# Shadow Honeypots

**Kostas G. Anagnostakis[1], Stelios Sidiroglou[2], Periklis Akritidis[1,3], Michalis Polychronakis[4], Angelos D. Keromytis[4], Evangelos P. Markatos[5]**

[1] Niometrics R&D, Singapore
*kostas@niometrics.com*
[2] Computer Science and Artificial Intelligence Laboratory, MIT, USA
*stelios@csail.mit.edu*
[3] University of Cambridge, UK
*pa280@cl.cam.ac.uk*
[4] Department of Computer Science, Columbia University, USA
*{mikepo, angelos}@cs.columbia.edu*
[5] Institute of Computer Science, Foundation for Research & Technology – Hellas, Greece
*markatos@ics.forth.gr*

**Abstract:** ***We present Shadow Honeypots, a novel hybrid architecture that combines the best features of honeypots and anomaly detection. At a high level, we use a variety of anomaly detectors to monitor all traffic to a protected network or service. Traffic that is considered anomalous is processed by a "shadow honeypot" to determine the accuracy of the anomaly prediction. The shadow is an instance of the protected software that shares all internal state with a regular ("production") instance of the application, and is instrumented to detect potential attacks. Attacks against the shadow are caught, and any incurred state changes are discarded. Legitimate traffic that was misclassified will be validated by the shadow and will be handled correctly by the system transparently to the end user. The outcome of processing a request by the shadow is used to filter future attack instances and could be used to update the anomaly detector. Our architecture allows system designers to fine-tune systems for performance, since false positives will be filtered by the shadow. We demonstrate the feasibility of our approach in a proof-of-concept implementation of the Shadow Honeypot architecture for the Apache web server and the Mozilla Firefox browser. We show that despite a considerable overhead in the instrumentation of the shadow honeypot (up to 20% for Apache), the overall impact on the system is diminished by the ability to minimize the rate of false-positives.***

**Keywords:** honeypots, anomaly detection

## 1. Introduction

Due to the increasing level of malicious activity seen on today's Internet, organizations are beginning to deploy mechanisms for detecting and responding to new attacks or suspicious activity, called Intrusion Prevention Systems (IPS). Since current IPSes use rule-based intrusion detection systems (IDS) such as Snort [1] to detect attacks, they are limited to protecting, for the most part, against already known attacks. As a result, new detection mechanisms are being developed for use in more powerful reactive-defense systems. The two primary such mechanisms are honeypots [2], [3], [4], [5], [6], [7] and anomaly detection systems (ADS) [8], [9], [10], [11], [12], [13]. In contrast with IDSes, honeypots and ADSes offer the possibility of detecting (and thus responding to) previously unknown attacks, also referred to as zero-day attacks.

Honeypots and anomaly detection systems offer different tradeoffs between accuracy and scope of attacks that can be detected, as shown in Figure 1. Honeypots can be heavily instrumented to accurately detect attacks, but depend on an attacker attempting to exploit a vulnerability against them. This makes them good for detecting scanning worms [14], [15], [3], but ineffective against manual directed attacks or topological and hit-list worms [16], [17]. Furthermore, honeypots can typically only be used for server-type applications. Anomaly detection systems can theoretically detect both types of attacks, but are usually much less accurate. Most such systems offer a tradeoff between false positive (FP) and false negative (FN) rates. For example, it is often possible to tune the system to detect more *potential* attacks, at an increased risk of *misclassifying* legitimate traffic (low FN, high FP); alternatively, it is possible to make an anomaly detection system more insensitive to attacks, at the risk of missing some real attacks (high FN, low FP). Because an ADS-based IPS can adversely affect legitimate traffic (e.g., drop a legitimate request), system designers often tune the system for low false positive rates, potentially misclassifying attacks as legitimate traffic.

We propose a novel hybrid approach that combines the best features of honeypots and anomaly detection, named *Shadow Honeypots*. At a high level, we use a variety of anomaly detectors to monitor all traffic to a protected network. Traffic that is considered anomalous is processed by a shadow honeypot. The shadow version is an instance of the protected application (e.g., a web server or client) that shares all internal state with a "normal" instance of the application, but is instrumented to detect potential attacks. Attacks against the shadow honeypot are caught and any incurred state changes are discarded. Legitimate traffic that was misclassified by the anomaly detector will be validated by the shadow honeypot and will be transparently handled correctly by the system (i.e., an HTTP request that was mistakenly flagged as suspicious will be served correctly). Our approach offers several advantages over stand-alone ADSes or honeypots:

- First, it allows system designers to tune the anomaly detection system for low false negative rates, minimizing the risk of misclassifying a real attack as legitimate traffic, since any false positives will be weeded out by the shadow honeypot.

- Second, and in contrast to typical honeypots, our approach can defend against attacks that are tailored against a specific site with a particular internal state. Honeypots may be blind to such attacks, since they are not typically mirror images of the protected application.
- Third, shadow honeypots can also be instantiated in a form that is particularly well-suited for protecting against client-side attacks, such as those directed against web browsers and P2P file-sharing clients.
- Finally, our system architecture facilitates easy integration of additional detection mechanisms.

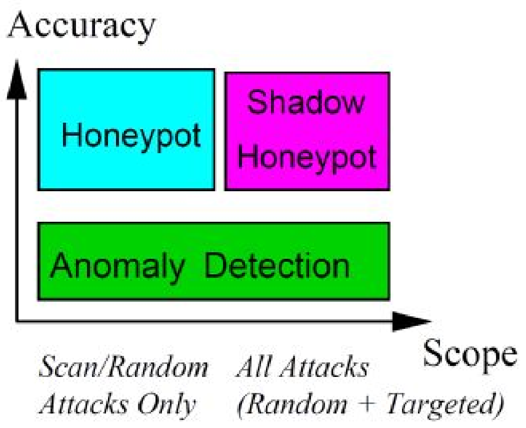

**Figure 1.** A simple classification of honeypots and anomaly detection systems, based on attack detection accuracy and scope of detected attacks. Targeted attacks may use lists of known (potentially) vulnerable servers, while scan-based attacks will target any system that is believed to run a vulnerable service. AD systems can detect both types of attacks, but with lower accuracy than a specially instrumented system (honeypot). However, honeypots are blind to targeted attacks, and may not see a scanning attack until after it has succeeded against the real server.

We apply the concept of shadow honeypots to a proof-of-concept implementation tailored against memory violation attacks. Specifically, we developed a tool that allows for automatic transformation of existing code into its "shadow version." The resulting code allows for traffic handling to happen through the regular or shadow version of the application, contingent on input derived from an array of anomaly detection sensors. When an attack is detected by the shadow version of the code, state changes effected by the malicious request are rolled back. Legitimate traffic handled by the shadow is processed successfully, albeit at higher latency. Note that the shadow may be an entirely separate process, possibly running on a different machine (*loose coupling*), or it may be a different thread running in the same address space (*tight coupling*). These two approaches reflect different tradeoffs in state-sharing overhead, ease of deployment, and transparency to the user.

In addition to the server-side scenario, we also investigate a client-targeting attack-detection scenario, unique to shadow honeypots, where we apply the detection heuristics to content retrieved by protected clients and feed any positives to shadow honeypots for further analysis. Unlike traditional honeypots, which are idle whilst waiting for active attackers to probe them, this scenario enables the detection of passive attacks, where the attacker lures a victim user to download malicious data. We use the recent `libpng` vulnerability of Mozilla [18] (which is similar to the buffer overflow vulnerability in the Internet Explorer's JPEG-handling logic) to demonstrate the ability of our system to protect client-side applications.

Our shadow honeypot prototype consists of several components. At the front-end of our system, we use a high-performance intrusion-prevention system based on the Intel IXP network processor and a set of modified Snort sensors running on normal PCs. The network processor is used as a smart load-balancer, distributing the workload to the sensors. The sensors are responsible for testing the traffic against a variety of anomaly detection heuristics, and coordinating with the IXP to tag traffic that needs to be inspected by shadow honeypots. This design leads to the scalability needed in high-end environments such as web server farms, as only a fraction of the servers need to incur the penalty of providing shadow honeypot functionality.

In our implementation, we have used a variety of anomaly detection techniques, including Abstract Payload Execution (APE) [10], the Earlybird algorithm [19], and network-level emulation [13]. The feasibility of our approach is demonstrated by examining both false-positive and true attack scenarios. We show that our system has the capacity to process all false positives generated by APE and EarlyBird and successfully detect attacks. Furthermore, it enhances the robustness of network-level emulation against advanced evasion attacks. We also show that when the anomaly detection techniques are tuned to increase detection accuracy, the resulting additional false positives are still within the processing budget of our system. More specifically, our benchmarks show that although instrumentation is expensive (20-50% overhead), the shadow version of the Apache Web server can process around 1300 requests per second, while the shadow version of the Mozilla Firefox client can process between 1 and 4 requests per second. At the same time, the front-end and anomaly detection algorithms can process a fully-loaded Gbit/s link, producing 0:3 to 0:5 false positives per minute when tuned for high sensitivity, which is well within the processing budget of our shadow honeypot implementation.

The remainder of this paper is organized as follows. Section 2 discusses the shadow honeypot architecture in greater detail. We describe our implementation in Section 3, and our experimental and performance results in Section 4. Some of the limitations of our approach are briefly discussed in Section 5. We give an overview of related work in Section 6, and conclude the paper with a summary of our work and plans for future work in Section 7.

## 2. Architecture

The Shadow Honeypot architecture is a systems approach to handling network-based attacks, combining filtering, anomaly detection systems, and honeypots in a way that exploits the best features of these mechanisms, while shielding their limitations. We focus on transactional applications, i.e., those that handle a series of discrete requests. Our architecture is not limited to server applications, but can be used for clientside applications such as web browsers and P2P clients. As shown in Figure 2, the architecture is composed of three main components: a filtering engine, an array of anomaly detection

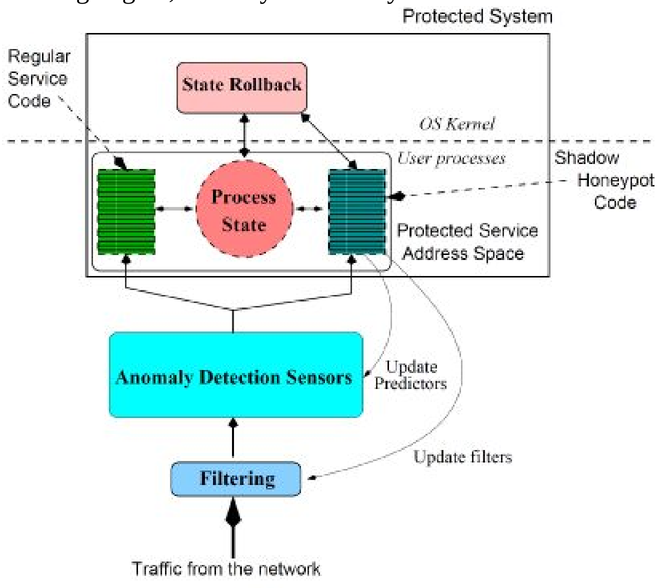

**Figure 2.** Shadow Honeypot architecture.

sensors, and the shadow honeypot, which validates the predictions of the anomaly detectors. The processing logic of the system is shown in Figure 3.

The filtering component blocks known attacks. Such filtering is done based either on payload content [20], [21] or on the source of the attack, if it can be identified with reasonable confidence (e.g., confirmed traffic bi-directionality). Effectively, the filtering component short-circuits the detection heuristics or shadow testing results by immediately dropping specific types of requests before any further processing is done.

Traffic passing the first stage is processed by one or more anomaly detectors. There are several types of anomaly detectors that may be used in our system, including payload analysis [9], [19], [22], [10], [13] and network behavior [23], [24]. Although we do not impose any particular requirements on the AD component of our system, it is preferable to tune such detectors towards high sensitivity (at the cost of increased false positives). The anomaly detectors, in turn, signal to the protected application whether a request is potentially dangerous.

Depending on this prediction by the anomaly detectors, the system invokes either the regular instance of the application or its *shadow*. The shadow is an instrumented instance of the application that can detect specific types of failures and rollback any state changes to a known (or presumed) good state, e.g., before the malicious request was processed. Because the shadow is (or should be) invoked relatively infrequently, we can employ computationally expensive instrumentation to detect attacks. The shadow and the regular application fully share state to avoid attacks that exploit differences between the two; we assume that an attacker can only interact with the application through the filtering and AD stages, i.e., there are no side-channels. The level of instrumentation used in the shadow depends on the amount of latency we are willing to impose on suspicious traffic (whether truly malicious or misclassified legitimate traffic). In our implementation, described in Section 3, we focus on memory-violation attacks, but any attack that can be determined algorithmically can be

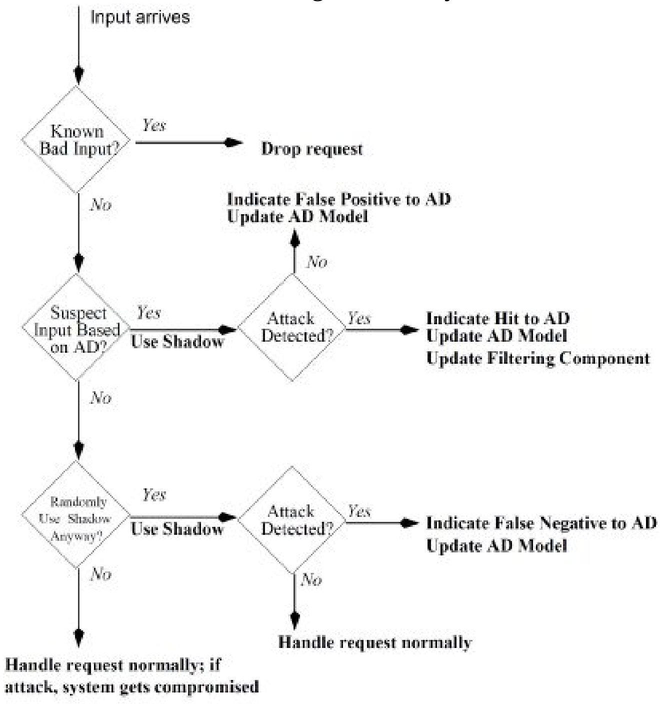

**Figure 3.** System workflow.

detected and recovered from, at the cost of increased complexity and potentially higher latency.

If the shadow detects an actual attack, we notify the filtering component to block further attacks. If no attack is detected, we update the prediction models used by the anomaly detectors. Thus, our system could in fact self-train and fine-tune itself using verifiably bad traffic and known mis-predictions, but this aspect of the approach is outside the scope of this paper.

As we mentioned above, shadow honeypots can be integrated with servers as well as clients. In this paper, we consider tight coupling with both server and client applications, where the shadow resides in the same address space as the protected application.

- **Tightly coupled with server.** This is the most practical scenario, in which we protect a server by diverting suspicious requests to its shadow. The application and the honeypot are tightly coupled, mirroring functionality and state. We have implemented this configuration with the Apache web server, described in Section 3.
- **Tightly coupled with client.** Unlike traditional honeypots, which remain idle while waiting for active attacks, this scenario targets passive attacks, where the

attacker lures a victim user to download data containing an attack, as with the recent buffer overflow vulnerability in Internet Explorer’s JPEG handling [25]. In this scenario, the context of an attack is an important consideration in replaying the attack in the shadow. It may range from data contained in a single packet to an entire flow, or even set of flows. Alternatively, it may be defined at the application layer. For our testing scenario using HTTP, the request/response pair is a convenient context.

Tight coupling assumes that the application can be modified. The advantage of this configuration is that attacks that exploit differences in the state of the shadow vs. the application itself become impossible. However, it is also possible to deploy shadow honeypots in a *loosely coupled* configuration, where the shadow resides on a different system and does not share state with the protected application. The advantage of this configuration is that management of the shadows can be “outsourced” to a third entity.

Note that the filtering and anomaly detection components can also be tightly coupled with the protected application, or may be centralized at a natural aggregation point in the network topology (e.g., at the firewall).

Finally, it is worth considering how our system would behave against different types of attacks. For most attacks we have seen thus far, once the AD component has identified an anomaly and the shadow has validated it, the filtering component will block all future instances of it from getting to the application. However, we cannot depend on the filtering component to prevent polymorphic or metamorphic [26] attacks. For low-volume events, the cost of invoking the shadow for each attack may be acceptable. For high-volume events, such as a Slammer-like outbreak, the system will detect a large number of correct AD predictions (verified by the shadow) in a short period of time; should a configurable threshold be exceeded, the system can enable filtering at the second stage, based on the unverified verdict of the anomaly detectors. Although this will cause some legitimate requests to be dropped, this could be acceptable for the duration of the incident. Once the number of (perceived) attacks seen by the ADS drop beyond a threshold, the system can revert to normal operation.

## 3. Implementation

### 3.3 Filtering and Anomaly Detection

During the composition of our system, we were faced with numerous design issues with respect to performance and extensibility. When considering the deployment of the shadow honeypot architecture in a high-performance environment, such as a Web server farm, where speeds of at least 1 Gbit/s are common and we cannot afford to misclassify traffic, the choice for off-the-shelf components becomes very limited. To the best of our knowledge, current solutions, both standalone PCs and network-processor-based network intrusion detection systems (NIDSes), are well under the 1 Gbit/s mark [27], [28].

Faced with these limitations, we considered a distributed design, similar in principle to [29], [30]: we use a network processor (NP) as a scalable, custom load balancer, and implement all detection heuristics on an array of (modified) Snort sensors running on standard PCs that are connected to the network processor board. We chose not to implement any of the detection heuristics on the NP for two reasons. First, currently available NPs are designed primarily for simple forwarding and lack the processing capacity required for speeds in excess of 1 Gbit/s. Second, they remain harder to program and debug than standard general purpose processors. For our implementation, we used the IXP1200 network processor. A high-level view of our implementation is shown in Figure 4.

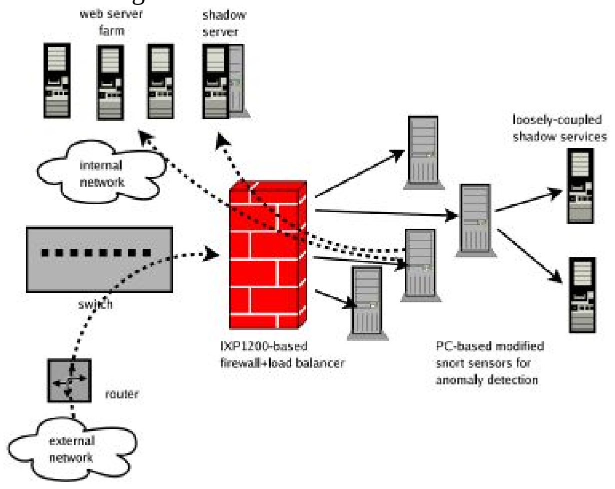
web server farm
shadow server
internal network
loosely-coupled shadow services
switch
IXP1200-based firewall+load balancer
PC-based modified snort sensors for anomaly detection
router
external network

**Figure 4.** High-level diagram of prototype shadow honeypot implementation.

A primary function of the anomaly detection sensor is the ability to divert potentially malicious requests to the shadow honeypot. For web servers in particular, a reasonable definition of the attack context is the HTTP request. For this purpose, the sensor must construct a request, run the detection heuristics, and forward the request depending on the outcome. This processing must be performed at the HTTP level thus an HTTP proxy-like function is needed. We implemented the anomaly detection sensors for the tightly-coupled shadow server case by augmenting an HTTP proxy with ability to apply the APE detection heuristic on incoming requests and route them according to its outcome.

For the shadow client scenario, we use an alternative solution based on passive monitoring. Employing the proxy approach in this situation would be prohibitively expensive, in terms of latency, since we only require detection capabilities. For this scenario, we reconstruct the TCP streams of HTTP connections and decode the HTTP protocol to extract suspicious objects.

As part of our proof-of-concept implementation we have used three anomaly detection heuristics: payload sifting, abstract payload execution, and network-level emulation. Payload sifting as developed in [19] derives fingerprints of rapidly spreading worms by identifying popular substrings in network traffic. It is a prime example of an anomaly detection based system that is able to detect novel attacks at the expense of false positives. However, if used in isolation (e.g., outside our shadow honeypot environment) by the time it has reliably detected a worm epidemic, it is very likely that many systems would have already been compromised. This may reduce its usage potential in the tightly-coupled server protection scenario without external help. Nevertheless, if fingerprints generated by a distributed payload sifting system are disseminated to interested parties that run shadow honeypots locally, matching traffic against such fingerprints can be of use as a detection heuristic in the shadow honeypot system. Of further interest is the ability to use this technique in the loosely-coupled shadow server scenario, although we do not further consider this scenario here.

The second heuristic we have implemented is buffer overflow detection via abstract payload execution (APE), as proposed in [10]. The heuristic detects buffer overflow attacks by searching for sufficiently long sequences of valid instructions in network traffic. Long sequences of valid instructions can appear in non-malicious data, and this is where the shadow honeypot fits in. Such detection mechanisms are particularly attractive because they are applied to individual attacks and will trigger detection upon encountering the first instance of an attack, unlike many anomaly detection mechanisms that must witness multiple attacks before flagging them as anomalous.

Finally, as discussed in Section 3.3, the third heuristic we use is network-level emulation [13], [31], a detection method that scans network traffic streams for the presence of previously unknown polymorphic shellcode. The approach is based on the execution of all potential malicious instruction sequences found in the inspected traffic on a NIDS-embedded CPU emulator. Based on a behavioral heuristic, the detection algorithm can discriminate between the execution of benign and malicious code.

## 3.4 Shadow Honeypot Creation

The creation of a shadow honeypot is based on a code-transformation tool that takes as input the original application source code and “weaves” into it the shadow honeypot code. In this paper, we focus on memory-violation errors and show source-code transformations that detect buffer overflows, although other types of failures can be caught (e.g., input that causes illegal memory dereferences) with the appropriate instrumentation, but at the cost of higher complexity and larger performance bottleneck. For the code transformations we use TXL [32], a hybrid functional and rule-based language which is well-suited for performing source-to-source transformation and for rapidly prototyping new languages and language processors. The grammar responsible for parsing the source input is specified in a notation similar to Extended Backus-Naur (BNF). In our prototype, called DYBOC, we use TXL for C-to-C transformations with the GCC C front-end.

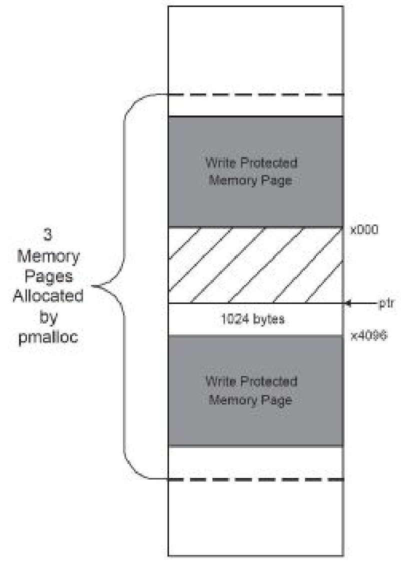
3 Memory Pages Allocated by pmalloc
Write Protected Memory Page
x000
ptr
1024 bytes
x4096
Write Protected Memory Page

**Figure 5.** Example of `pmalloc()`-based memory allocation: the trailer and edge regions (above and below the write-protected pages) indicate “waste” memory. This is needed to ensure that `mprotect()` is applied on complete memory pages.

*Original code*

```
int func()
{
    char buf[100];
...
    other_func(buf, sizeof(buf);
...
    return 0;
}
```

*Modified code*

```
int func()
{
    char *buf;
    char _buf[100];
    if (shadow_enable())
      buf = pmalloc(100);
    else
      buf = _buf;
...
    other_func(buf, sizeof(_buf));
...
    if (shadow_enable()) {
      pfree(buf);
    }
    return 0;
}
```

**Figure 6.** Transforming a function to its shadow-supporting version. The `shadow_enable()` macro simply checks the status of a shared-memory variable (controlled by the anomaly detection system) on whether the shadow honeypot should be executing instead of the regular code.

The instrumentation itself is conceptually straightforward: we move all static buffers to the heap by dynamically allocating the buffer upon entering the function in which it was previously declared; we de-allocate these buffers upon exiting the function, whether implicitly (by reaching the end of the function body) or explicitly (through a return statement). We take care to properly handle the `sizeof` construct, a fairly straightforward task with TXL. Pointer aliasing is not a problem, since we instrument the allocated memory regions; any illegal accesses to these will be caught.

For memory allocation, we use our own version of `malloc()`, called `pmalloc()`, that allocates two additional zero-filled, write-protected pages that bracket the requested buffer, as shown in Figure 5. The guard pages are `mmap()`'ed from `/dev/zero` as read-only. As `mmap()` operates at memory page granularity, every memory request is rounded up to the nearest page. The pointer that is returned by `pmalloc()` can be adjusted to immediately catch any buffer overflow or underflow depending on where attention is focused. This functionality is similar to that offered by the ElectricFence memory-debugging library, the difference being that `pmalloc()` catches both buffer overflow and underflow attacks. Because we `mmap()` pages from `/dev/zero`, we do not waste physical memory for the guards (just page-table entries). Memory is wasted, however, for each allocated buffer, since we allocate to the next closest page. While this can lead to considerable memory waste, we note that this is only incurred when executing in shadow mode, and in practice has proven easily manageable.

Figure 6 shows an example of such a translation. Buffers that are already allocated via `malloc()` are simply switched to `pmalloc()`. This is achieved by examining declarations in the source and transforming them to pointers where the size is allocated with a `malloc()` function call. Furthermore, we adjust the C grammar to free the variables before the function returns. After making changes to the standard ANSI C grammar that allow entries such as `malloc()` to be inserted between declarations and statements, the transformation step is trivial. For single-threaded, non-reentrant code, it is possible to only use `pmalloc()` once for each previously-static buffer. Generally, however, this allocation needs to be done each time the function is invoked.

Any overflow (or underflow) on a buffer allocated via `pmalloc()` will cause the process to receive a Segmentation Violation (SEGV) signal, which is caught by a signal handler we have added to the source code in `main()`. The signal handler simply notifies the operating system to abort all state changes made by the process while processing this request. To do this, we added a new system call to the operating system, `transaction()`. This is conditionally (as directed by the shadow `enable()` macro) invoked at three locations in the code:

- Inside the main processing loop, prior to the beginning of handling of a new request, to indicate to the operating system that a new transaction has begun. The operating system makes a backup of all memory page permissions, and marks all heap memory pages as read-only. As the process executes and modifies these pages, the operating system maintains a copy of the original page and allocates a new page (which is given the permissions the original page had from the backup) for the process to use, in exactly the same way copy-on-write works in modern operating system. Both copies of the page are maintained until `transaction()` is called again, as we describe below. This call to `transaction()` must be placed manually by the programmer or system designer.
- Inside the main processing loop, immediately after the end of handling a request, to indicate to the operating system that a transaction has successfully completed. The operating system then discards all original copies of memory pages that have been modified during processing this request. This call to `transaction()` must also be placed manually.
- Inside the signal handler that is installed automatically by our tool, to indicate to the operating system that an exception (attack) has been detected. The operating system then discards all modified memory pages by restoring the original pages.

Although we have not implemented this, a similar mechanism can be built around the filesystem by using a private copy of the buffer cache for the process executing in shadow mode. The only difficulty arises when the process must itself communicate with another process while servicing a request; unless the second process is also included in the transaction definition (which may be impossible, if it is a remote process on another system), overall system state may change without the ability to roll it back. For example, this may happen when a web server communicates with a remote back-end database. Our system does not currently address this, i.e., we assume that any such state changes are benign or irrelevant (e.g., a DNS query). Specifically for the case of a back-end database, these inherently support the concept of a transaction rollback, so it is possible to undo any changes.

The signal handler may also notify external logic to indicate that an attack associated with a particular input from a specific source has been detected. The external logic may then instantiate a filter, either based on the network source of the request or the contents of the payload [20].

### 3.5 Using Feedback to Improve Network-level Detection

A significant benefit stemming from the combination of network-level anomaly detection techniques with host-level attack prevention mechanisms is that it allows for increasing the detection accuracy of current network-level detectors. This improvement may go beyond simply increasing the sensitivity of the detector and then mitigating the extra false positives through the shadow honeypot. In certain cases, it is also possible to enhance the robustness of the anomaly detection algorithm itself against evasion attacks. In this section, we describe how shadow honeypots enhance the detection ability of network-level emulation, one of the detection techniques that we have used in our implementation.

Network-level emulation [13], [31] is a passive network monitoring approach for the detection of previously unknown polymorphic shellcode. The approach relies on a NIDSembedded CPU emulator that executes every potential instruction sequence in the inspected traffic, aiming to identify the execution behavior of polymorphic shellcode. The principle behind network-level emulation is that the machine code interpretation of arbitrary data results to random code, which, when it is attempted to run on an actual CPU, usually crashes soon, e.g., due to the execution of an illegal instruction. In contrast, if some network request actually contains a polymorphic shellcode, then the shellcode runs normally, exhibiting a certain detectable behavior.

Network-level emulation does not rely on any exploit or vulnerability specific signatures, which allows the detection of previously unknown attacks. Instead, it uses a generic heuristic that matches the runtime behavior of polymorphic shellcode. At the same time, the actual execution of the attack code on a CPU emulator makes the detector robust to evasion techniques such as highly obfuscated or self-modifying code. Furthermore, each input is inspected autonomously, which makes the approach effective against targeted attacks, while from our experience so far with real-world deployments, it has not produced any false positives.

The detector inspects either or both directions of each network flow, which may contain malicious requests towards vulnerable services, or malicious content served by some compromised server towards a vulnerable client. Each input is mapped to a random memory location in the virtual address space of the emulator, as shown in Figure 7. Since the exact position of the shellcode within the input stream is not known in advance, the emulator repeats the execution multiple times, starting from each and every position of the stream. Before the beginning of a new execution, the state of the CPU is randomized, while any accidental memory modifications in the addresses where the attack vector has been mapped to are rolled back after the end of each execution. The execution of polymorphic shellcode is identified by two key behavioral characteristics: the execution of some form of GetPC code, and the occurrence of several read operations from the memory addresses of the input stream itself, as illustrated in Figure 7. The GetPC code is used for finding the absolute address of the injected code, which is mandatory for subsequently decrypting the encrypted payload, and involves the execution of some instruction from the `call` or `fstenv` instruction groups.

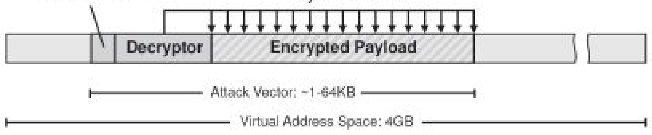

**Figure 7.** A typical execution of a polymorphic shellcode using network-level emulation.

There exist situations in which the execution of benign inputs, which are interpreted by the emulator as random code, might not stop soon, or even not at all, due to the accidental formation of loop structures that may execute for a very large number of iterations. To avoid extensive performance degradation due to stalling on such seemingly "endless" loops, if the number of executed instructions for a given input reaches a certain execution threshold, then the execution is terminated.

This unavoidable precaution introduces an opportunity for evasion attacks against the detection algorithm through the placement of a seemingly endless loop before the decryptor code. An attacker could construct a decryptor that spends millions of instructions just for reaching the execution threshold before revealing any signs of polymorphic behavior. We cannot simply skip the execution of such loops, since the loop body may perform a crucial computation for the subsequent correct execution of the decoder, e.g., computing the decryption key.

Such "endless" loops are a well-known problem in the area of dynamic code analysis [33], and we are not aware of any effective solution so far. However, employing network-level emulation as a first-stage detector for shadow honeypots mitigates this problem. Without shadow honeypot support, the network-level detector does not alert on inputs that reach the execution threshold without exhibiting signs of malicious behavior, which can potentially result to false negatives. In contrast, when coupling network-level emulation with shadow honeypots, such undecidable inputs can be treated more conservatively by considering them as potentially dangerous, and redirecting them to the shadow version of the protected service. If an undecidable input indeed corresponds to a code injection attack, then it will be detected by the shadow honeypot. In Section 4.3 we show, through analysis of real network traffic, that the number of such streams that are undecidable in reasonable time (and thus have to be forwarded to the shadow) is a small, manageable fraction of the overall traffic.

## 4. Experimental Evaluation

We have tested our shadow honeypot implementation against a number of exploits, including a recent Mozilla PNG bug and several Apache-specific exploits. In this section, we report on performance benchmarks that illustrate the efficacy of our implementation.

First, we measure the cost of instantiating and operating shadow instances of specific services using the Apache web server and the Mozilla Firefox web browser. Second, we evaluate the filtering and anomaly detection components, and determine the throughput of the IXP1200-based load balancer as well as the cost of running the detection heuristics. Third, we look at the false positive rates and the trade-offs associated with detection performance. Based on these results, we determine how to tune the anomaly detection heuristics in order to increase detection performance while not exceeding the budget allotted by the shadow services.

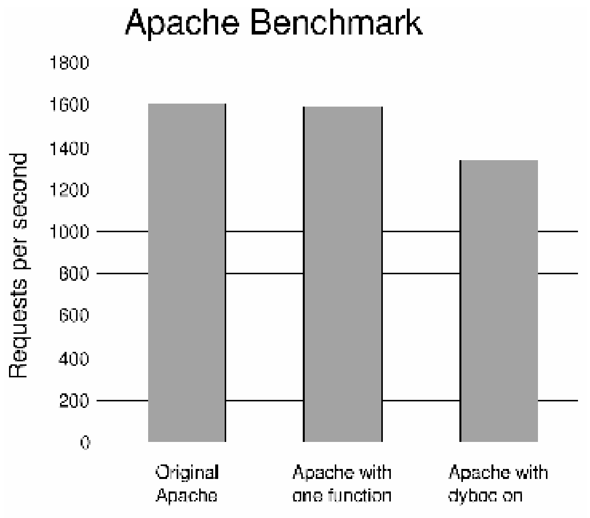

**Figure 8.** Apache benchmark results.

### 4.1 Performance of Shadow Services

*Apache:* In this experiment, we determine the workload capacity of the shadow honeypot environment, using DYBOC on the Apache web server, version 2.0.49. We chose Apache due to its popularity and source code availability. Basic Apache functionality was tested, omitting additional modules. The tests were conducted on a PC with a 2GHz Intel P4 processor and 1GB of RAM, running Debian Linux (2.6.5- 1 kernel).

We used ApacheBench [34], 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.

Figure 8 illustrates the requests per second that Apache can handle. There is a 20.1% overhead for the patched version of Apache over the original, which is expected since the majority of the patched buffers belong to utility functions that are not heavily used. This result is an indication of the worst-case analysis, since all the protection flags were enabled; although the performance penalty is high, it is not outright prohibitive for some applications. For the instrumentation of a single buffer and a vulnerable function that is invoked once per HTTP transaction, the overhead is 1.18%.

Of further interest is the increase in memory requirements for the patched version. A naive implementation of `pmalloc()` would require two additional memory pages for each transformed buffer. Full transformation of Apache translates into 297 buffers that are allocated with `pmalloc()`, adding an overhead of 2.3MB if all of these buffers are invoked simultaneously during program execution. When protecting `malloc()`'ed buffers, the amount of required memory can skyrocket.

To avoid this overhead, we use an `mmap()` based allocator. The two guard pages are `mmap()`'ed write-protected from `/dev/zero`, without requiring additional physical memory to be allocated. Instead, the overhead of our mechanism is 2 page-table entries (PTEs) per allocated buffer, plus one file descriptor (for `/dev/zero`) per program. As most modern processors use an MMU cache for frequently used PTEs, and since the guard pages are only accessed when a fault occurs, we expect their impact on performance to be small.

*Mozilla Firefox:* For the evaluation of the client case, we used the Mozilla Firefox browser. For the initial validation tests, we back-ported the recently reported `libpng` vulnerability [18] that enables arbitrary code execution if Firefox (or any application using `libpng`) attempts to display a specially crafted PNG image. Interestingly, this example mirrors a recent vulnerability of Internet Explorer, and JPEG image handling [35], which again enabled arbitrary code execution when displaying specially crafted images.

In the tightly-coupled scenario, the protected version of the application shares the address space with the unmodified version. This is achieved by transforming the original source code with our DYBOC tool. Suspicious requests are tagged by the ADS so that they are processed by the protected version of the code as discussed in Section 3.2.

For the loosely-coupled case, when the AD component marks a request for processing on the shadow honeypot, we launch the instrumented version of Firefox to replay the request. The browser is configured to use a null X server as provided by `Xvfb`. All requests are handled by a transparent proxy that redirects these requests to an internal Web server. The Web server then responds with the objects served by the original server, as captured in the original session. The workload that the shadow honeypot can process in the case of Firefox is determined by how many responses per second a browser can process and how many different browser versions can be checked.

Our measurements show that a single instance of Firefox can handle about one request per second with restarting after processing each response. Doing this only after detecting a successful attack improves the result to about four requests per second. By restarting, we avoid the accumulation of various pop-ups and other side-effects. Unlike the server scenario, instrumenting the browser does not seem to have any significant impact on performance. If that was the case, we could have used the rollback mechanism discussed previously to reduce the cost of launching new instances of the browser.

We further evaluate the performance implications of fully instrumenting a web browser. These observations apply to both loosely-coupled and tightly-coupled shadow honeypots. Web browsing performance was measured using a Mozilla Firefox 1.0 browser to run a benchmark based on the i-Bench benchmark suite [36]. i-Bench is a comprehensive, cross-platform benchmark that tests the performance and capability of Web clients. Specifically, we use a variant of the benchmark that allows for scrolling of a web page and uses cookies to store the load times for each page. Scrolling is performed in order to render the whole page, providing a pessimistic emulation of a typical attack. The benchmark consists of a sequence of 10 web pages containing a mix of text and graphics; the benchmark was ran using both the scrolling option and the standard page load mechanisms.

For the standard page load configuration, the performance degradation for instrumentation was 35%. For the scrolling configuration, where in addition to the page load time, the time taken to scroll through the page is recorded, the overhead was 50%.

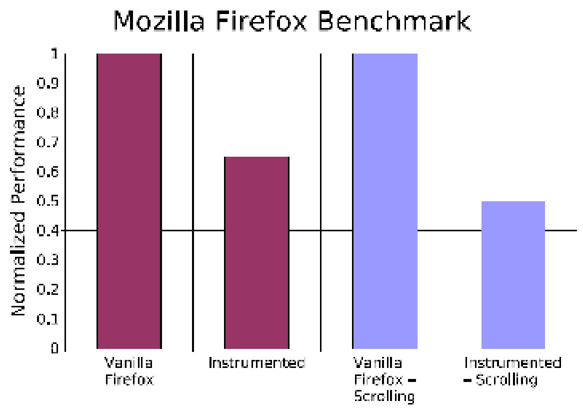

**Figure 9.** Normalized Mozilla Firefox benchmark results using a modified version of i-Bench.

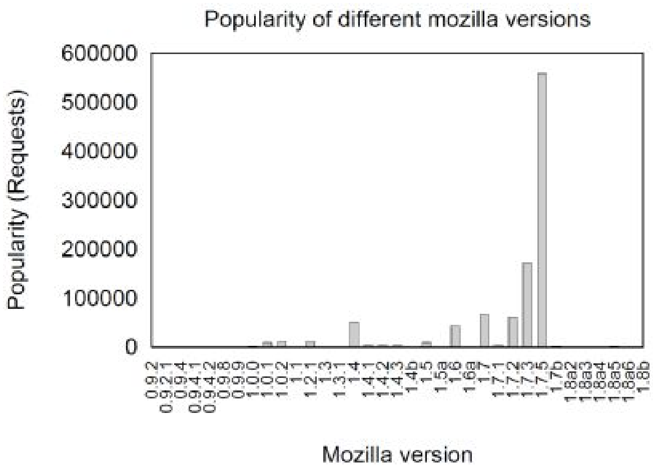

**Figure 10.** Popularity of different Mozilla versions, as measured in the logs of the CIS Department Web server at the University of Pennsylvania.

The results follow our intuition as more calls to `malloc()` are required to fully render the page. Figure 9 illustrates the normalized performance results. It should be noted that depending on the browser implementation (whether the entire page is rendered on page load) mechanisms such at the automatic scrolling need to be implemented in order to protect against targeted attacks. Attackers may hide malicious code in unrendered parts of a page or in javascript code activated by user-guided pointer movement.

How many different browser versions would have to be checked by the system? Figure 10 presents some statistics concerning different versions of Mozilla. The statistics were collected over a five-week period from the CIS Department web server at the University of Pennsylvania. As evidenced by the figure, one can expect to check up to six versions of a particular client. We expect that this distribution will be more stabilized around final release versions and expect to minimize the number of different versions that need to be checked based on their popularity.

### 4.2 Filtering and Anomaly Detection

*IXP1200-based firewall/load-balancer:* We first determine the performance of the IXP1200-based firewall/load balancer. The IXP1200 evaluation board we use has two Gigabit Ethernet interfaces and eight Fast Ethernet interfaces. The Gigabit Ethernet interfaces are used to connect to the internal and external network and the Fast Ethernet interfaces to communicate with the sensors. A set of client workstations is used to generate traffic through the firewall. The firewall forwards traffic to the sensors for processing and the sensors determine if the traffic should be dropped, redirected to the shadow honeypot, or forwarded to the internal network.

Previous studies [37] have reported forwarding rates of at least 1600 Mbit/s for the IXP1200, when used as a simple forwarder/router, which is sufficient to saturate a Gigabit Ethernet interface. Our measurements show that despite the added cost of load balancing, filtering, and coordinating with the sensors, the firewall can still handle the Gigabit interface at line rate.

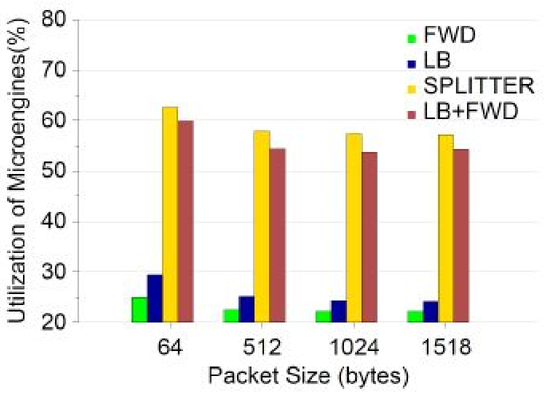

**Figure 11.** Utilization(%) of the IXP1200 Microengines, for forwarding-only (FWD), load-balancing-only (LB), both (LB+FWD), and full implementation (FULL), in stress-tests with 800 Mbit/s worst-case 64-byte-packet traffic.

To gain insight into the actual overhead of our implementation, we carry out a second experiment using Intel's cycle-accurate IXP1200 simulator. We assume a clock frequency of 232 MHz for the IXP1200, and an IX bus configured to be 64- bit wide with a clock frequency of 104 MHz. In the simulated environment, we obtain detailed utilization measurements for the microengines of the IXP1200. The results are shown in Figure 11. The results show that even at line rate with worst-case traffic, the implementation is quite efficient as the microengines operate at 50.9%-71.5% of their processing capacity.

*PC-based sensor performance:* In this experiment, we measure the throughput of the PC-based sensors that cooperate with the IXP1200 for analyzing traffic and performing anomaly detection. We use a 2.66 GHz Pentium IV Xeon processor with hyper-threading disabled. The PC has 512 Mbytes of DDR DRAM at 266 MHz. The PCI bus is

64- bit wide clocked at 66 MHz. The host operating system is Linux (kernel version 2.4.22, Red-Hat 9.0).

We use LAN traces to stress-test a single sensor running a modified version of Snort that, in addition to basic signature matching, provides the hooks needed to coordinate with the IXP1200 as well as the APE and payload sifting heuristics. We replay the traces from a remote system through the IXP1200 at different rates to determine the maximum loss-free rate (MLFR) of the sensor. For the purpose of this experiment, we connected a sensor to the second Gigabit Ethernet interface of the IXP1200 board.

**Table 1**: Sensor throughput for different detection mechanisms.

| Detection Method | Throughput/Sensor |
|---|---|
| Content Matching | 225 Mbit/s |
| APE | 190 Mbit/s |
| Payload Sifting | 268 Mbit/s |

The measured throughput of the sensor for signature matching using APE and Earlybird is shown in Table 1. The throughput per sensor ranges between 190 Mbit/s (APE) and 268 Mbit/s (payload sifting), while standard signature matching can be performed at 225 Mbit/s. This means that we need at least 4-5 sensors behind the IXP1200 for each of these mechanisms. Note, however, that these results are rather conservative and based on unoptimized code, and thus only serve the purpose of providing a ballpark figure on the cost of anomaly detection.

*False positive vs. detection rate trade-offs:* We determine the workload that is generated by the AD heuristics, by measuring the false positive rate. We also consider the trade-off between false positives and detection rate, to demonstrate how the AD heuristics could be tuned to increase detection rate in our shadow honeypot environment. We use the payload sifting implementation from [38], and the APE algorithm from [10]. The APE experiment corresponds to a scenario with a tightly-coupled shadow server, while the payload sifting experiment examines a loosely-coupled shadow honeypot scenario that can be used for worm detection.

We run the modified Snort sensor implementing APE and payload sifting on packet-level traces captured on an enterprise LAN with roughly 150 hosts. Furthermore, the traces contain several instances of the Welchia worm. APE was applied on the URIs contained in roughly one-billion HTTP requests gathered by monitoring the same LAN.

Figure 12 demonstrates the effects of varying the distinct destinations threshold of the content sifting AD on the false positives (measured in requests to the shadow services per minute) and the (Welchia worm) detection delay (measured in ratio of hosts in the monitored LAN infected by the time of the detection).

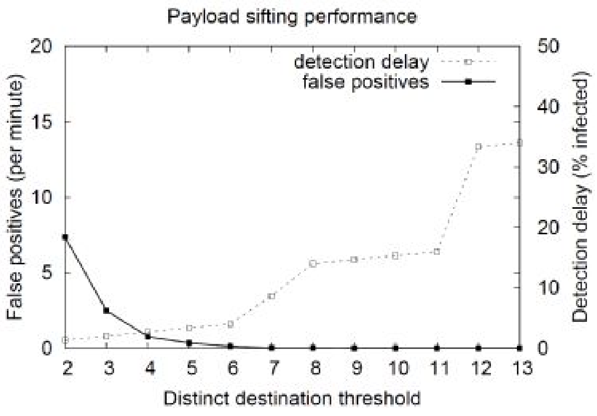

**Figure 12.** False positives for payload sifting.

Increasing the threshold means more attack instances are required for triggering detection, and therefore increases the detection delay and reduces the false positives. It is evident that to achieve a zero false positives rate without shadow honeypots we must operate the system with parameters that yield a suboptimal detection delay. The detection rate for APE is the minimum sled length that it can detect and depends on the sampling factor and the MEL parameter (the number of valid instructions that trigger detection). A high MEL value means less false positives due to random valid sequences but also makes the heuristic blind to sleds of smaller lengths.

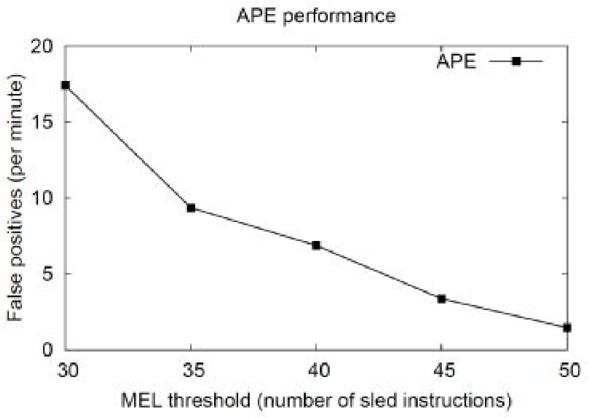

**Figure 13.** False positives for APE.

Figure 13 shows the effects of MEL threshold on the false positives. APE can be used in a tightly coupled scenario, where the suspect requests are redirected to the instrumented server instances. The false positives (measured in requests to the shadow services per minute by each of the normal services under maximum load) can be handled easily by a shadow honeypot. APE alone has false positives for the entire range of acceptable operational parameters; it is the combination with shadow honeypots that removes the problem.

## 4.3 Fine-tuning Network-level Emulation

In this scheme, the redirection criterion is whether a given input reaches the CPU execution threshold of the network-level detector. Since most of the time the system will not be under attack, and thus the inspected inputs will be benign, an issue that we should take into account is how often a benign inspected input may look “suspicious” and causes a redirection to the shadow honeypot. If the fraction of such undecidable inputs is large, then the shadow server may be overloaded with a higher request rate than it can normally handle. To evaluate this effect, we used full payload traces of real network traffic captured at ICS-FORTH and the University of Crete. The set of traces contains more than 2.5 million user requests to ports 80, 445, and 139, which are related to the most exploited vulnerabilities.

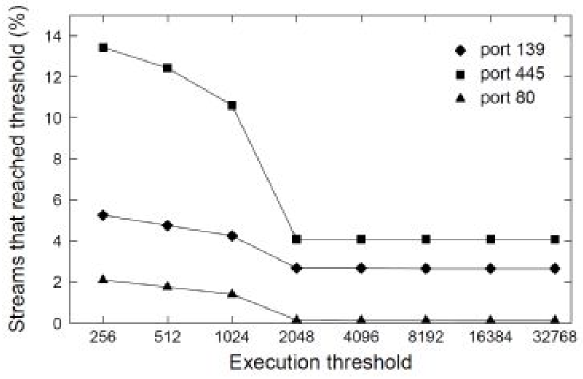

**Figure 14.** Percentage of benign network streams reaching the execution threshold of the network-level detector.

Figure 14 shows the percentage of streams with at least one instruction sequence that, when executed on the CPU emulator of network-level detector, reached the given execution threshold. As the execution threshold increases, the number of streams that reach it decreases. This effect occurs only for low threshold values, due to large code blocks with no branch instructions that are executed linearly. For example, the execution of linear code blocks with more than 256 but less than 512 valid instructions is terminated before reaching the end when using a threshold of 256 instructions, but completes correctly with a threshold of 512 instructions. However, the occurrence probability of such blocks is reversely proportional to their length, due to the illegal or privileged instructions that accidentally occur in random code. Thus, the percentage of streams that reach the execution threshold stabilizes beyond the value of 2048. After this value, the execution threshold is reached solely due to instruction sequences with “endless” loops, which usually require a prohibitive number of instructions for the slow CPU emulator in order to complete.

Fortunately, for an execution threshold above 2048 instructions, which allows for accurate polymorphic shellcode detection with a decent operational throughput [13], the fraction of streams that reach the execution threshold is only around 4% for port 445, 2.6% for port 139, and 0.1% for port 80. Binary traffic (ports 445 and 139) is clearly more likely to result to an instruction sequence that reaches the execution threshold in contrast to the mostly ASCII traffic of port 80. In any case, even in the worst case of binary-only traffic, the percentage of benign streams that reach the execution threshold is very small, so the extra overhead incurred to the shadow server is modest.

## 5. Limitations

There are two limitations of the shadow honeypot design presented in this paper that we are aware of. The effectiveness of the rollback mechanism depends on the proper placement of calls to `transaction()` for committing state changes, and the latency of the detector. The detector used in this paper can instantly detect attempts to overwrite a buffer, and therefore the system cannot be corrupted. Other detectors, however, may have higher latency, and the placement of commit calls is critical to recovering from the attack. Depending on the detector latency and how it relates to the cost of implementing rollback, one may have to consider different approaches. The trade-offs involved in designing such mechanisms are thoroughly examined in the fault-tolerance literature (c.f. [39]).

Furthermore, the loosely coupled client shadow honeypot is limited to protecting against relatively static attacks. The honeypot cannot effectively emulate user behavior that may be involved in triggering the attack, for example, through DHTML or Javascript. The loosely coupled version is also weak against attacks that depend on local system state on the user’s host that is difficult to replicate. This is not a problem with tightly coupled shadows, because we accurately mirror the state of the real system. In some cases, it may be possible to mirror state on loosely coupled shadows as well, but we have not considered this case in the experiments presented in this paper.

## 6. Related Work

Much of the work in automated attack reaction has focused on the problem of network worms, which has taken truly epidemic dimensions (pun intended). For example, the system described in [24] detects worms by monitoring probes to unassigned IP addresses (“dark space”) or inactive ports and computing statistics on scan traffic, such as the number of source/destination addresses and the volume of the captured traffic. By measuring the increase on the number of source addresses seen in a unit of time, it is possible to infer the existence of a new worm when as little as 4% of the vulnerable machines have been infected. A similar approach for isolating infected nodes inside an enterprise network [40] is taken in [23], where it was shown that as little as four probes may be sufficient in detecting a new port-scanning worm.

Smirnov and Chiueh [41] describe an approximating algorithm for quickly detecting scanning activity that can be efficiently implemented in hardware. Newsome et al. [42] describe a combination of reverse sequential hypothesis testing and credit-based connection throttling to quickly detect and quarantine local infected hosts. These systems are effective only against scanning worms (not topological, or “hit-list” worms), and rely on the assumption that most scans will result in non-connections. As such, they are susceptible to false positives, either accidentally (e.g., when a host is joining a peer-to-peer network such as Gnutella, or during a temporary network outage) or on purpose (e.g., a malicious web page with many links to images in random/notused IP addresses). Furthermore, it may be possible for several instances of a worm to collaborate in providing the illusion of several successful connections, or to use a list of known repliers to blind the anomaly detector. Another algorithm for finding fast-spreading worms using 2-level filtering based on sampling from the set of distinct source-destination pairs is described in [43].

Wu et al. [22] describe an algorithm for correlating packet payloads from different traffic flows, towards deriving a worm signature that can then be filtered [44]. The technique is promising, although further improvements are required to allow it to operate in real time. Earlybird [19] presents a more practical algorithm for doing payload sifting, and correlates these with a range of unique sources generating infections and destinations being targeted. However, polymorphic and metamorphic worms [26] remain a challenge; Spinelis [45] shows that it is an NP-hard problem. Vigna et al. [46] discuss a method for testing detection signatures against mutations of known vulnerabilities to determine the quality of the detection model and mechanism. Polygraph [47] attempts to detect polymorphic exploits by identifying common invariants among the various attack instances, such as return addresses, protocol framing and poor obfuscation.

Toth and Kruegel [10] propose to detect buffer overflow payloads (including previously unseen ones) by treating inputs received over the network as code fragments. They use restricted symbolic execution to show that legitimate requests will appear to contain relatively short sequences of valid x86 instruction opcodes, compared to attacks that will contain long sequences. They integrate this mechanism into the Apache web server, resulting in a small performance degradation. STRIDE [48] is a similar system that seeks to detect polymorphic NOP-sleds in buffer overflow exploits. [49] describes a hybrid polymorphic-code detection engine that combines several heuristics, including NOP-sled detector and abstract payload execution.

HoneyStat [3] runs sacrificial services inside a virtual machine, and monitors memory, disk, and network events to detect abnormal behavior. For some classes of attacks (e.g., buffer overflows), this can produce highly accurate alerts with relatively few false positives, and can detect zero-day worms. Although the system only protects against scanning worms, “active honeypot” techniques [4] may be used to make it more difficult for an automated attacker to differentiate between HoneyStats and real servers. FLIPS (Feedback Learning IPS) [50] is a similar hybrid approach that incorporates a supervision framework in the presence of suspicious traffic. Instruction-set randomization is used to isolate attack vectors, which are used to train the anomaly detector. The authors of [51] propose to enhance NIDS alerts using host-based IDS information. Nemean [52] is an architecture for generating semantics-aware signatures, which are signatures aware of protocol semantics (as opposed to general byte strings). Shield [20] is a mechanism for pushing to workstations vulnerability-specific, application-aware filters expressed as programs in a simple language.

The Internet Motion Sensor [7] is a distributed blackhole monitoring system aimed at measuring, characterizing, and tracking Internet-based threats, including worms. [53] explores the various options in locating honeypots and correlating their findings, and their impact on the speed and accuracy in detecting worms and other attacks. [54] shows that a distributed worm monitor can detect non-uniform scanning worms two to four times as fast as a centralized telescope [55], and that knowledge of the vulnerability density of the population can further improve detection time. However, other recent work has shown that it is relatively straightforward for attackers to detect the placement of certain types of sensors [56], [57]. Shadow Honeypots [58] are one approach to avoiding such mapping by pushing honeypot-like functionality at the end hosts.

The HACQIT architecture [59], [60], [61], [62] uses various sensors to detect new types of attacks against secure servers, access to which is limited to small numbers of users at a time. Any deviation from expected or known behavior results in the possibly subverted server to be taken off-line. A sandboxed instance of the server is used to conduct “clean room” analysis, comparing the outputs from two different implementations of the service (in their prototype, the Microsoft IIS and Apache web servers were used to provide application diversity). Machine-learning techniques are used to generalize attack features from observed instances of the attack. Content-based filtering is then used, either at the firewall or the end host, to block inputs that may have resulted in attacks, and the infected servers are restarted. Due to the feature-generalization approach, trivial variants of the attack will also be caught by the filter. [8] takes a roughly similar approach, although filtering is done based on port numbers, which can affect service availability. Cisco’s Network-Based Application Recognition (NBAR) [21] allows routers to block TCP sessions based on the presence of specific strings in the TCP stream. This feature was used to block CodeRed probes, without affecting regular web-server access. Porras et al. [63] argue that hybrid defenses using complementary techniques (in their case, connection throttling at the domain gateway and a peer-based coordination mechanism), can be much more effective against a wide variety of worms.

DOMINO [64] 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 [65]. Malkhi and Reiter [66] describe an architecture 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.

Suh et al. [67], propose a hardware-based solution that can be used to thwart control-transfer attacks and restrict executable instructions by monitoring "tainted" input data. In order to identify "tainted" data, they rely on the operating system. If the processor detects the use of this tainted data as a jump address or an executed instruction, it raises an exception that can be handled by the operating system. The authors do not address the issue of recovering program execution and suggest the immediate termination of the offending process. DIRA [68] is a technique for automatic detection, identification and repair of control-hijaking attacks. This solution is implemented as a GCC compiler extension that transforms a program's source code adding heavy instrumentation so that the resulting program can perform these tasks. Unfortunately, the performance implications of the system make it unusable as a front line defense mechanism. Song and Newsome [69] propose dynamic taint analysis for automatic detection of overwrite attacks. Tainted data is monitored throughout the program execution and modified buffers with tainted information will result in protection faults. Once an attack has been identified, signatures are generated using automatic semantic analysis. The technique is implemented as an extension to Valgrind and does not require any modifications to the program's source code but suffers from severe performance degradation. One way of minimizing this penalty is to make the CPU aware of memory tainting [70]. Crandall et al. report on using a taint-based system for capturing live attacks in [71].

The Safe Execution Environment (SEE) [72] allows users to deploy and test untrusted software without fear of damaging their system. This is done by creating a virtual environment where the software has read access to the real data; all writes are local to this virtual environment. The user can inspect these changes and decide whether to commit them or not. We envision use of this technique for unrolling the effects of filesystem changes in our system, as part of our future work plans. A similar proposal is presented in [73] for executing untrusted Java applets in a safe "playground" that is isolated from the user's environment.

## 7. Conclusion

We have described a novel approach to dealing with zeroday attacks by combining features found today in honeypots and anomaly detection systems. The main advantage of this architecture is providing system designers the ability to fine tune systems with impunity, since any false positives (legitimate traffic) will be filtered by the underlying components. We have implemented this approach in an architecture called Shadow Honeypots. In this approach, we employ an array of anomaly detectors to monitor and classify all traffic to a protected network; traffic deemed anomalous is processed by a shadow honeypot, a protected instrumented instance of the application we are trying to protect. Attacks against the shadow honeypot are detected and caught before they infect the state of the protected application. This enables the system to implement policies that trade off between performance and risk, retaining the capability to re-evaluate this trade-off effortlessly.

Our experience so far indicates that despite the considerable cost of processing suspicious traffic on our Shadow Honeypots and overhead imposed by instrumentation, such systems are capable of sustaining the overall workload of protecting services such as a Web server farm, as well as vulnerable Web browsers. We have also demonstrated how the impact on performance can be minimized by reducing the rate of false positives and tuning the AD heuristics using a feedback loop with the shadow honeypot. We believe that shadow honeypots can form the foundation of a type of application community.

## Acknowledgments

This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-06-2-0221, and by the National Science Foundation under NSF Grant CNS-09-14845. Evangelos Markatos is also with the University of Crete.

## References

[1] M. Roesch. Snort: Lightweight intrusion detection for networks. In Proceedings of USENIX LISA, November 1999. (software available from http://www.snort.org/).

[2] N. Provos. A Virtual Honeypot Framework. In Proceedings of the 13th USENIX Security Symposium, pages 1–14, August 2004.

[3] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. HoneyStat: Local Worm Detection Using Honepots. In Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID), pages 39–58, October 2004.

[4] V. Yegneswaran, P. Barford, and D. Plonka. On the Design and Use of Internet Sinks for Network Abuse Monitoring. In Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID), pages 146–165, October 2004.

[5] L. Spitzner. Honeypots: Tracking Hackers. Addison-Wesley, 2003.

[6] J. G. Levine, J. B. Grizzard, and H. L. Owen. Using Honeynets to Protect Large Enterprise Networks. IEEE Security & Privacy, 2(6):73– 75, Nov./Dec. 2004.

[7] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In Proceedings of the 12th ISOC Symposium on Network and Distributed Systems Security (SNDSS), pages 167–179, February 2005.

[8] T. Toth and C. Kruegel. Connection-history Based Anomaly Detection. In Proceedings of the IEEE Workshop on Information Assurance and Security, June 2002.

[9] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In Proceedings of the 7th International Symposium on Recent Advanced in Intrusion Detection (RAID), pages 201–222, September 2004.

[10] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID), October 2002.

[11] M. Bhattacharyya, M. G. Schultz, E. Eskin, S. Hershkop, and S. J. Stolfo. MET: An Experimental System for Malicious Email Tracking. In Proceedings of the New Security Paradigms Workshop (NSPW), pages 1–12, September 2002.

[12] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), pages 251–261, October 2003.

[13] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Network-level polymorphic shellcode detection using emulation. In Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), pages 54–73, July 2006.

[14] 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.

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

[16] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In Proceedings of the 11th USENIX Security Symposium, pages 149–167, August 2002.

[17] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In Proceedings of the ACM Workshop on Rapid Malcode (WORM), pages 33–42, October 2004.

[18] US-CERT Technical Cyber Security Alert TA04-217A: Multiple Vulnerabilities in libpng. http://www.us-cert.gov/cas/techalerts/TA04-217A.html, August 2004.

[19] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI), December 2004.

[20] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In Proceedings of the ACM SIGCOMM Conference, pages 193–204, August 2004.

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

[22] H. Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In Proceedings of the 13th USENIX Security Symposium, pages 271–286, August 2004.

[23] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In Proceedings of the IEEE Symposium on Security and Privacy, May 2004.

[24] J. Wu, S. Vangala, L. Gao, and K. Kwiat. An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques. In Proceedings of the ISOC Symposium on Network and Distributed System Security (SNDSS), pages 143–156, February 2004.

[25] Microsoft Security Bulletin MS04-028, September 2004. http://www.microsoft.com/technet/security/Bulletin/MS04-028.mspx.

[26] P. Ször and P. Ferrie. Hunting for Metamorphic. Technical report, Symantec Corporation, June 2003.

[27] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In Proceedings of the 3rd Workshop on Network Processors and Applications (NP3), February 2004.

[28] L. Schaelicke, T. Slabach, B. Moore, and C. Freeland. Characterizing the Performance of Network Intrusion Detection Sensors. In Proceedings of Recent Advances in Intrusion Detection (RAID), September 2003.

[29] Top Layer Networks. http://www.toplayer.com.

[30] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In Proceedings of the IEEE Symposium on Security and Privacy, pages 285–294, May 2002.

[31] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Emulation-based detection of non-self-contained polymorphic shellcode. In Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID), September 2007.

[32] A. J. Malton. The Denotational Semantics of a Functional Tree-Manipulation Language. Computer Languages, 19(3):157–168, 1993.

[33] P. Ször. The Art of Computer Virus Research and Defense. Addison- Wesley Professional, February 2005.

[34] ApacheBench: A complete benchmarking and regression testing suite. http://freshmeat.net/projects/apachebench/, July 2003.

[35] Microsoft Security Bulletin MS04-028: Buffer Overrun in JPEG Processing Could Allow Code Execution. http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx, September 2004.

[36] i-Bench. http://http://www.veritest.com/benchmarks/i-bench/default.asp.

[37] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), pages 216–229, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[38] P. Akritidis, K. Anagnostakis, and E. P. Markatos. Efficient contentbased fingerprinting of zero-day worms. In Proceedings of the IEEE International Conference on Communications (ICC), May 2005.

[39] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery

protocols in message-passing systems. ACM Comput. Surv., 34(3):375–408, 2002.

[40] S. Staniford. Containment of Scanning Worms in Enterprise Networks. Journal of Computer Security, 2005. (to appear).

[41] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In Proceedings of the 13th USENIX Security Symposium, pages 29–44, August 2004.

[42] S. E. Schechter, J. Jung, and A. W. Berger. Fast Detection of Scanning Worm Infections. In Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID), October 2004.

[43] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In Proceedings of the 12th ISOC Symposium on Network and Distributed Systems Security (SNDSS), pages 149–166, February 2005.

[44] 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.

[45] D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. IEEE Transactions on Information Theory, 49(1):280– 284, January 2003.

[46] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS), pages 21–30, October 2004.

[47] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In Proceedings of the IEEE Security & Privacy Symposium, pages 226–241, May 2005.

[48] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic Sled Detection through Instruction Sequence Analysis. In Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC), June 2005.

[49] U. Payer, P. Teufl, and M. Lamberger. Hybrid Engine for Polymorphic Shellcode Detection. In Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), July 2005.

[50] M. Locasto, K. Wang, A. Keromytis, and S. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In Proceedings of the 8th Symposium on Recent Advances in Intrusion Detection (RAID), September 2005.

[51] H. Dreger, C. Kreibich, V. Paxson, and R. Sommer. Enhancing the Accuracy of Network-based Intrusion Detection with Host-based Context. In Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), July 2005.

[52] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An Architecture for Generating Semantics-Aware Signatures. In Proceedings of the 14th USENIX Security Symposium, pages 97–112, August 2005.

[53] E. Cook, M. Bailey, Z. M. Mao, and D. McPherson. Toward Understanding Distributed Blackhole Placement. In Proceedings of the ACM Workshop on Rapid Malcode (WORM), pages 54–64, October 2004.

[54] M. A. Rajab, F. Monrose, and A. Terzis. On the Effectiveness of Distributed Worm Monitoring. In Proceedings of the 14th USENIX Security Symposium, pages 225–237, August 2005.

[55] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denialof- Service Activity. In Proceedings of the 10th USENIX Security Symposium, pages 9–22, August 2001.

[56] J. Bethencourt, J. Franklin, and M. Vernon. Mapping Internet Sensors With Probe Response Attacks. In Proceedings of the 14th USENIX Security Symposium, pages 193–208, August 2005.

[57] Y. Shinoda, K. Ikai, and M. Itoh. Vulnerabilities of Passive Internet Threat Monitors. In Proceedings of the 14th USENIX Security Symposium, pages 209–224, August 2005.

[58] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. P. Markatos, and A. D. Keromytis. Detecting Targetted Attacks Using Shadow Honeypots. In Proceedings of the 14th USENIX Security Symposium, pages 129–144, August 2005.

[59] J. E. Just, L. A. Clough, M. Danforth, K. N. Levitt, R. Maglich, J. C. Reynolds, and J. Rowe. Learning Unknown Attacks – A Start. In Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID), October 2002.

[60] J. C. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. The Design and Implementation of an Intrusion Tolerant System. In Proceedings of the International Conference on Dependable Systems and Networks (DSN), June 2002.

[61] J.C. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. Online Intrusion Protection by Detecting Attacks with Diversity. In Proceedings of the 16th Annual IFIP 11.3 Working Conference on Data and Application Security Conference, April 2002.

[62] 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.

[63] P. Porras, L. Briesemeister, K. Levitt, J. Rowe, and Y.-C. A. Ting. A Hybrid Quarantine Defense. In Proceedings of the ACM Workshop on Rapid Malcode (WORM), pages 73–82, October 2004.

[64] V. Yegneswaran, P. Barford, and S. Jha. Global Intrusion Detection in the DOMINO Overlay System. In Proceedings of the ISOC Symposium on Network and Distributed System Security (SNDSS), February 2004.

[65] 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.

[66] 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.

[67] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. SIGOPS Operating Systems Review, 38(5):85–96, 2004.

[68] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In Proceedings of the 12th ISOC Symposium on Network and Distributed System Security (SNDSS), February 2005.

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

[70] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and C. Verbowski. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In Proceedings of the International Conference on Dependable Systems and Networks (DSN), pages 378–387, June 2005.

[71] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences Using Minos as a Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities. In Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), July 2005.

[72] W. Sun, Z. Liang, R. Sekar, and V. N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In Proceedings of the 12th ISOC Symposium on Network and Distributed Systems Security (SNDSS), February 2005.

[73] D. Malkhi and M. K. Reiter. Secure Execution of Java Applets Using a Remote Playground. IEEE Trans. Softw. Eng., 26(12):1197– 1209, 2000.