Agrawal and A. Malpani. Efficient dissemination of information in computer networks. *Comput. J.*, 34(6):534–541, 1991.

[7] K. Anagnostakis, M. B. Greenwald, S. Ioannidis, A. D. Keromytis, and D. Li. A Cooperative Immunization System for an Untrusting Internet. In *Proceedings of the 11th IEEE International Conference on Networks (ICON)*, pages 403–408, October 2003.

[8] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. J. Stolfo. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. In *Proceedings of the* $5^{th}$ *International Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.

[9] M. Atighetchi, P. Pal, F. Webber, and C. Jones. Adaptive Use of Network-Centric Mechanisms in Cyber-Defense. In *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications*, April 2003.

[10] D. Aucsmith. Monocultures Are Hard To Find In Practice. *IEEE Security & Privacy*, 1(6):15–16, November/December 2003.

[11] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[12] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, October 2003.

[13] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[14] J. Bowring, A. Orso, and M. J. Harrold. Monitoring Deployed Software Using Software Tomography. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, November 2002.

[15] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the* $2^{nd}$ *Applied Cryptography and Network Security Conference (ACNS)*, pages 292–302, June 2004.

[16] S. Brilliant, J. C. Knight, and N. G. Leveson. Analysis of Faults in an N-Version Software Experiment. *IEEE Transactions on Software Engineering*, 16(2), February 1990.

[17] A. B. Brown and D. A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, 2003.

[18] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.

[19] M. Cai, K. Hwang, Y.-K. Kwok, S. Song, and Y. Chen. Collaborative Internet Worm Containment. *IEEE Security & Privacy Magazine*, 3(3):25–33, May/June 2005.

[20] G. Candea and A. Fox. Crash-only software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[21] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of OSDI*, 2002.

[22] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the* $14^{th}$ *USENIX Security Symposium*, pages 177–191, August 2005.

[23] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[24] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Can We Contain Internet Worms? In *Proceedings of the* $3^{rd}$ *Workshop on Hot Topics in Networks (HotNets)*, November 2004.

[25] C. 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. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[26] J. R. Cranall and F. T. Chong. Minos: Architectural Support for Software Security Through Control Data Integrity. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, December 2004.

[27] D. E. Geer. Monopoly Considered Harmful. *IEEE Security & Privacy*, 1(6):14 & 17, November/December 2003.

[28] A. K. Ghosh and J. M. Voas. Innoculating Software for Survivability. *Communications of the ACM*, 42(7), 1999.

[29] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Procedings of the 1996 USENIX Annual Technical Conference*, 1996.

[30] G. Goth. Addressing the Monoculture. *IEEE Security & Privacy*, 1(6):8–10, November/December 2003.

[31] J. Gray and D. Siewiorek. High-availability Computer Systems. *IEEE Computer*, 24(9):39–48, September 1991.

[32] R. Janakiraman, M. Waldvogel, and Q. Zhang. Indra: A peer-topeer approach to network intrusion detection and prevention. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, June 2003.

[33] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 272–280, October 2003.

[34] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proceedings of ACM SIGCOMM*, pages 61–72, August 2002.

[35] A. D. Keromytis, J. Parekh, P. N. Gross, G. Kaiser, V. Misra, J. Nieh, D. Rubenstein, and S. Stolfo. A Holistic Approach to Service Survivability. In *Proceedings of the 1st ACM Workshop on Survivable and Self-Regenerative Systems (SSRS)*, pages 11–22, October 2003.

[36] J. C. Knight and N. G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, January 1986.

[37] O. P. Kreidl and T. M. Frazier. Feedback Control Applied to Survivability: A Host-Based Autonomic Defense System. *IEEE Transactions on Reliability*, 2002.

[38] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[39] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 12–15 2005.

[40] M. Locasto, J. Parekh, S. Stolfo, A. Keromytis, T. Malkin, and V. Misra. Collaborative Distributed Intrusion Detection. Technical Report CUCS-012-04, Columbia University Department of Computer Science, 2004.

[41] M. Locasto, S. Sidiroglou, and A. D. Keromytis. Application Communities: Using Monoculture for Dependability. In *Proceedings of the* $1^{st}$ *Workshop on Hot Topics in System Dependability (HotDep)*, pages 288–292, June 2005.

[42] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the IEEE Infocom Conference*, April 2003.

[43] J. Newsome and D. Dong. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* $12^{th}$ *Annual Symposium on Network and Distributed System Security (SNDSS)*, February 2005.

[44] A. J. O'Donnell and H. Sethu. On Achieving Software Diversity for Improved Network Security using Distributed Coloring Algorithms. In *Proceedings of the* $11^{th}$ *ACM Conference on Computer and Communications Security (CCS)*, pages 121–131, October 2004.

[45] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma System: Continuous Evolution of Software After Deployment. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[46] P. Porras, L. Briesemeister, K. Skinner, K. Levitt, J. Rowe, and Y. A. Ting. A Hybrid Quarantine Defense. In *Proceedings of the ACM CCS Workshop on Rapid Malcode (WORM) 2004*, October 2004.

[47] V. Prevelakis. A Secure Station for Network Monitoring and Control. In *Proceedings of the* $8^{th}$ *USENIX Security Symposium*, August 1999.

[48] J. C. Reynolds, J. Just, L. Clough, and R. Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Generate-and-Test, and Generalization. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS)*, January 2003.

[49] J. Seward and N. Nethercote. Valgrind, an open-source memory debugger for x86-linux. `http://developer.kde.org/~sewardj/`.

[50] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the* $11^{th}$ *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[51] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building A Reactive Immune System for Software Services. In *Proceedings of the* $11^{th}$ *USENIX Annual Technical Conference*, pages 149–161, April 2005.

[52] M. Stamp. Risks of Monoculture. *Communications of the ACM*, 47(3):120, March 2004.

[53] A. Steven. Defeating compiler-level buffer overflow protection. *USENIX ;login:*, 30(3):59–71, June 2005.

[54] S. J. Stolfo. Worm and Attack Early Warning. *IEEE Security and Privacy*, 2(3):73–75, May-June 2004.

[55] J. D. Strunk, G. R. Goodson, A. G. Pennington, C. Soules, and G. Ganger. Intrusion Detection, Diagnosis, and Recovery with Self-Securing Storage. Technical Report CMU-CS-02-140, CMU Computer Science, May 2002.

[56] J. A. Whittaker. No Clear Answers on Monoculture Issues. *IEEE Security & Privacy*, 1(6):18–19, November/December 2003.

[57] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Intrusion Prevention. In *Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS)*, pages 123–130, February 2003.

[58] V. Yegneswaran, P. Barford, and S. Jha. Global Intrusion Detection in the DOMINO Overlay System. In *Proceedings of NDSS*, February 2004.

[59] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and Early Warning for Internet Worms. In *Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS)*, pages 190–199, October 2003.