FLIPS: Hybrid Adaptive Intrusion Prevention

Michael E. Locasto, Ke Wang, Angelos D. Keromytis, and Salvatore J. Stolfo

Department of Computer Science, Columbia University
1214 Amsterdam Avenue
Mailcode 0401
New York, NY 10027
+1 212 939 7177
{*locasto, kewang, angelos, sal*}@*cs.columbia.edu*

Abstract. Intrusion detection systems are fundamentally passive and fail–open. Because their primary task is classification, they do nothing to prevent an attack from succeeding. An intrusion prevention system (IPS) adds protection mechanisms that provide fail–safe semantics, automatic response capabilities, and adaptive enforcement. We present *FLIPS* (Feedback Learning IPS), a hybrid approach to host security that prevents binary code injection attacks. It incorporates three major components: an anomaly-based classifier, a signature-based filtering scheme, and a supervision framework that employs Instruction Set Randomization (ISR). Since ISR prevents code injection attacks and can also precisely identify the injected code, we can tune the classifier and the filter via a learning mechanism based on this feedback. Capturing the injected code allows FLIPS to construct signatures for zero-day exploits. The filter can discard input that is anomalous *or* matches known malicious input, effectively protecting the application from additional instances of an attack – even zero-day attacks or attacks that are metamorphic in nature. FLIPS does not require a known user base and can be deployed transparently to clients and with minimal impact on servers. We describe a prototype that protects HTTP servers, but FLIPS can be applied to a variety of server and client applications.

Keywords: Adaptive Response, Intrusion Prevention, Intrusion Tolerance

1 Introduction

One key problem for network defense systems is the inability to automatically mount a reliable, targeted, and adaptive response [21]. This problem is magnified when exploits are delivered via previously unseen inputs. Network defense systems are usually composed of network-based IDS's and packet filtering firewalls. These systems have shortcomings that make it difficult for them to identify and characterize new attacks and respond intelligently to them.

Since IDS's passively classify information, they can enable but not enact a response. Both signature-based and anomaly-based approaches to classification merely warn that an attack may have occurred. Attack prevention is a task often

left to a firewall, and it is usually accomplished by string matching signatures of known malicious content or dropping packets according to site policy. Of course, successfully blocking the correct traffic requires a flexible and well defined policy. Furthermore, signature matching large amounts of network traffic often requires specialized hardware and presumes the existence of accurate signatures. In addition, encrypted and tunneled network traffic poses problems for both firewalls and IDS's. To compound these problems, since neither IDS's or firewalls know for sure how a packet is processed at an end host, they may make an incorrect decision [10].

These obstacles motivate the argument for placing protection mechanisms closer to the end host (*e.g.*, distributed firewalls [11]). This approach to system security can benefit not only enterprise-level networks, but home users as well. The principle of "defense-in-depth" suggests that traditional perimeter defenses like firewalls be augmented with host-based protection mechanisms. This paper advocates one such system that employs a hybrid anomaly and signature detection scheme to adaptively react to new exploits.

1.1   Hybrid Detection

In general, detection systems that rely solely on signatures cannot enable a defense against previously unseen attacks. On the other hand, anomaly-based classifiers can recognize new behavior, but are often unable to distinguish between previously unseen "good" behavior and previously unseen "bad" behavior. This blind spot usually results in a high false positive rate and requires that these classifiers be extensively trained.

A hybrid approach to detection can provide the basis for an Intrusion Prevention System (IPS): an automated response system capable of stopping an attack from succeeding. The core of our hybrid system is an anomaly-based classifier that incorporates feedback to both tune its models and automatically generate signatures of known malicious behavior. Our anomaly detector is based on PayL [38], but other classifiers can be used [17].

The biggest obstacle for a hybrid system is the source of the feedback information. Ideally, it should be automated and transparent to users. For example, the feedback to email spam classifiers may be a user hitting a button in their email client that notifies the mail server to reconsider an inappropriately classified email as spam. This feedback loop is an example of supervised online learning and distributes the burden of supervision to users of the system. The feedback mechanism in our system facilitates unsupervised online learning. The source of information is based on an *x86* emulator, STEM [29], that is augmented to protect processes with Instruction Set Randomization.

1.2   Instruction Set Randomization

ISR is the process of creating a unique execution environment to effectively negate the success of code-injection attacks. This unique environment is created

by performing some reversible transformation on the instruction set; the transformation is driven by a random key for each executable. The binary is then decoded during runtime with the appropriate key.

Since an attacker crafts an exploit to match some expected execution environment (*e.g. x86* machine instructions) and the attacker cannot easily reproduce the transformation for his exploit code, the injected exploit code will most likely be invalid for the specialized execution environment. The mismatch between the language of the exploit code and the language of the execution environment causes the exploit to fail. Without knowledge of the key, otherwise valid (from the attacker's point of view) machine instructions resolve to invalid opcodes or eventually crash the program by accessing illegal memory addresses. Previous approaches to ISR [3] [12] have proved successful in defeating code injection attacks. Such techniques are typically combined with address-space obfuscation [4] to prevent "jump into libc" attacks.

Randomizing an instruction set requires that the execution environment possess the ability to de-randomize or decode the binary instruction stream during runtime. For machine code, this requirement means that either the processor hardware must contain the decoding logic or that the processor be emulated in software. STEM minimizes the cost of executing in software by *selectively* emulating parts of an application. During the application's runtime, control can freely switch between the real and the virtual processors. By carefully selecting the pieces of the application that are emulated, it is possible to minimize the runtime overhead of the emulation.

This practical form of ISR allows us to capture injected code and correlate it with input that has been classified as anomalous. Barrantes *et al.* [3] show that code injection attacks against protected binaries fail within a few bytes (two or three instructions) of control flow switching to the injected code. Therefore, the code pointed to by the instruction pointer at the time the program halts is (with a high probability) malicious code. We can extract this code and send it to our filter to create a new signature and update our classifier's model.

1.3 Contributions

The main contribution of this paper is a complete system that uses information confirming an attack to assist a classifier and update a signature-based filter. Filtering strategies are rarely based solely on anomaly detection; anomaly-based classifiers usually have a high false positive rate. However, when combined with feedback information confirming an attack, the initial classification provided by the anomaly detector can assist in creating a signature. This signature can then be deployed to the filter to block further malicious input. It is important to note that our protection mechanism catches the exploit code itself. Having the exploit code allows very precise signature creation and tuning of the classifier. Furthermore, this signature can be exchanged with other instances of this system via a centralized trusted third party or a peer-to-peer network. Such information exchange [7], [14] can potentially inoculate the network against a zero-day worm attack [1], [13], [18], [35].

We present the design of FLIPS, a host-based application-level firewall that adapts to new malicious input. Our prototype implementation adjusts its filtering capability based on feedback from two sources: (a) an anomaly-based classifier [38] that is specialized to the content flows for a specific host and (b) a binary supervision framework [29] that prevents code-injection attacks via ISR and captures injected code. The details of our design are presented in Section 3, and we describe the prototype implementation of the system for an HTTP server in Section 4. We discuss related work in Section 2, our experimental validation of FLIPS in Section 5, directions for future research in Section 6, and conclude the paper in Section 7.

2 Related Work

Augmenting detection systems with an adaptive response mechanism is an emerging area of research. Intrusion prevention, the design and selection of mechanisms to automatically respond to network attacks, has recently received an amount of attention that rivals its equally difficult sibling intrusion detection. Response systems vary from the low–tech (manually shut down misbehaving machines) to the highly ambitious (on the fly "vaccination", validation, and replacement of infected software). In the middle lies a wide variety of practical techniques, promising technology, and nascent research.

The system proposed by Anagnostakis *et al.* [2] has many of the same goals as FLIPS. However, there are a number of differences in architecture and implementation. Most importantly, our use of ISR allows FLIPS to detect and stop all instances of code injection attacks, not just stack-based buffer overflows. Also, FLIPS is meant to protect a single host without the need for a "shadow."

Two other closely related systems are the network worm vaccine architecture [28] and the HACQIT system [25]. More recently, researchers have investigated transparently detecting malicious email attachments [27] with techniques similar to ours and [28]. HACQIT employs a pair of servers in which the outputs of the primary and secondary server are compared. If the outputs are different, then a failure has occurred. The HACQIT system then attempts to classify the input that caused this error and generalize a rule for blocking it. The network and email worm vaccine architectures propose the use of honeypot and auxiliary servers, respectively, to provide supervised environments where malware can infect instrumented instances of an application. The system can then construct a fix based on the observed infection vector and deploy the fix to the production server. In the case of the email worm vaccine, the email can be silently dropped, stripped of the attachment, or rejected.

In contrast, FLIPS is meant to protect a single host without the need for additional infrastructure. Since the system is modular, it is an implementation choice whether or not to distribute the components across multiple machines. FLIPS also precisely identifies attack code by employing ISR. It does not need to correlate input strings against other services or try to deduce where attack code is placed inside a particular input request. In addition, our anomaly detection

component can construct models of both good and "bad" inputs to detect and block slight variants of malicious input.

2.1 Code Injection and ISR

One of the major contributions of this work is the use of a practical form of ISR. The basic premise of ISR [3] [12] is to prevent code injection attacks [23] from succeeding by creating unique execution environments for individual processes. Code injection is not limited to overflowing stack buffers or format strings. Other injection vectors include web forms that allow arbitrary SQL expressions (a solution to this problem using SQL randomization is proposed in [5]), CGI scripts that invoke shell programs based on user input, and log files containing character sequences capable of corrupting the terminal display.

Our *x86* emulator STEM can selectively derandomize portions of an instruction stream, effectively supporting two different instruction sets at the same time. Various processors support the ability to emulate or execute other instruction sets. These abilities could conceivably be leveraged to provide hardware support for ISR. For example, the Transmeta Crusoe chip[1] employs a software layer for interpreting code into its native instruction format. The PowerPC chip employs "Mixed-Mode" execution[2] for supporting the Motorola 68k instruction set. Likewise, the ARM chip can switch freely between executing its regular instruction set and executing the Thumb instruction set. A processor that supports ISR could use a similar capability to switch between executing regular machine instructions and randomized machine instructions. In fact, this is almost exactly what STEM does in software. Having hardware support for ISR would obviate the need for (along with the performance impact of) software-level ISR.

2.2 Anomaly Detection and Remediation

Anomaly-based classification is a powerful method of detecting inputs that are probably malicious. This conclusion is based on the assumption that malicious inputs are rare in the normal operation of the system. However, since a system can evolve over time, it is also likely that new *non-malicious* inputs will be seen [9] [32]. Indeed, some work [16] has shown that it is possible to evade anomaly-based classifiers. Therefore, anomaly-based detectors [38] [17] require an additional source of information that can confirm or reject the initial classification. Pietraszek [22] presents a method that uses supervised machine learning to tune an alert classification system based on observations of a human expert. Sommer and Paxon [33] explore a related problem: how to augment signature-based NIDS to make use of context when applying signatures.

FLIPS receives feedback from an emulator that monitors the execution of a vulnerable application. If the emulator tries to execute injected code, it catches the fault and notifies the classifier and filter. It can then terminate and restart

---

[1] `http://www.transmeta.com/crusoe/codemorphing.html`
[2] `http://developer.apple.com/documentation/mac/runtimehtml/RTArch-75.html`

the process, or simulate an error return from the current function. While our prototype system employs ISR, there are many other types of program supervision that can provide useful information. Each could be employed in parallel to gather as much information as possible. These approaches include input taint tracking [36] [20], program shepherding [15], (a similar technique is proposed in [24]) and compiler-inserted checks [31]. One advantage of FLIPS's feedback mechanism is that it can identify with high confidence the binary code of the attack. In an interesting approach to detection, Toth and Kruegel [37] and Stig *et al.* [34]) consider the problem of finding *x86* code in network packets.

Effective remediation strategies remain a challenge. The typical response of protection mechanisms has traditionally been to terminate the attacked process. This approach is unappealing for a variety of reasons; to wit, the loss of accumlated state is an overarching concern. Several other approaches are possible, including failure oblivious computing [26], STEM's error virtualization [29], DIRA's rollback of memory updates [31], crash-only software [6], and data structure repair [8]. Remediation strategies sometimes include the deployment of firewall rules that block malicious input. The most common form of this strategy is based on dropping packets from "malicious" hosts. Even with whitelists to counter spoofing, this strategy is too coarse a mechanism. Our system allows for the generation of very precise signatures because the actual exploit code can be caught "in the act."

Automatically creating reliable signatures of zero-day exploits is the focus of intense research efforts [13]. Signatures of viruses and other malware are currently produced by manual inspection of the malware source code. Involving humans in the response loop dramatically lengthens response time and does nothing to stop the initial infection. In addition, deployed signatures and IDS rules do nothing to guard against new threats. Singh *et al.* describe the Earlybird system for automatically generating worm signatures and provide a good overview of the shortcomings of current approaches to signature generation [30].

3 FLIPS – A Learning Application Filter

While we describe our implementation of FLIPS in Section 4, this section provides an overview of the design space for a host-based intrusion prevention system. The system is composed of a number of modules that provide filtering, classification, supervision, and remediation services. We can use the metrics proposed by Smirnov and Chiueh [31] to classify FLIPS: it detects attacks, identifies the attack vector, and provides an automatic repair mechanism.

The goal of the system is to provide a modular and compact application-level firewall with the ability to automatically learn and drop confirmed zero-day attacks. In addition, the system should be able to generate zero-day worm and attack signatures, even for slightly metamorphic attack input. We tune the anomaly detection by catching code injection attacks with our supervision component. Only attacks that actually inject and execute code are confirmed

as malicious and fed back to the anomaly detector and filter. As a result, only confirmed attacks are dropped in the future.

3.1 FLIPS Design

The design of FLIPS is based on two major components: a filtering proxy and an application supervision framework. A major goal of the design is to keep the system modular *and* deployable on a single host. Figure 1 shows a high-level view of this design. The protected application can be either a server waiting for requests or a client program receiving input. Input to a client program or requests to a server are passed through the filtering proxy and dropped if deemed malicious. If the supervision framework detects something wrong with the protected application, it signals the filter to update its signatures and models. Although server replies and outgoing client traffic can also be modeled and filtered, our current implementation does not perform this extra step. Outgoing filtering is useful in protecting a client application by stopping information leaks or the spread of self-propagating malware.
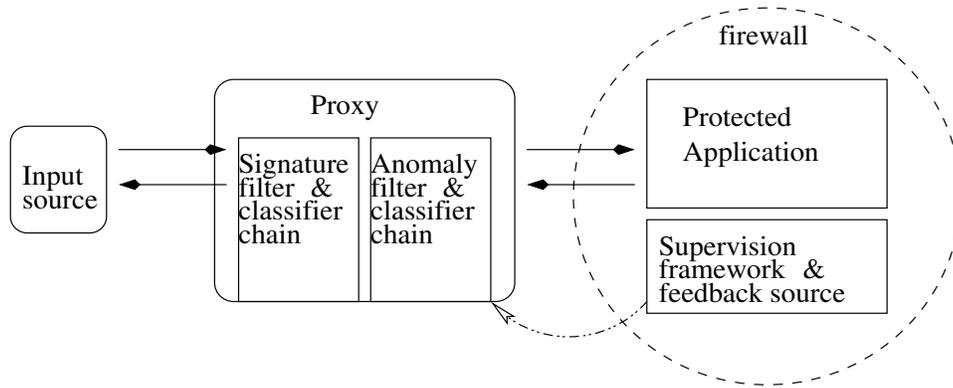

Fig. 1. *General Architecture of FLIPS.* Requests are passed through a filtering proxy and dropped if deemed malicious. The application should be protected by a packet filtering firewall that only allows the local proxy instance to contact the application. The application processes the requests and sends the response back through the proxy. If the input causes a code injection attack, the supervision framework contacts the proxy with the injected code and the proxy updates its models and signatures.

The function of the proxy is to grade or score the input and optionally drop it. The proxy is a hybrid of the two major classification schemes, and its subcomponents reflect this dichotomy. A chain of signature-based filters can score and drop a request if it matches known malicious data, and a chain of anomaly-based classifiers can score and drop the input if it is outside the normal model. Either

chain can allow the request to pass even if it is anomalous or matches previous malicious input. The default policy for our prototype implementation is to only drop requests that match a signature filter. Requests that the anomaly classifier deems suspicious are copied to a cache and forwarded on to the application. We adopt this stance to avoid dropping requests that the anomaly component mislabels (false positives). The current implementation only drops requests that have been confirmed to be malicious to the protected application and requests that are closely related to such inputs.

The function of the application supervision framework is to provide a way to stop an exploit, automatically repair the exploited vulnerability, and report information about an exploit back to the filters and classifiers. Similar to the filtering and classification chains, the supervision framework could include a number of host-based monitors to provide a wide array of complementary feedback information to the proxy. Our prototype implementation is based on one type of monitor (ISR) and will only provide feedback information related to code-injection attacks. Many other types of attacks are possible, and whether something is an attack or not often depends on context. FLIPS's design allows for an array of more complicated monitors. STEM allows the application to recover from a code injection attack by simulating an error return from the emulated function after notifying the proxy about the injected code.

3.2   Threat Model

In this work, we assume a threat model that closely matches that of previous ISR efforts. Specifically, we assume that an attacker does not have access to the randomized binary or the key used to effect achieve this randomization. These objects are usually stored on a system's disk or in system memory; we assume the attacker does not have local access to these resources. In addition, the attacker's intent is to inject code into a running process and thereby gain control over the process by virtue of the injected instructions. ISR is especially effective against these types of threats because it interferes with an attacker's ability to automate the attack. The entire target population executes binaries encoded under keys unique to each instance. A successful breach on one machine does not weaken the security of other target hosts.

3.3   Caveats and Limitations

While the design of FLIPS is quite flexible, the nature of host-based protection and our choices for a prototype implementation impose several limitations. First, host-based protection mechanisms are thought to be difficult to manage because of the potential scale of large deployments. Outside the enterprise environment, home users are unlikely to have the technical skill to monitor and patch a complicated system. We purposefully designed FLIPS to require little management beyond installation and initial training. PayL can perform unsupervised training. One task that should be performed during system installation is the addition

of a firewall rule that redirects traffic aimed at the protected application to the proxy and only allows the proxy to contact the protected application.

Second, the performance of such a system is an important consideration in deployment. We show in Section 5 that the benefit of automatic protection and repair (as well as generation of zero-day signatures) is worth the performance impact of the system. If the cost is deemed too high, the system can still be deployed as a honeypot or a "twin system" that receives a copy of input meant for another host. Third, the proxy should be as simple as possible to promote confidence in its codebase that it is not susceptible to the same exploits as the protected application. We implement our proxy in Java, a type-safe language that is not vulnerable to the same set of binary code injection attacks as a C program. Our current implementation only considers HTTP request lines. Specifically, it does not train or detect on headers or HTTP entity bodies. Therefore, it only protects against binary code injection attacks contained in the request line. However, nothing prevents the scope of training and detecting from being expanded, and other types of attacks can be detected at the host.

4   Implementation

This section deals with the construction of our prototype implementation. The proxy was written in Java and includes PayL (400 lines of code) and a simple HTTP proxy that incorporates the signature matching filter (about 5000 lines of code). The supervision framework is provided by STEM (about 19000 lines of C code). One advantage of writing the proxy in Java is that it provides an implicit level of diversity for the system. The small codebase of PayL and the proxy allows for easy auditing.
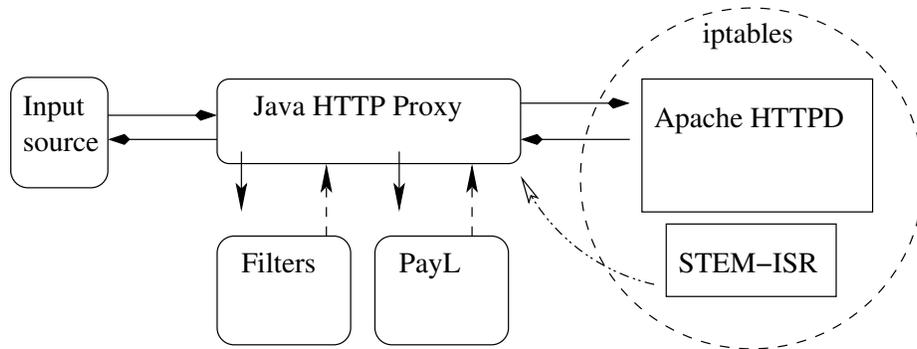

Fig. 2. *FLIPS's Prototype Implementation Components.* We constructed an HTTP proxy to protect HTTP servers (in this example, Apache) from malicious requests. The proxy invokes a chain of three filtering mechanisms and PayL to decide what to do with each HTTP request.

4.1 HTTP Proxy and PayL

The HTTP proxy is a simple HTTP server that spawns a new thread instance for each incoming request. During the service routine, the proxy invokes a chain of Filter objects on the HTTP request. Our default filter implementation maintains three signature-based filters and a Classifier object. PayL implements the Classifier interface to provide an anomaly-based score for each HTTP request. When the proxy starts, it creates an instance of PayL and provides PayL with a sample traffic file to train on.

The core of the filter implementation is split between two subcomponents. The *checkRequest()* method performs the primary filtering and classification work. It maintains four data structures to support filtering. The first is a list of "suspicious" input requests (as determined by PayL). This list is a cache that provides the feedback mechanism a good starting point for matching confirmed malicious input. Note that this list is not used to drop requests. The remaining data collections form a three level filtering scheme that trade off complexity and cost with a more aggressive filtering posture. These lists are not populated by PayL, but rather by the feedback mechanism. The first level of filtering is direct match. This filter is the least expensive, but it is the least likely to block malicious requests that are even slightly metamorphic. The second filter is a reverse lookup filter that stores requests by the score they receive from PayL. Finally, a longest common substring filter provides a fairly expensive but effective means of catching malicious requests.

The second component serves as the feedback mechanism in the proxy. It is a background thread listening for connections from STEM that contains malicious binary code. This thread simply reads in a sequence of bytes and checks if they match previously seen "suspicious" input (as classified by PayL). If not, then the thread widens its scope to include a small cache of all previously seen requests. Matching is done using the longest common substring algorithm. If a match is found, then that request is used in the aforementioned filtering data structures. If not, then a new request is created and inserted into the filters based on the malicious byte sequence.

4.2 STEM

Our supervision framework is an application-level library that provides an emulator capable of switching freely between derandomizing the instruction stream and normal execution of the instruction stream on the underlying hardware. As shown in Figure 3, four special tags are wrapped around the segment of code that will be emulated.

STEM is an *x86* emulator that can be selectively invoked for arbitrary code segments, allowing us to mix emulated and non-emulated execution inside the same process. The emulator lets us (*a*) monitor for derandomization failures when executing the instruction, (*b*) undo any memory changes made by the code function inside which the fault occurred, and (*c*) simulate an error return from said function. One of our key assumptions is that we can create a mapping

```
void foo()
{
    int a = 1;
    emulate_init();
    emulate_begin(stem_args);
    a++;
    emulate_end();
    emulate_term();
    printf("a = %d\n", a);
}
```

Fig. 3. *An example of using* STEM *tags*. The emulate_* calls invoke and terminate execution of *STEM*. The code inside that region is executed by the emulator. In order to illustrate the level of granularity that we can achieve, we show only the increment statement as being executed by the emulator.

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. Due to space limitations, the reader is referred to [29] for details on the general implementation of STEM. In this section, we describe our additions to enable STEM to derandomize an instruction stream and provide feedback to the FLIPS proxy.

4.3 ISR Technique

The main loop of the emulator fetches, decodes, executes, and retires one instruction at a time. Before fetching an instruction, de-randomization takes place. Since the *x86* architecture contains variable-length instructions, translating enough bytes in the instruction stream is vital for the success of decoding. Otherwise, invalid operations may be generated. To simplify the problem, we assume the maximum length (16 bytes) for every instruction. For every iteration of the loop, 16-bit words are XOR'd with a 16-bit key and copied to a buffer. The fetch/decode function reads the buffer and extracts one instruction. The program counter is incremented by the exact length of the processed instruction. In cases where instructions are fifteen bytes or less, unnecessary de-randomization takes place, but this is an unavoidable side-effect of variable-length instructions. If injected code resides anywhere along the execution path, the XOR function will convert it to an illegal opcode or an instruction which will access an invalid memory address. If an exception occurs during emulation, STEM notifies the proxy of the code at the instruction pointer. STEM captures 1KB of code and opens a simple TCP socket to the proxy (the address and port of the feedback mechanism are included in the startup options for *emulate_begin()*). STEM then simulates an error return from the function it was invoked in.

5 Evaluation

Inserting a detection system into the critical path of an application is a controversial proposal because of the anticipated performance impact of the detection algorithms and the correctness of the decision that the detection component reaches. Our primary aim is to show that the combined benefit of automatic protection and exploit signature generation is worth the price of even a fairly unoptimized proxy implementation. Our evaluation has three major aims:

1. show that the system is good at classification
2. show that the system can perform end-to-end (E2E)
3. show that the system has relatively good performance

The first aim is accomplished by calculating the ROC curve for PayL. The second aim is accomplished by an E2E test showing how quickly the system can detect an attack, register the attack bytes with the filters, create the appropriate filter rules, and drop the next instance of the attack. We send a request stream consisting of the same attack at the proxy and measure the time (in both number of 'slipped' attacks and real time) it takes the proxy to filter the next instance of the attack. The third aim is accomplished by measuring the additional time the proxy adds to the overall processing with two different HTTP traces. We were unable to test how well FLIPS blocked real metamorphic attack instances. However, the use of the Longest Common Substring algorithm should provided some measure of protection, as our last experiments showed. We plan to evaluate this capability in future work on the system.

5.1 Hypotheses and Experiments

We investigate four hypotheses to support our aims.

- **Hypothesis 1**: *The use of ISR imposes a manageable performance overhead.* We evaluate this hypothesis with experiments on STEM that explore the impact of partial emulation vs. full emulation on Apache requests.
- **Hypothesis 2**: *The efficacy of PayL is good.* We evaluate this hypothesis by showing the ROC curve for PayL.
- **Hypothesis 3**: *The proxy imposes a manageable performance overhead.* This performance overhead is introduced by a few sources:
  1. the use of an interpreted language (Java) to implement the proxy and the anomaly detector.
  2. the implementation choices of the proxy (*e.g.*, multi-threaded but synchronized at one filter manager). Performance can be improved by adding multiple filter manager objects.
  3. the basic cost of performing proxying, including reading data from the network and parsing it for sanity
  4. the cost of invoking PayL on each request
  5. the cost of training PayL (incurred once at system startup, about 5 seconds for a 5MB file of HTTP requests)

We evaluate this hypothesis by using a simple client to issue requests to the production server and measure the change in processing time when each proxy subcomponent is introduced. Table 2 describes these results.

– **Hypothesis 4**: *The system can run end to end and block a new exploit.* A positive result provides proof for zero-day protection and precise, tuned, automated filtering. To prove this hypothesis, we run the crafted exploit against the full system continuously and see how quickly the proxy blocks it. We determine the latency between STEM aborting the emulated function and the proxy updating the filters.

5.2 Experimental Setup

The experimental setup for *Hypothesis 3* and *Hypothesis 4* included an instance of Apache 2.0.52 as the production server with one simple modification to the basic configuration file: the "KeepAlive" attribute was set to "Off." Then, a simple *awk* script reconstructed HTTP requests from dump of HTTP traffic and passed the request over the *netcat* utility to either the production server or the proxy. The proxy was written in Java, compiled with the Sun JDK 1.5.0 for Linux, and run in the Sun JVM 1.5.0 for Linux. The proxy was executed on a dual Xeon 2.0GHz with 1GB of RAM running Fedora Core 3, kernel 2.6.10-1.770_FC3smp. The production server platform runs Fedora Core 3, kernel 2.6.10-1.770_FC3smp on a dual Xeon 2.8GHz processor with 1GB of RAM. The proxy server and the production server were connected via a Gigabit Ethernet switch. The servers were reset between tests. Each test was run for 10 trials.

5.3 Hypothesis 1: Performance Impact of ISR

We evaluated the performance impact of STEM by instrumenting the Apache web server and performing micro-benchmarks on some shell utilities. We chose the Apache *flood httpd* testing tool to evaluate how quickly both the non-emulated and emulated versions of Apache would respond and process requests. In our experiments, we chose to measure performance by the total number of requests processed, as reflected in Figure 4. The value for total number of requests per second is extrapolated (by *flood*'s reporting tool) from a smaller number of requests sent and processed within a smaller time slice; the value should not be interpreted to mean that our Apache instances actually served some 6000 requests per second.

   We selected some common shell utilities and measured their performance for large workloads running both with and without *STEM*. For example, we issued an '*ls -R*' command on the root of the Apache source code with both *stderr* and *stdout* redirected to */dev/null* in order to reduce the effects of screen I/O. We then used *cat* and *cp* on a large file (also with any screen output redirected to */dev/null*). Table 1 shows the result of these measurements. As expected, there is a large impact on performance when emulating the majority of an application. Our experiments demonstrate that only emulating potentially vulnerable sections of code offers a significant advantage over emulation of the entire system.
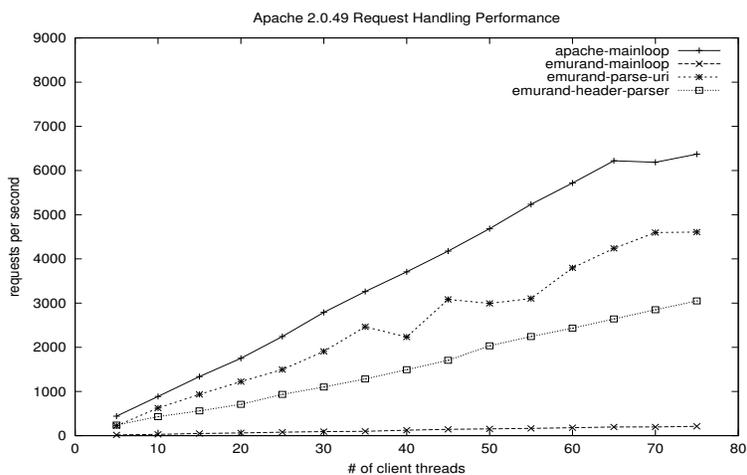
Fig. 4. *Performance of STEM under various levels of emulation.* While full emulation is fairly expensive, selective emulation of input handling routines appears quite sustainable. The "emurand" designation indicates the use of STEM (emulated randomization).

Table 1. Microbenchmark performance times for various command line utilities.

Test Type	trials	mean (s)	Std. Dev.	Min	Max	Instr. Emulated
ls (non-emu)	25	0.12	0.009	0.121	0.167	0
ls (emu)	25	42.32	0.182	42.19	43.012	18,000,000
cp (non-emu)	25	16.63	0.707	15.80	17.61	0
cp (emu)	25	21.45	0.871	20.31	23.42	2,100,000
cat (non-emu)	25	7.56	0.05	7.48	7.65	0
cat (emu)	25	8.75	0.08	8.64	8.99	947,892

5.4 Hypothesis 2: Efficacy of PayL

PayL [38] is a content-based anomaly detector. It builds byte distribution models for the payload part of normal network traffic by creating one model for each payload length. Then it computes the Mahalanobis distance of the test data against the models, and decides that input is anomalous if it has a large Mahalanobis distance compared to the calculated norms.

PayL's results have been presented elsewhere; this section describes how well PayL performed on traffic during our tests. For the purpose of incorporating PayL in FLIPS, we adapted PayL to operate on HTTP requests (it previously evaluated TCP packets). To test the efficacy of PayL's operations on the web requests, we collected 5MB (totaling roughly 109000 requests) of HTTP traffic from one of our test machines. This data collection contains various CodeRed and other malicious request lines. As the baseline, we manually identified the malicious requests in the collection. The ROC curve is presented in Figure 5.
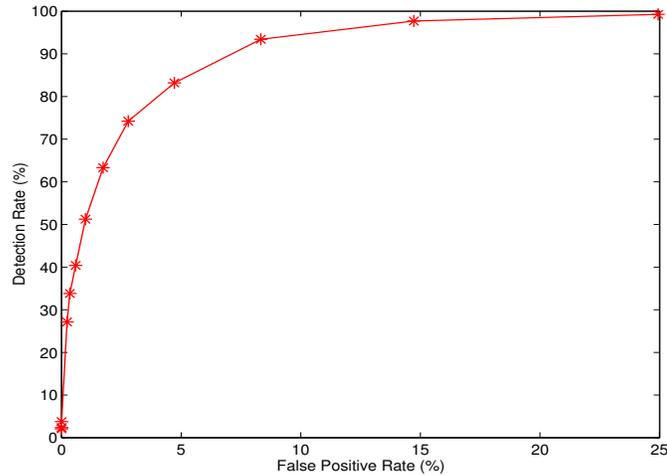

Fig. 5. PayL ROC Curve.

From the plot we can see that the classification result of PayL on the HTTP queries is somewhat mediocre. While all the CodeRed and Nimda queries can be caught successfully, there are still many "looks not anomalous" bad queries that PayL cannot identify. For example, the query "HEAD /cgi-dos/args.cmd HTTP/1.0" is a potentially malicious one for a web server, but has no anomalous content considering its byte distribution. If PayL was used to classify the entire HTTP request, including the entity body, results will be more precise. PayL alone is not enough for protecting a server, and it requires more information to tune its models. We emphasize that FLIPS assumes this requirement as part of its design; we do not filter based on PayL's evidence alone.

5.5 Hypothesis 3: Proxy Performance Impact

We discovered the performance impact of our unoptimized, Java-based proxy on the time it took to service two different traffic traces. Our results are displayed in Table 2 and graphically in Figure 6. Note that our experimental setup is not designed to stress test Apache or the proxy, but rather to elucidate the relative overhead that the proxy and the filters add. Baseline performance is roughly 210 requests per second. Adding the proxy degrades this throughput to roughly 170 requests per second. Finally, adding the filter reduces it to around 160 requests per second.
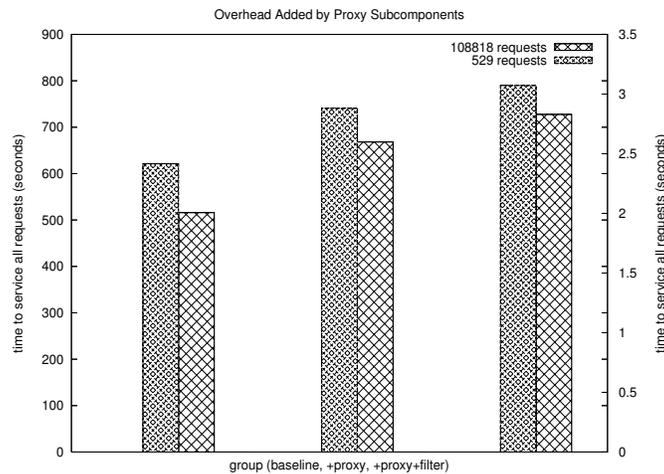

Fig. 6. *Performance Impact of FLIPS Proxy Subcomponents.* A demonstration of how the proxy affects baseline performance for two different traffic traces. Note that the smaller trace (529 requests) is measured on the vertical axis on the right side of the graph. This graph shows the increase in average time to service some number of requests when the proxy is inserted between the client and the HTTP server, and again when the filtering in the proxy is turned on.

5.6 Hypothesis 4: The End-to-End Test

To demonstrate the operation of the system, we inserted a synthetic code injection vulnerability into Apache. The vulnerability was a simple stack-based overflow of a local fixed size buffer. The function was protected with STEM, and we observed how long it took FLIPS to stop the attack and deploy a filter against further instances.

Inserting a vulnerability into Apache proved to be the most challenging part of this experiment. The platform that FLIPS was deployed on (Fedora Core 3)

Table 2. *Performance Impact of FLIPS Proxy Subcomponents.* Baseline performance is compared to adding FLIPS's HTTP proxy and FLIPS's HTTP proxy with filtering and classification turned on. Baseline performance is measured by a client script hitting Apache directly. The addition of the proxy is done by directing the script to contact the FLIPS HTTP proxy rather than the production server directly. Finally, filtering in the FLIPS HTTP proxy is turned on.

Component	# of Requests	Mean Time (s)	Std. Dev.
Baseline	529	2.42	0.007
Baseline	108818	516	65.7
+Proxy	529	2.88	0.119
+Proxy	108818	668	9.68
+Proxy, +Filter	529	3.07	0.128
+Proxy, +Filter	108818	727	21.15

employs address space randomization (via the Exec-Shield) utility. We turned this off by changing the value in */proc/sys/kernel/exec-shield-randomize* to zero. In addition, we marked the *httpd* binary as needing an executable stack via the *execstack* utility.

To test the end-to-end functionality, we directed two streams of attack instances against Apache through our proxy. We first sent a stream of 67 identical attack instances and then followed this with 22 more attacks that included slight variations of the original attack. In the first attack stream, FLIPS successfully blocked 61 of the 67 attack instances. It let the first six instances through before STEM had enough time to feedback to FLIPS. It took roughly one second for FLIPS to start blocking the attacks. After that, each subsequent identical attack instance was blocked by the direct match filter. The second attack stream contained 22 variations of the original. The LCS filter (with a threshold of 60%) successfully blocked twenty of these. This result provides some evidence that FLIPS can stop metamorphic attacks. Our results are summarized in Table 3.

Table 3. *End to end response time of FLIPS filtering.* Once FLIPS has had feedback from STEM, it will block all future identical attack instances. With the LCS filter threshold set at 60%, FLIPS was able to filter 20 of 22 attack variations. Most of the blocked attacks had an LCS of 80% or more. Obviously, attacks that are extremely different will not be caught by the LCS filter, but if they cause STEM to signal FLIPS about them, they will then be blocked on their own merits.

Attack Stream	Total # of Requests	Time to Block	Requests Blocked
Homogeneous Stream	67	1 sec	61
Mixed Stream	22	n/a	20

6 Future Work

There remains a great deal of work in the space of intrusion prevention. We plan on enhancing our implementation of FLIPS along several axes. First, we will extend the proxy to handle different services and clients. Second, we will extend our current treatment of HTTP to include the request headers and entity bodies. Doing so can enable us to verify our experimental results against real Apache vulnerabilities. Third, we plan to augment our set of supervision elements by adding mechanisms like input taint-tracking that may be less expensive than ISR. We also intend to explore using iptables and *libipq* as the basis of input for a more general architecture. Finally, we are currently researching methods of exchanging signatures that have been generated by FLIPS with other FLIPS instances to provide inoculation to members of an Application Community [19].

7 Conclusions

Intrusion detection systems traditionally focus on identifying attempts to breach computer systems and networks. Since detecting intrusions remains a hard problem, reacting in an automated and intelligent way to intrusion alerts has remained largely unaddressed and is often a manual process executed by overburdened system administrators.

We presented FLIPS, an intrusion prevention system that employs a combination of anomaly classification and signature matching to block binary code injection attacks. The feedback for this hybrid detection system is provided by STEM, an *x86* emulator capable of performing instruction set randomization (ISR). STEM can identify injected code, automatically recover from an attack, and forward the attack code to the anomaly and signature classifiers. We have shown how FLIPS can detect, halt, repair, and create a signature for a previously unknown attack. While we demonstrated an implementation of FLIPS that protects an HTTP server, FLIPS's mechanisms are broadly applicable to host-based intrusion prevention.

References

1. 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.
2. K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the* 14th *USENIX Security Symposium. (to appear).*, August 2005.
3. 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)*, October 2003.

4. 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.

5. S. Boyd and A. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security (ACNS)*, pages 292–302, June 2004.

6. G. Candea and A. Fox. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, May 2003.

7. F. Cuppens and A. Miege. Alert Correlation in a Cooperative Intrusion Detection Framework. In *IEEE Security and Privacy*, 2002.

8. B. Demsky and M. C. Rinard. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

9. S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

10. M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the USENIX Security Conference*, 2001.

11. S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a Distributed Firewall. In *Proceedings of the 7th ACM International Conference on Computer and Communications Security (CCS)*, pages 190–199, November 2000.

12. G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.

13. H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Conference*, 2004.

14. S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching Intrusion Alerts Through Multi-host Causality. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2005.

15. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

16. A. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. Technical report, Georgia Tech College of Computing, 2004.

17. C. Krugel, T. Toth, and E. Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2002.

18. M. E. Locasto, J. J. Parekh, A. D. Keromytis, and S. J. Stolfo. Towards Collaborative Security and P2P Intrusion Detection. In *Proceedings of the IEEE Information Assurance Workshop (IAW)*, pages 333–339, June 2005.

19. M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Application Communities: Using Monoculture for Dependability. In *Proceedings of the 1st Workshop on Hot Topics in System Dependability (HotDep-05)*, June 2005.

20. J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *The 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

21. R. E. Overill. How Re(Pro)active Should an IDS Be? In *Proceedings of the 1st International Workshop on Recent Advances in Intrusion Detection (RAID)*, September 1998.

22. T. Pietraszek. Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2004.

23. J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overflows. *IEEE Security & Privacy*, 2(4):20–27, July/August 2004.

24. J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code. In *Proceedings of the Workshop on Rapid Malcode (WORM)*, 2003.

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

26. M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and J. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

27. S. Sidiroglou, J. Ioannidis, A. D. Keromytis, and S. J. Stolfo. An Email Worm Vaccine Architecture. In *Proceedings of the 1st Information Security Practice and Experience Conference (ISPEC)*, April 2005.

28. S. Sidiroglou and A. D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 220–225, June 2003.

29. S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

30. S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

31. A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *The 12th Annual Network and Distributed System Security Symposium*, February 2005.

32. A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

33. R. Sommer and V. Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 262–271, 2003.

34. A. Stig, A. Clark, and G. Mohay. Network-based Buffer Overflow Detection by Exploit Code Analysis. In *AusCERT Conference*, May 2004.

35. S. Stolfo. Worm and Attack Early Warning: Piercing Stealthy Reconnaissance. *IEEE Privacy and Security*, May/June 2004.

36. G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution Via Dynamic Information Flow Tracking. *SIGOPS Operating Systems Review*, 38(5):85–96, 2004.

37. T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

38. K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 203–222, September 2004.