Moving Target Defenses in the Helix Self-Regenerative Architecture

Claire Le Goues, Anh Nguyen-Tuong, Hao Chen, Jack W. Davidson, Stephanie Forrest, Jason D. Hiser, John C. Knight and Matthew Van Gundy

Abstract In this chapter we describe the design, development and application of the Helix Metamorphic Shield (HMS). The HMS: (1) continuously shifts the program's attack surface in both the spatial and temporal dimensions, and (2), reduces the program's attack surface by applying novel evolutionary algorithms to automatically repair vulnerabilities. The symbiotic interplay between shifting and reducing the attack surface results in the automated evolution of new program variants whose quality improves over time.

1 Introduction

Despite years of research warning of the dangers of the software monoculture, most systems today are still deployed in a relatively static configuration. An attack that works on one system is easily and quickly adapted to work on all similarly-configured systems. Even when software vendors regularly release security-critical patches, the window of vulnerability remains unacceptably high. Patches are not released quickly enough to combat zero day-attacks–attacks that take advantage of latent vulnerabilities known to attackers (but not necessarily known to defenders). Further, even when such patches are available, they are often not applied in a timely manner. To remedy this situation, the notion of a moving target defense (MTD) has been put forward as a "game-changing" capability [20]. A moving target defense

Claire Le Goues, Anh Nguyen-Tuong, Jack W. Davidson, Jason D. Hiser, John C. Knight and
University of Virginia, e-mail: {legoues,nguyen,jwd,hiser,knight}@cs.virginia.edu

Hao Chen and Matthew Van Gundy
University of California, Davis, e-mail: {mdvangundy,hchen}@ucdavis.edu

Stephanie Forrest
University of New Mexico, e-mail: forrest@cs.unm.edu

seeks to thwart attacks by invalidating knowledge that an adversary must possess to mount an effective attack against a vulnerable target.

In this chapter, we describe the design, development and application of the Helix metamorphic shield (HMS). The HMS: (1) continuously shifts the program's attack surface in both the spatial and temporal dimensions, and (2) reduces the program's attack surface by using novel evolutionary algorithms to automatically repair vulnerabilities. Continuously shifting the attack surface both increases the effort required for a successful attack and results in "noisier" attacks, as adversaries must repeatedly probe targeted programs to reveal critical information, e.g., a random key. Helix then turns the table on adversaries and uses information contained in these probing attempts as hints for automatically generating, vetting and deploying candidate patches. Furthermore, the presence of the HMS sets up an interesting dynamic between competing adversaries. A single vulnerability can lead to attacks of varying severity depending on the attack payload and the value of the target. An adversary that launches a low-value attack that results in a Helix repair becomes a spoiler for other adversaries eyeing higher-value targets.

The HMS is under active development. In its current form, however, the HMS already displays the following characteristics:

- The HMS has demonstrated the ability to continuously shift the attack surface of programs, thereby presenting adversaries with an ever-changing attack surface.
- The HMS concept applies at various levels of the software stack, ranging from high-level web applications to executable binaries.
- The HMS generates repairs for both security-critical and non-security critical vulnerabilities. We have demonstrated the automated generation and vetting of patches for self-repair to provide protection against a wide-range of attack classes, including infinite loops, segmentation faults, remote heap buffer overflows, non-overflow denials of service, local stack buffer overflows, and format string vulnerabilities, among others.
- The HMS turns the table on adversaries. Helix uses information revealed by adversaries to automatically repair programs.
- The HMS has been demonstrated in a closed-loop system, repairing programs automatically in response to Helix attack sensors.
- The HMS leverages inexpensive cloud computing infrastructures for its analysis engine. We have been able to repair real-world programs using Amazon's EC2 cloud infrastrucure for less than \$8 per bug.

1.1 Helix Architecture

Figure 1 provides a high-level overview of the Helix Metamorphic Shield architecture. Helix takes as input software in source or binary form and performs the necessary transformations to augment the input software with the ability to shift its attack surface on a continuous basis. The first generation of the deployed software retains the original vulnerabilities (shown as two circles in the box labelled

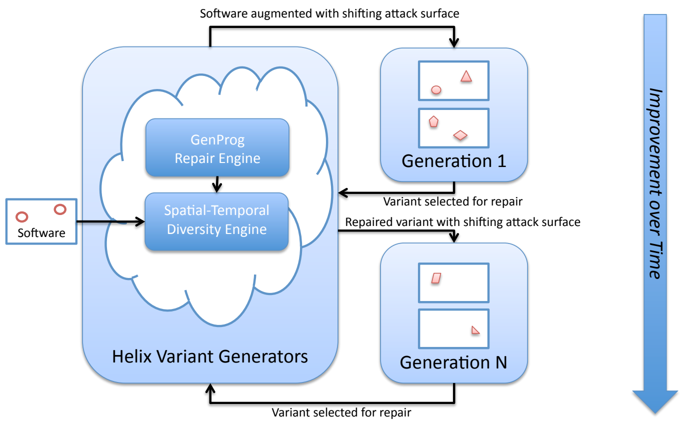
Software augmented with shifting attack surface
GenProg Repair Engine
Spatial-Temporal Diversity Engine
Software
Helix Variant Generators
Generation 1
Variant selected for repair
Repaired variant with shifting attack surface
Generation N
Variant selected for repair
Improvement over Time

Fig. 1 Helix Metamorphic Shield Architecture. Every software generation is augmented with a continuously shifting attack surface using spatial-temporal diversity transformations. Over time, via its GenProg component, Helix automatically reduces the attack surface by automatically generating and vetting candidate repair patches.

Software on the left-hand side in the figure) but is already able to increase the attackers' workload for mounting a successful attack through the use of both spatial and temporal diversity techniques (shifting attack surface shown as vulnerabilities with various shapes). Section 2 describes how the spatio-temporal diversity engine imbues software with a dynamically shifting attack surface, and discusses security and performance results for both binary executables (Section 2.1) and web applications (Section 2.2).

When attack attempts are thwarted and detected, Helix seeks to automatically patch the targeted vulnerabilities via the GenProg Repair Engine (Section 3). GenProg uses evolutionary algorithms to create and vet candidate repair patches. The repair search and validation process is computationally expensive but is inherently parallelizable. In Section 3, we present results for automatically repairing real-world programs using the Amazon EC2 cloud infrastucture. Once a repaired variant (or set of variants) is generated and selected for deployment, Helix again will augment the variant with the ability to continuously shift its attack surface. The net result of this process is the creation of software variants that improve over time as each new generation contains fewer vulnerabilities (Generation N in figure 1).

In the scenario just outlined, the repair process was triggered reactively as a result of detecting potential attacks. Section 4 presents future work opportunities such as triggering repair proactively based on a variety of possible events.

2 Continuously Shifting the Attack Surface

The primary insight underlying our dynamically shifting attack surface is a combination of static diversity techniques with a fast-moving temporal component. A natural, but misleading intuition, would be that the effectiveness of such an approach is proportional to the rate of re-randomization. This intuition only holds true in the case of information leakage vulnerabilities, in which attackers probe a target system to expose or infer knowledge used in further attacks [33, 20]. For brute-force attacks in which attackers exhaustively perform a state-space search, such as the derandoming attack on ASLR, dynamic diversity only increases the attacker's workload by at most a factor of two [46].

Examples of information leakage attacks include side-channel attacks [9, 8], format string vulnerabilities [12], incremental attacks that probe a target system to infer knowledge of the secret key used in diversity defenses [47], or careless errors that simply reveal secret keys (such as printing it in an exception handler). To study the effectiveness of dynamically shifting the attack surface to protect against information leakage attacks, we developed an analytic model that studies the effect of re-randomization as it relates to a given rate of information leakage [33]. The model validates the notion that the rate of re-randomization should be faster than the rate at which an attacker can infer and use information about a target system.

For network-facing servers, the implication is that the attack surface must be shifted at a very high frequency, since attack probes only take a few milliseconds. In the next subsections, we show how the Helix architecture meets this requirement. First, we describe results in designing and prototyping a dynamic version of instruction-set randomization (ISR) that re-randomizes binaries at a rate of every 100 msec. We then show our results for another dynamic variant of ISR at the web-application level, potentially operating at the rate of every network request.

2.1 Dynamic Diversity for Protecting Binary Executables

In designing techniques for protecting binaries via dynamic diversity, we put forth the following set of of requirements:

- The technique should operate on x86 binaries directly.
- The technique should proactively and continuously shift the attack surface.
- The technique should operate efficiently and at a rate that is fast enough to provide protection.
- The software architecture should be flexible and allow for a wide variety of possible diversity (and non-diversity) transformations.

These requirements were motivated by the need for more effective protection against information leakage attacks and the desire for the metamorphic shield to be practical (i.e., efficient and easily deployable), and effective as a generic platform for deploying defenses on arbitrary binaries.

2.1.1 Dynamic Binary Rewriting: Strata Virtual Machine

To fulfill these requirements, we based our architecture on software dynamic translation (SDT) techniques. SDT enables software malleability and adaptivity at the instruction level, by providing facilities for transparent binary run-time monitoring and modification. SDT can affect an executing program by injecting new code, modifying existing code, or arbitrarily controlling program execution.

Examples of software dynamic translation systems include Strata [42, 43, 44], Pin [29], HDTrans [48] and DynamoRIO [25]. The flexibility afforded by software dynamic translations makes them well-suited for implementing a wide rande of security transformations and policies. SDTs have been used in numerous security applications, including:

- augmenting binaries with arbitrary sensors and actuators [11],
- restricting control flow transfers [25],
- thwarting return-oriented programming attacks by monitoring suspicious code sequences [10] or by relocating instructions [16],
- diversity techniques [18, 38, 39, 16, 55, 35, 33],
- fine-grained applications of access-control policies [36],
- regulating resource consumptions [43], and
- protecting against fine-grained memory errors [15].

Many of these techniques have been incorporated and developed within the Helix project [16, 38, 33, 11]. Within the context of the moving-target defense, we apply Strata to dynamically shift the attack surface. We use a dynamic version of instruction-set randomization (ISR) as an exemplar. Strata was designed to be easily reconfigured and retargeted for new applications and computing platforms.

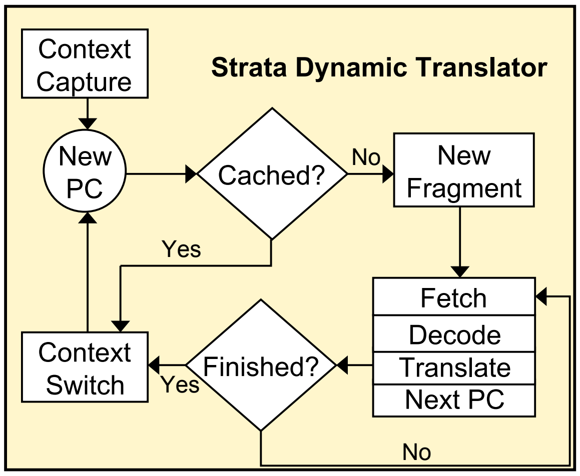

Fig. 2 Strata Application-Level Virtual Machine. Any step in Strata's fetch-decode-translate-execute cycle is easily replaced or augmented with new functionality. Helix leverages Strata's flexibility to implement a wide variety of security transformations directly on executable binaries.

As shown in Figure 2, Strata is organized as a virtual machine that mediates execution of an applications instructions. Strata dynamically loads a binary application and mediates application execution by examining and possibly translating an application's instructions before they execute on the host CPU. Translated application instructions are held in a Strata-managed code cache called the fragment cache. A fragment is the basic unit of translation, similar to a basic block. Once a fragment finishes execution, the Strata VM captures and saves the application context (e.g., PC, condition codes, registers, etc.). Following context capture, Strata processes the next application instruction. If a translation for this instruction has been cached, a context switch restores the application context and begins executing cached translated instructions on the host CPU. Otherwise, the instruction is translated (and, possibly, instrumentated) and the translation is placed into the fragment cache and is executed on the host CPU.

In the next section, we describe the dynamic instruction set randomization algorithm and its implementation over the Strata framework.

2.1.2 Dynamic Instruction Set Randomization

Harold Thimbleby first proposed using randomization to create a unique instruction set as a technique for preventing the spread of viruses [51]. Researchers at the University of New Mexico and Columbia University independently proposed ISR as a method for protecting against code-injection attacks [4, 5, 24]. Both groups originally implemented ISR prototypes for the x86 using emulation (Valgrind at New Mexico [32] and Bochs at Columbia [27]. Subsequent work used software dynamic translation to improve the efficiency of ISR [18, 55, 35].

A simple but effective implementation of ISR is to encode at load time (or earlier) the native binary form of a program using an XOR key [4, 5, 24, 33]. Just prior to execution, the program is decoded using the same XOR key to recover the original instruction stream. Injected code that is not encoded will likely result in the execution of random instructions that will lead to the target program crashing.

We developed a tool to analyze x86 ELF binary programs and to identify the ranges where executable instructions can exist. When Strata starts up, it encrypts code sections using a simple XOR scheme with an n-byte key:

$P' = P \oplus Key$

Strata intercepts the dynamic loading of libraries and performs a similar operation, typically using the same original key K, though different keys may be used.

To recover the original instructions, a decryption module between the fetch and decode modules of the Strata virtual machine applies the following transformation:

$P = Key \oplus P'$

An attack that attempts to inject code will most likely result in a program crash, as the decoding step will transform the injected code into random instructions. Existing XOR-based implementations exploit the likelihood of a program crash, and only shift the attack surface at load-time, randomizing the key on program startup. However, forking servers in which children processes are spawned to handle requests are common, and by default will inherit their parents' key.

To address this concern, instead of randomizing the key only at load-time, our dynamic instruction set randomization technique continuously shifts the attack surface during program execution. Figure 3 illustrates our dynamic ISR implementation using the Strata virtual machine.

Rekeying the application consists of applying the old XOR key, followed by the application of a new random XOR key:

$P' = Key_{new} \oplus Key \oplus P$

$Key = Key_{new}$

Subsequently, the decryption module uses the new random key to recover the original program text.

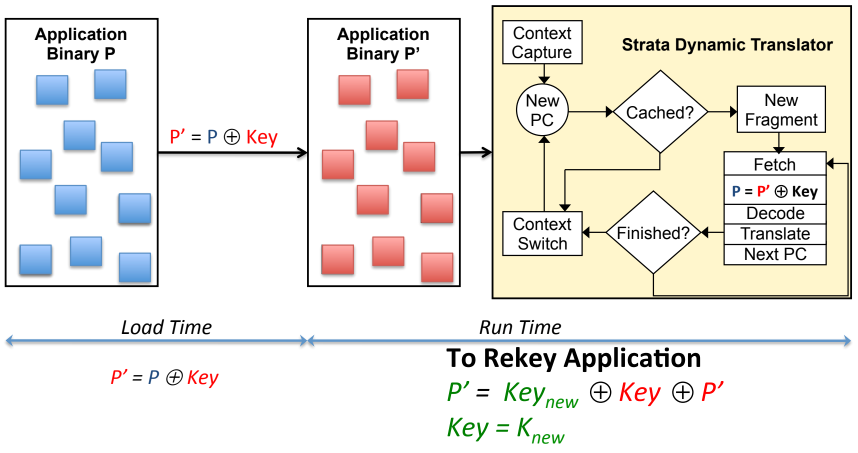

Fig. 3 Dynamic ISR Implementation. At load-time, the program text is encrypted using an XOR key. The Strata VM is augmented with a decryption module that reapplies the key to recover the original program text. Throughout program execution, the program text is re-encrypted under a new key resulting in a dynamically shifting attack surface.

2.1.3 Results

We evaluated the performance of our XOR-based ISR implementation of a Metamorphic Shield using the SPEC2000 benchmark. We present performance results for a re-randomization rate of 100 milliseconds. All performance numbers were averaged over three runs for each of the program in SPEC2000. These numbers were obtained using version 8 of Fedora Core Linux, running in a VMWare image on a dedicated Mac Pro.

Figure 4 shows the performance of executing the benchmarks with and without the metamorphic shield. For a rekeying rate of 100 msec, the performance of the metamorphic shield is essentially the same as that of running the Strata virtual machine. This result is encouraging because it indicates that the metamorphic shield adds virtually no overhead beyond that of Strata itself. Despite measuring the performance on an unoptimized configuration of Strata, the overall average performance overhead is only 14%.

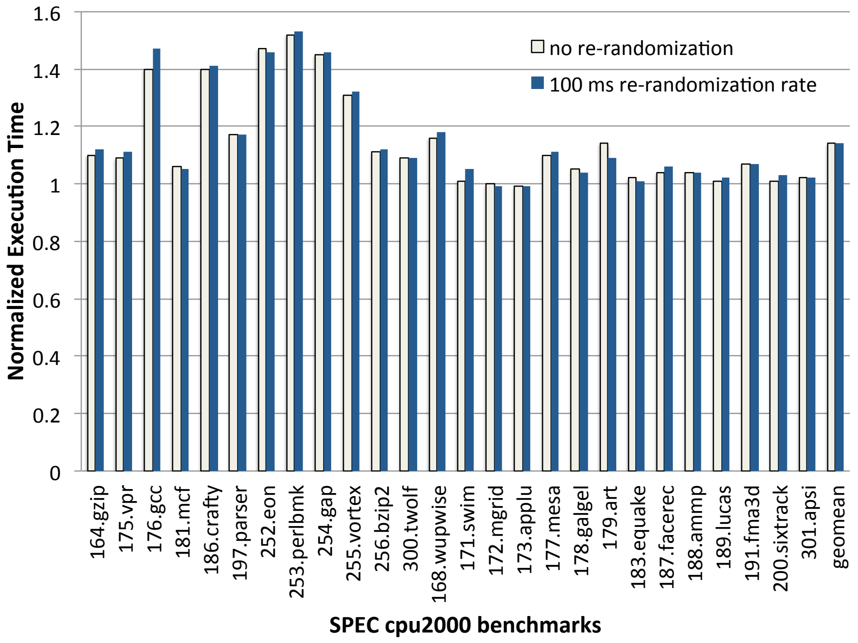

Fig. 4 Performance overhead of dynamic ISR over native execution. Average performance overhead over native execution is 14%. Dynamic ISR at a 100 msec re-randomization rate adds virtually no overhead to the performance of running the Strata virtual machine.

2.1.4 Discussion

Sovarel et. al demonstrated incremental information-leakage attacks on weak XOR-based ISR implementations [47]. In their setup, the XOR key was not re-randomized at load-time, such as in a web server that spawns child processes to service requests. Their attack consisted of two phases. In the first phase, the XOR key is incrementally inferred via the judicious use of attack probes. By repeatedly leaking information, they were able to reconstruct the XOR key in full. Once the key is determined, the attack payload is first XOR'ed using the key. The target program will then proceed to reapply the XOR-key, executing the attacker's payload.

Is a scheme that re-randomizes the key at a 100 msec refresh rate sufficient to thwart incremental attacks? Answering this question requires making real-time assumptions about the probe rate. For example, the average probe time in the attack by Sovarel et al. is approximately 20 msec. A 100 msec rate therefore corresponds to re-randomizing after every fifth probe. However, a motivated adversary could control a botnet, issuing probes in parallel. Our analytical model showed that re-randomization should be performed frequently, as effectiveness depends critically on the re-randomization rate. The difference in the probability of attack success when re-randomizing after every 100th probe or every 4th probe spans six orders of magnitude [33, 13].

Instead of re-randomizing based on a real-time trigger, we are investigating the performance of re-randomizing based on event-driven triggers such as system calls. Since we cannot readily distinguish between normal traffic and attack probing traffic, we need to assume conservatively that every packet read over the network is potentially a probe. In the limit, we would like to re-randomize after every read system call. Furthermore, we are investigating the use of anomaly detection techniques to distinguish between normal traffic and attack probes and thereby reduce the required rate of re-randomization.

The work just described sought to continuously shift the attack surface at the binary level. We next present Noncespaces, a component of the HMS that continuously shifts the attack surface at the level of web applications.

2.2 Dynamic Diversity for Protecting Web Applications

Cross-site scripting (XSS) attacks pose a serious threat to the security of modern web applications. In this section, we present *Noncespaces*, an approach inspired by instruction set randomization [24, 4] that thwarts such attacks by shifting the attack surface of web applications. Noncespaces is an end-to-end mechanism that allows a server to identify untrusted content, reliably convey this information to the client, and allow the client to enforce a security policy on the untrusted content. Noncespaces randomizes (X)HTML tags and attributes to identify and defeat injected malicious web content, extending the concept of randomization up the software stack. Randomization serves two purposes. First, it identifies untrusted content so that the client can use a policy to limit the capabilities of untrusted content. Second, it prevents the untrusted content from distorting the document tree. Since the randomized tags are not guessable, the attacker cannot embed proper delimiters in the untrusted content to split the containing node without causing parsing errors.

2.2.1 Background

A cross-site scripting (XSS) vulnerability allows an attacker to inject malicious content into web pages served by a trusted web application. Because the browser receives the malicious content from a trusted server, the malicious content runs with the same privileges as trusted content, which allows it to run malicious code within the browser, impersonate the user to trusted servers, steal a victim user's private data and authentication credentials, or present forged content to the victim. Such attacks may be reflected or stored; in both scenarios, untrusted user input is returned to a victim user—immediately in the case of a reflected XSS attack or at some later time in the case of a stored XSS attack.

Currently, web browsers protect multiple web applications running within the same browser instance by isolating them according to the Same Origin Policy, which prevents web applications from accessing the private data of other web applications. However, this policy presumes first, that all content from a single web application is equally trustworthy, and second, that all content can be granted access to all data associated with the application. To prevent these vulnerabilities, all the untrusted (user-contributed) content in a web page must be sanitized. However, proper sanitization is very challenging. The context in which untrusted data is interpreted determines the forms of sanitization that are appropriate. There are many ways for an attacker to take advantage of the discrepency between the way sanitization is performed by the server and the way the browser interprets the content [40]. Alternatively, one could let the client sanitize untrusted content. However, without the server's help, the client cannot distinguish between trusted and untrusted content in a web page, since both appear to originate from the trusted server.

We can avoid ambiguity between the client and server by requiring the server to identify untrusted content and requiring the client to ensure that it is displayed safely. However, challenges remain. After the server identifies untrusted content, it needs to tell the client the locations of the untrusted content in the document tree. However, the untrusted content can evade sanitization by distorting the document tree (without executing). To achieve this, the untrusted content can contain node delimiters that split the original node, where untrusted content resides, into multiple nodes. This is known as a *Node-splitting attack* [21]. To defend against this attack without restricting the richness of user provided content, the server must take care to remove only those node delimiters which would introduce new trusted nodes. Noncespaces addresses these concerns by randomizing the XHTML namespace, allowing the client to enforce security policies that limit the capabilities of untrusted content and prevent untrusted content from distorting the document tree.

2.2.2 Approach

The goal of Noncespaces is to allow the client to safely display documents that contain both trusted content generated by a web application and untrusted user-provided content. The browser enforces a configurable security policy to eliminate the client-server semantic gap and to adapt to differing security needs. Such a policy specifies the browser capabilities that each type of content can exercise, thus restricting the capabilities of attacker-injected malicious content.

The client must be able to determine the trustworthiness of all content in a document to faithfully enforce such a server-specified policy. Therefore, the server must first classify content into discrete trust classes. The server then must communicate the content, trust classification, and policy to the client. Finally, the client can enforce the policy. This process is depicted in Figure 5.

As long as the server's content classification is conservative, the server faithfully communicates its classifications to the client, and the client faithfully enforces

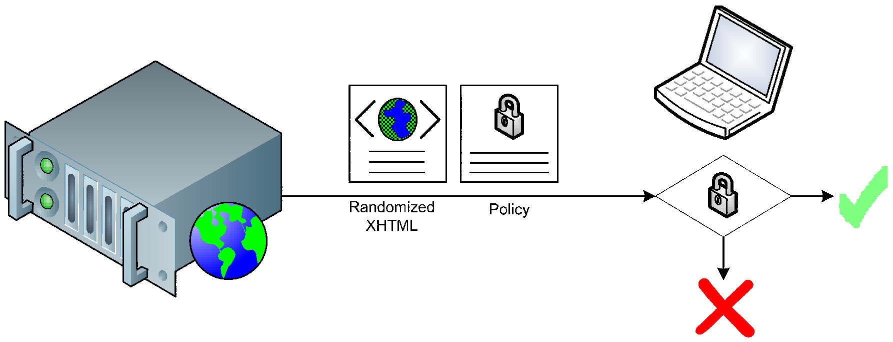

Fig. 5 Noncespaces Overview. The server delivers a XHTML document with randomized namespace prefixes and a policy to the client. The client accepts the document only if it is a well-formed XML document and satisfies the policy.

the server-specified policy, untrusted content will be confined to the capabilities expressly permitted to it by the policy. This ensures that XSS attacks will not succeed.
Communicating Trust Information. The server securely communicates trust information in (X)HTML to the client using randomization. We associate a different randomization function with each content trust class. The names of all elements and attributes in a trust class are remapped according to the associated randomization function so that no injected content can correctly name (X)HTML elements or attributes in other trust classes.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<head>  <title>nile.com : ++Shopping</title>  </head>
<body>  <h1 id="title">{item_name}</h1>
  <p class='review'>{review.text}
     -- <a href='{review.contact}'>{review.author}</a>  </p>
</body>
</html>
```

Fig. 6 Vulnerable web page template used to render dynamic web pages

Consider the vulnerable web template in Figure 6. We can defeat XSS attacks against this document by annotating it. For example, let the randomly chosen string `r60` denote trusted content. For HTML documents, we can prefix trusted tags and attributes with our random identifier, shown in Figure 8. For XHTML documents, we can preserve the original XML semantics of the document while annotating by using our random identifier as an XML namespace prefix, shown in Figure 7.

Attackers cannot inject malicious content and cause it to be interpreted as trusted (as in a node-splitting attack) because they do not know the random prefix. They also cannot escape from the enclosing paragraph element, because they do not know the random prefix, and therefore cannot embed a closing tag with this prefix (in the HTML document, the `<script>` element is the child of an `<r60p>` element, not

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<r60:html xmlns="http://www.w3.org/1999/xhtml" r60:lang="en"
    xmlns:r60="http://www.w3.org/1999/xhtml">
<r60:head><r60:title>nile.com : ++Shopping</r60:title></r60:head>
<r60:body><r60:h1 r60:id="title">Useless Do-dad</r60:h1>
  <r60:p r60:class='review'></p><script>attack()</script><p>
    -- <r60:a href=''></r60:a>  </r60:p>
</r60:body>
</r60:html>
```

Fig. 7 Random prefix applied to trusted content in an XHTML document containing a node-splitting attack injected by a malicious user

```
<!DOCTYPE html>
<r60html r60lang="en">
<r60head>  <r60title>nile.com : ++Shopping</r60title>  </r60head>
<r60body>  <r60h1 r60id="title">Useless Do-dad</r60h1>
  <r60p r60class='review'></p><script>attack()</script><p>
    -- <r60a href=''></r60a>  </r60p>
</r60body>
</r60html>
```

Fig. 8 Random prefix applied to trusted content in an HTML document containing a node-splitting attack injected by a malicious user

a `<p>` element). In the XHTML document, a closing tag that tries to close an open tag with unmatching prefixes will lead to an XML parse error.

To prevent attackers from guessing these (namespace) prefixes, we choose them uniformly at random each time a response is rendered, hence the term Noncespaces. Given a prefix space of appropriate size, knowing the random prefixes in one instance of the document does not help attackers predict prefixes in future instances of the document.

However, naïvely prohibiting all untrusted content will not work because most modern web applications are designed to accept some amount of rich content from users. Though we can use randomization to ensure integrity of trust class information, in practice, we still need a policy that places appropriate constraints on such user-provided content. Therefore, we also provide a mechanism for the server to specify a policy for the client to enforce when rendering the document. This mechanism is provided by new HTTP protocol headers, in which such a policy is specified, and a policy language used to describe the constraints.

Policy Specification. A Noncespaces policy specifies the browser capabilities that can be invoked by content in a given trust class. Figure 9 shows an example policy for XHTML documents. We designed the policy language to be similar to a firewall configuration language. Comments begin with an # character and extend to the end of the line. A minimal policy consists of a sequence of `allow`/`deny` rules. Each rule applies a policy decision—allow or deny— to a set of document nodes

```
 1  # Restrict untrusted content to safe subset of XHTML
 2  namespace  x   http://www.w3.org/1999/xhtml
 3  # Declare trust classes
 4  trustclass trusted
 5  trustclass untrusted
 6  order untrusted < trusted
 7
 8  #Policy for trusted content
 9  allow //x:*[ns:trust-class(., "=trusted")]  # all trusted elements
10  allow //@x:*[ns:trust-class(., "=trusted")] # all trusted attributes
11
12  # Allow safe untrusted elements
13  allow //x:b  |  //x:i    |  //x:u  |  //x:s  |  //x:pre  |  //x:q
14  allow //x:a  |  //x:img  |  //x:blockquote
15
16  # Allow HTTP protocol in the <a href> and <img src> attributes
17  allow //x:a/@href[starts-with(., "http:")]
18  allow //x:img/@src[starts-with(., "http:")]
19
20  # Deny all remaining elements and attributes
21  deny //*  |  //@*
```

Fig. 9 Noncespaces policy restricting untrusted content to BBCode [7]

matched by an XPath expression; XPath is well-suited for this domain because it was designed to query content from hierarchical documents.

Noncespaces additionally provides basic XPath functions for string normalization and additional boolean functions for matching based on trust class or whether an attribute value has changed from the language default. The example policy in Figure 9 specifies two trust classes, trusted and untrusted. There are no restrictions on which tags and attributes can appear in trusted content. Only tags and attributes that correspond to BBCode are allowed in untrusted content: stylistic markup, links to other HTTP resources, and images. Note that lines 17–18 only permits link and image tags to specify URLs for the (non-script) HTTP protocol.

When checking that a document conforms to a policy, the client considers each rule in order and matches the XPath expression against the nodes in the document's Document Object Model. When an allow rule matches a node, the client permits the node and will not consider the node when evaluating subsequent rules. When a deny rule matches a node, the client determines that the document violates the policy and will not render the document. To provide a fail-safe default, if any nodes remain unmatched after evaluating all rules, we consider those nodes to be policy violations (i.e. all policies end with an implicit `deny //*|//@*`). In the event that a policy author wishes to override the default behavior in order to specify a blacklist policy, he can specify `allow //*|//@*` as the last rule to allow all as of yet unmatched nodes.

Client Enforcement. Web browsers must ensure that a Noncespaces-encoded response conforms to the policy before rendering it. The overhead involved in policy retrieval should be minimal, given that most web pages are assembled from multiple requests. Client-side enforcement of the policy is necessary because it avoids possible semantic differences between the policy checker and the browser, which might lead the browser to interpret a document in a way that violates the policy even though the policy checker has verified the document.

2.2.3 Results

We evaluated Noncespaces to ensure that it is able to prevent a wide variety of XSS attacks. We tested Noncespaces against six XSS exploits targeting two vulnerable applications. The exploits were crafted to exhibit the various forms that an XSS attack may take [53]. The applications used in this evaluation were a version of TikiWiki [52] with a number of XSS vulnerabilities and Trustify, a custom web application that we developed to cover all the major XSS vectors.

We began by developing policies for each application. Because TikiWiki was developed before Noncespaces existed, it illustrates the applicability of Noncespaces to existing applications. We implemented a straightforward 37-rule, static-dynamic policy that allows unconstrained static content but restricts the capabilities of dynamic content to that of BBCode (similar to Figure 9). We also had to add exceptions for trusted content that TikiWiki generates dynamically by design, such as names and values of form elements, certain JavaScript links implementing collapsible menus, and custom style sheets based on user preferences.

For Trustify, our custom web application, we implemented a policy that does not take advantage of the static-dynamic model. Instead, the policy takes advantage of Noncespaces's ability to thwart node splitting attacks to implement an ancestry-based sandbox policy similar to the noexecute policy described in BEEP [21]. This policy denies common script-invoking tags and attributes from any namespace (e.g., `<script>` and `onclick`) that are descendants of a `<div>` tag with the `class="sandbox"` attribute. (Note: the policy does not attempt to be exhaustive. It does not enumerate non-standard browser-specific tags and attributes.) To allow the rules to apply to elements and attributes in any namespace we use the common XPath idiom of matching by each node's `local-name()`. An exerpt of the 28 line policy is given in Figure 10.

We first verified that each exploit succeeded without Noncespaces. We then enabled Noncespaces and verified that all exploits were blocked as policy violations.

2.2.4 Performance evaluation

Our performance evaluation seeks to measure the overhead of Noncespaces in terms of response latency and server throughput. Our test infrastructure consisted of the TikiWiki application that we used for our security evaluation running in a VMware virtual machine with 512 MB RAM running Fedora Core 3, Apache 2.0.52, and mod_php 5.2.6. The virtual machine ran on an Intel Pentium 4 3.2 GHz machine with 1 GB RAM running Ubuntu 7.10. Our client machine was an Intel Pentium 4 2 GHz machine with 256 MB RAM running Ubuntu 8.10 Server. These results rep-

```
trustclass unclassified

# Blacklist (possibly incomplete)
deny //*[local-name() = 'div' and @*[local-name() = 'class' \
          and . = 'sandbox']]\
     //*[local-name() = 'script']
deny //*[local-name() = 'div' and @*[local-name() = 'class' \
          and . = 'sandbox']]\
     //@*[local-name() = 'onload' \
          or local-name() = 'onunload'\
          or local-name() = 'onclick' \
          or local-name() = 'onmousedown' \
          or local-name() = 'onmouseover' \
          or local-name() = 'onfocus' \
          or local-name() = 'onsubmit' \
          or (local-name() = 'src' \
            and starts-with(ns:tolower(normalize-space(.)), \
                              "javascript:"))]
# Allow everything else
allow //*
allow //@*
allow //namespace::*
```

Fig. 10 Excerpt from an ancestry-based sandbox policy that denies all potential script-invoking tags and attributes that are descendants of a `<div>` node with the `class="sandbox"` attribute.

resent an upper bound on performance penalty as we have spent no effort optimizing our Noncespaces prototype. In each test we used `ab` [1] to retrieve an application page 1000 times. We varied the number of concurrent requests between 1, 5, 10, and 15, and the configuration of the client and server between the following:

- Baseline: measures original web application performance before applying Noncespaces.
- Randomization Only: measures impact of Noncespaces randomization on server without policy validation on client-side.
- Full Enforcement: measures the end-to-end impact of Noncespaces.

We ran three trials with each test configuration against the TikiWiki application. The response latency shows that enabling Noncespaces randomization on the server increased response time by at most 14%. Enabling the policy checking proxy resulted in response times that were at most 32% higher than the baseline response time. Though the overhead may appear significant at first glance, during interactive use latency typically increased by no more than 0.6 seconds.

We also examine the effect of Noncespaces on server throughput. With randomization enabled throughput is reduced by about 10%. After enabling policy checking, the throughput decreases by an additional 3% for higher numbers of concurrent requests. Because policy checking is performed on the client side, multiple simultaneous client requests has a minimal effect on server throughput.

2.3 Summary: Continuously Shifting the Attack Surface

We have demonstrated the feasibility of quickly shifting a program's attack surface using dynamic variations of instruction-set randomization for both binaries and web applications. Based on this experience, we believe that it is feasible to continuously shift the attack surface at all levels of the software stack, and using a variety of possible diversity techniques. Preliminary performance results show that shifting the attack surface continuously can be performed at reasonable cost.

Regardless of how quickly the attack surface is shifted, diversity techniques do not fundamentally address the underlying vulnerability that enables the attack. For example, unsuccessful attempts to inject binary code against an ISR-protected server will typically result in a program crash. Similarly, attempts to subvert other diversity techniques such as address-space layout randomization, will also result in a program crash, allowing adversaries to launch denial-of-service attacks.

Shifting the attack surface dynamically is an arms race. Shifting must be done quickly, to stave off an adversary's efforts to both learn critical information and to launch an attack on a vulnerable program based on this knowledge. The Helix metamorphic shield (HMS) seeks to exploit this arms race in order to end it. The primary reason that attacks succeed is that adversaries have successfully discovered vulnerabilities that remain unknown to the original developers (otherwise they would, or should, have fixed them before releasing their programs). Attacks attempted against a Helix-protected program that result in a detectable event such as a program crash reveals crucial information about latent vulnerabilities. Helix turns the table on adversaries and uses such information to trigger an automated repair process. The next section discusses GenProg, a new technique based on evolutionary methods to reduce the attack surface via automated program repair.

3 Reducing the Attack Surface: Genetic Programming for Automatic Program Repair

Mature software projects are forced to ship with both known and unknown bugs [28], because the number of outstanding software defects typically exceeds the resources available to address them [3]. Software maintenance, of which bug repair is a major component [37], is time-consuming and expensive, accounting for as much as 90% of the cost of a software project [45] at a total cost of up to $70 billion per year in the US [23, 49]. Even security-critical bugs take an average of 28 days for developers to address [50], further imbalancing the arena in favor of the attacker.

In this section, we describe GenProg, the repair component of the HMS. GenProg is an evolutionary computation technique that efficiently and automatically repairs bugs in off-the-shelf legacy programs. Efficient repair of existing vulnerabilities allows the HMS to *reduce* the program attack surface over time. We show that GenProg can repair a large number and variety of bugs in real-world, off-the-shelf C programs. We then demonstrate its promise in a closed-loop system for automated program repair

3.1 Approach

GenProg is an evolutionary algorithm that repairs existing programs by selectively searching through the space of related program variants until it discovers one that avoids a known defect but retains key functionality. GenProg takes as input a program and a test suite. The program currently passes the *positive test cases*, which encode required functionality. The *negative test cases* characterize the fault under repair; the input program fails these test cases. The goal of the search is to find a variant of the input program that passes the negative test cases while continuing to pass all of the positive test cases.

Figure 11 shows a high-level view of GenProg architecture. GenProg conducts the search using *Genetic programming* (GP), a computational method inspired by biological evolution which evolves computer programs tailored to a particular task [26]. The GP maintains a population of program variants, or *individuals*, each of which is a candidate solution to the problem at hand. In our case, each individual corresponds to a program that varies slightly from the original program. Each individual's suitability is evaluated using a task-specific *fitness function*, and the individuals with highest fitnesses are *selected* for continued evolution. Computational analogs of biological mutation and crossover produce variations of the high-fitness programs, and the process iterates. The search terminates either when it finds a candidate solution that passes all its positive and negative test cases, or when it exceeds a preset number of iterations.

A significant impediment for an evolutionary algorithm like GP is the potentially infinite-size search space of potential programs. To address this, we use novel GP representations, and make assumptions about the probable nature and location of the necessary repair, improving search efficiency. The rest of this subsection provides further algorithmic details.

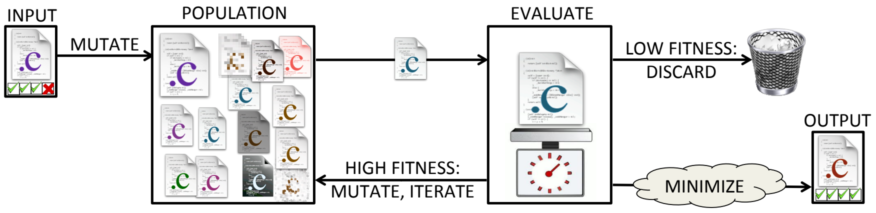

Fig. 11 GP architecture diagram.

3.1.1 Representation

We have primarily applied GenProg at the source level of C programs, though it is also possible to repair programs at the ASM or ELF levels [41]. Programs can be represented at multiple levels of abstraction or granularity. For example, C programs contain both *statements*, such as the conditional statement "`if (!p) { x=0; }`" and *expressions*, such as "`0`" or "`(!p)`". For scalability, we treat the statement as the basic unit, or gene. Thus, we never modify "`(!p)`" because doing so would involve changing an expression. Instead, we might delete the entire "`if` ..." statement, including its then- and else-branches. Additionally, individual variants do not need to store the entire program. Instead, at the source level, each variant is a *patch*, represented as sequence of edit operations. This increases scalability by avoiding the storage of redundant copies of untouched nodes.

3.1.2 Genetic Operators

The *mutation* and *crossover* operators produce new program variants by modifying individuals and recombining them, respectively. Because the basic unit of our representation is the statement, mutation is more complicated than the simple bit flip used in other evolutionary algorithms. A statement selected for mutation is randomly subjected to either *deletion* (the entire statement and all its sub-statements are deleted), *insertion* (another statement is inserted after it), or *swap* (two statements are replaced with one another). Crossover exchanges randomly-chosen subtrees between two individuals, which allows the GP to combine partial solutions.

3.1.3 Localization

We assume that software defects are local and that fixing one does not require changing the entire program. We therefore narrow the search space by biasing modifications towards statement nodes that are more closely associated with the fault [22]. Statements that are only executed by the negative test cases are weighted much more highly than those executed by both the negative and the positive test cases; statements that are not visited by the negative test case at all are not considered for mutation. In this respect, GenProg exploits information provided by an attacker.

We use the term *fix localization* to refer to the *source* of inserted or swapped code. We restrict inserted code to that which includes variables that are in-scope at the destination (so the result compiles; this presented a non-trivial concern in previous work [54]) and that are visited by at least one test case.

3.1.4 Fitness Function

The *fitness* function guides the search. The fitness of an individual in a program repair task should assess how well the program avoids the bug while still doing "everything else it is supposed to do." We use test cases, such as those that often ship with existing software, to measure fitness. Such testing accounts for as much as 45% of total software lifecycle costs [34], and finding test cases to cover all parts of the program and all required behavior is a difficult but well-studied problem in the field of software engineering.

The fitness function applies the edits associated with a variant to the input program and compiles the result into an executable. This executable is run against the set of positive and negative test cases, returning the weighted sum of the test cases passed. Programs that do not compile have fitness zero.

3.1.5 Minimizing the repair

The first variant that passes all positive and negative test cases is called the *primary repair*. However, GP may introduce irrelevant changes on the way to a repaired variant [14]. The minimization step uses tree-structured differencing [2] to express the primary repair as a set of changes to the original program. It then uses *delta debugging* [56] to efficiently compute a subset of these changes such that the changed program passes all test cases, but dropping any additional elements causes the program to fail at least one test case. Delta debugging is conceptually similar to binary search, but it returns a set instead of a single number. We call this smaller set of changes the *final repair*; in our experiments, the final repair is typically at least an order-of-magnitude smaller than the primary repair.

3.2 Efficacy

In the context of an MTD system, GenProg reduces the attack surface by *improving* software in response to detected defects. This section presents results supporting our claim that GenProg provides a scalable and general approach to automatically repairing detected bugs in programs.

3.2.1 Scalability

Success metrics. Bug repair has become such a pressing problem that many companies have begun offering *bug bounties* to outside developers, paying for candidate repairs. Well-known companies such as Mozilla[1] and Google[2] offer significant rewards for security fixes, with bounties raising to thousands of dollars in “bidding wars.”[3] Although security bugs command the highest prices, more wide-ranging bounties are available for bugs ranging from cosmetic concerns to security vulnerabilities (e.g., those provided by Tarsnap.com[4]). These examples suggest that relevant success metrics for automatic bug repair include the fraction of queries that produce code patches, monetary cost, and wall-clock time cost. This section uses these metrics to evaluate GenProg’s scalability .

Benchmarks. Benchmarks appear in the left-hand column of Table 1. We sought to define this set in as unbiased a manner as possible. At a high level, we selected these benchmarks by searching various program repositories to identify acceptable candidate programs (e.g., consisting of at least 50,000 lines of C code, 10 viable test cases, and 300 versions in a revision control system) and reproducible bugs within those programs. We searched systematically through each program’s source history, looking for revisions that caused the program to pass test cases that it failed in a previous revision. Such a scenario corresponds to a human-written repair for the bug corresponding to the failing test case. This approach ensures that benchmark bugs are important enough to merit a human fix and to affect the program’s test suite. The ultimate benchmark set consists of 8 subject C programs covering a variety of uses, comprising 5.1 MLOC and more than 10,000 test cases.

Cloud Computing framework. *Cloud computing*, in which virtualized processing power is purchased cheaply and on-demand, is becoming commonplace and less expensive over time [31]. To evaluate the cost of repairing a bug with GenProg, we used Amazon’s EC2 cloud computing infrastructure for the experiments. Each trial was given a “high-cpu medium (c1.medium) instance” with two cores and 1.7 GB of memory.[5] Simplifying a few details, the virtualization can be purchased as *spot instances* at \$0.074 per hour but with a one hour start time lag, or as *on-demand instances* at \$0.184 per hour.[6]

Experimental Parameters. We ran 10 random GenProg trials per bug. Each trial was terminated after 10 generations, 12 hours, or when another search found a repair, whichever came first. Population size is 40; each individual was mutated exactly once per generation; and 50% of the population is retained (with mutation) on each generation (known as elitism).

Repair results The right side of Table 1 reports results. We report costs in terms of monetary cost and wall clock time from the start of the request to the final result, recalling that the process terminates as soon as one parallel search finds a repair.

[1] `http://www.mozilla.org/security/bug-bounty.html` \$3,000/bug

[2] `http://blog.chromium.org/2010/01/encouraging-more-chromium-security.html` \$500/bug

[3] `http://www.computerworld.com/s/article/9179538/Google_calls_raises_Mozilla_s_bug_bounty_for_Chrome_flaws`

[4] `http://www.tarsnap.com/bugbounty.html`

[5] `http://aws.amazon.com/ec2/instance-types/`

[6] These August–September 2011 prices summarize CPU, storage and I/O charges; `http://aws.amazon.com/ec2/pricing/`

Program	LOC	Description	Tests	Defects Repaired	Patches Per Bug	Cost per Non-Repair Hours	Cost per Non-Repair US$	Cost per Repair Hours	Cost per Repair US$
`fbc`	97,000	legacy programming	773	1 / 3	1.0	8.52	5.56	6.52	4.08
`gmp`	145,000	multiple precision math	146	1 / 2	2.0	9.93	6.61	1.60	0.44
`gzip`	491,000	data compression	12	1 / 5	8.0	5.11	3.04	1.41	0.30
`libtiff`	77,000	image manipulation	78	17 / 24	6.8	7.81	5.04	1.05	0.04
`lighttpd`	62,000	web server	295	5 / 9	4.6	10.79	7.25	1.34	0.25
`php`	1,046,000	web programming	8,471	28 / 44	5.6	13.00	8.80	1.84	0.62
`python`	407,000	general programming	355	1 / 11	5.0	13.00	8.80	1.22	0.16
`wireshark`	2,814,000	network packet analyzer	63	1 / 7	7.0	13.00	8.80	1.23	0.17
total or **avg**	*5,139,000*		*10,193*	*55 / 105*	**5.8**	**11.22h**		**1.60h**	

Table 1 Subject C programs, test suites and historical defects: defects are defined as test case failures fixed by developers in previous versions. 55 of the 105 defects (52%) were repaired successfully and are reported under the "Cost per Repair" columns. The remaining 50 are reported under the "Non-Repair"s columns. "Hours" columns report the wall-clock time between the submission of the repair request and the response, including cloud-computing spot instance delays. "US$" columns reports the total cost of cloud-computing CPU time and I/O. The total cost of generating these results was $403. "Patches per bug" shows the number of unique patches per bug.

Results are reported for cloud computing spot instances, and thus include a one-hour start lag but lower CPU-hour costs.

GenProg repaired 55 of the defects (52%) within the allocated time/generation limits, including at least one defect for each subject program. The successful repairs return a result in 1.6 hours each, on average. The 50 unsuccessful repairs required 11.22 hours each, on average. The total cost for all 105 attempted repairs is $403, or $7.32 per successful run. These costs could be traded off in various ways. For example, an organization that valued speed over monetary cost could use on-demand cloud instances, reducing the average time per repair by 60 minutes to 36 minutes, but increasing the average cost per successful run from $7.32 to $18.30.

Diverse solutions to the same problem may provide several options to developers, or enable consideration of multiple diverse attack surfaces. To investigate GenProg's utility in generating multiple repairs, we additionally allowed all of the bug trials to run to completion (instead of terminating when any trial found a repair). The "Patches per Bug" column in Table 1 shows how many different patches were discovered in this use-case. GenProg produced 318 unique patches for 55 repairs, or an average of 5.8 distinct patches per repaired bug. The unique patches are typically similar, often involving different formulations of guards for inserted blocks or different computations of required values. Such diverse patches can contribute to multiple semantically-equivalent variants, helping reduce the software monoculture.

3.2.2 Generalizability

The previous section showed that GenProg can scale to real bugs in millions of lines of real code. In this section, we substantiate our claim that GenProg provides a *general* means for program repair by evaluating it on a benchmark set designed to cover a variety of bug types.

Programs and Defects. The benchmarks for these experiments are shown in the left-hand side of Table 2. `zune` is a fragment of code that caused all Microsoft Zune media players to freeze on December 31st, 2008. The Unix utilities were taken from Miller *et al.*'s work on *fuzz testing*, in which programs crash when given random inputs [30]. The remaining benchmarks are taken from public vulnerability reports. The defects considered cover eight defect classes: infinite loop, segmentation fault, remote heap buffer overflow to inject code, remote heap buffer overflow to overwrite variables, non-overflow denial of service, local stack buffer overflow, integer overflow, and format string vulnerability.

Test Cases. For each program, we used a single negative test case that elicits the given fault. We selected a small number (e.g., 2–6) of positive test cases per program. In some cases, we used non-crashing fuzz inputs; in others, we manually created simple cases, focusing on testing relevant program functionality; for `openldap`, we used part of its test suite.

Experimental Parameters. We ran 100 random GenProg per each bug. Otherwise, the parameter set is the same here as in the previous subsection.

Repair Results. Table 2 summarizes repair results for the fifteen C programs. The "Time" column reports the average wall-clock time per trial that produced a primary repair. It does not include the minimization time, which is considerably less than the time taken to repair. Repairs are found in 357 seconds on average. The "Success" column gives the fraction of trials that were successful. On average, over 77% of the trials produced a repair, although most of the benchmarks either succeeded very frequently or very rarely. Low success rates can be mitigated by running multiple independent trials in parallel. The "Size" column lists the size of the final (minimized) repair `diff` in lines. The final minimized patch is quite manageable, averaging 5.1 lines. The "Effect" column shows a summary of the effect of the final repair, as judged by manual inspection.

Patch effect. Manual inspection suggests that the majority of produced patches are acceptable, meaningfully changing the program semantics to guard against the error in question while otherwise maintaining functionality. Of the fifteen patches, six insert code (`zune`, `look-u`, `look-s`, `units`, `ccrypt`, and `indent`) seven delete code (`uniq`, `deroff`, `openldap`, `lighttpd`, `flex`, `atris`, and `php`), and two both insert and delete code (`nullhttpd` and `wu-ftpd`).

Patches that delete code do not necessarily degrade functionality: the deleted code may have been included erroneously, or the patch may compensate for the deletion with an insertion. The `uniq`, `deroff`, and `flex` patches delete erroneous code and do not degrade untested functionality. The `openldap` patch removes unnecessary faulty code (handling of multi-byte BER tags, when only 30 tags are used), and thus does not degrade functionality in practice. The `nullhttpd` and `wu-ftpd` patches delete faulty code and replace them with non-faulty code found elsewhere. The effect of the `lighttpd` patch is machine-specific: it may reduce functionality on certain inputs, though in our experiments, it did not.

Program	Lines of Code	Description	Fault	Time (s)	Success (%)	Size	Effect
`zune`	28	example [6]	infinite loop†	42	72	3	I
`uniq utx`	1146	text processing	segmentation fault	34	100	4	D
`look utx`	1169	dictionary lookup	segmentation fault	45	99	11	I
`look svr`	1363	dictionary lookup	infinite loop	55	100	3	I
`units svr`	1504	metric conversion	segmentation fault	109	7	4	I
`deroff utx`	2236	text processing	segmentation fault	131	97	3	D
`nullhttpd`	5575	webserver	remote heap buffer overflow (code)†	578	36	5	B
`openldap`	293k	directory protocol	non-overflow denial of service†	665	100	16	D
`ccrypt`	7515	encryption utility	segmentation fault†	330	100	14	I
`indent`	9906	code processing	infinite loop	546	7	2	I
`lighttpd`	52k	webserver	remote heap buffer overflow (variables)†	394	100	3	D
`flex`	19k	lexical analyzer generator	segmentation fault	230	5	3	D
`atris`	22k	graphical game	local stack buffer exploit†	80	82	3	D
`php`	764k	scripting language	integer overflow†	56	100	10	D
`wu-ftpd`	67k	FTP server	format string vulnerability†	2256	75	5	B
average	1246781			356.5	77.0%	5.7	

Table 2 Experimental results on bugs from programs totaling 1.25M lines of source code. Size of programs given in lines of code (LOC). A † indicates an openly-available exploit. We report averages for 100 random trials. "Time" gives the average time taken for each successful trial and "Success" (how many of the random trials resulted in a repair). "Size" reports the average Unix **`diff`** size between the original source and the final repair, in lines. "Effect" describes the operations performed by an indicative final patch: a patch may insert code (I), delete code (D), or both insert and delete code (B).

In many cases, it is possible to insert code without negatively affecting the functionality. The **`zune`** benchmark contains an infinite loop when calculating dates involving leap years. The repair inserts code to one of three branches that decrements the day in the main body of the loop. The insertion is carefully guarded so as to apply only to relevant inputs, and thus does not negatively impact other functionality. Similar behavior is seen for **`look-s`**, where a buggy binary search over a dictionary never terminates if the input dictionary is not pre-sorted. Our repair inserts a new exit condition to the loop (i.e., a guarded **`break`**). A more complicated example is **`units`**, in which user input is read into a static buffer without bounds checks, a pointer to the result is passed to a **`lookup()`** function, and the result of **`lookup()`** is possibly dereferenced. Our repair inserts code into **`lookup()`** so that it calls an existing initialization function on failure (i.e., before the **`return`**), re-initializing the static buffer and avoiding the segfault. These changes are indicative of repairs involving inserted code.

Functionality degradation. Only one of the patches, for **`php`**, obviously degrades functionality. Disabling functionality to suppress a security violation is often a legitimate response: many systems can be operated in a "safe mode" or "read-only mode." Although acceptable in this situation, disabling functionality could have deleterious consequences in other settings. We explore this patch in more detail and evaluate its impact on functionality in the next section.

3.3 Closed-loop repair

The *automated* repair system evaluated above relies on manual initialization and dispatch of GenProg. However, automated detection techniques in the Helix system can signal the repair process to complete the automation loop. This proposed integration and the corresponding removal of the human from the loop present several areas of additional experimental concern, particularly related to the *quality* of the repairs. Incomplete test suites may lead to fragile or inadequate repairs, further compromising the system. Additionally, a closed-loop system may be subject to *false positives*, where a detector incorrectly signals the existence of a vulnerability; repairs made in response to such false positives may negatively impact the system. GenProg's success in the Helix infrastructure is partially predicated on the practical impact of its generated repairs absent human review.

This section therefore evaluates GenProg in a closed-loop repair system, with several experimental goals: 1) measure the performance impact of repair time and quality on a real, running system, including the effects of a functionality reducing repair on system throughput 2) analyze the functional quality of the generated repairs using fuzz testing and variant bug-inducing input and 3) measure the costs associated with intrusion-detection system false positives.

3.3.1 Closed-Loop System Overview

Our closed-loop prototype is designed with webservers in mind, because they provide compelling case studies for closed-loop automatic repair in the Helix system while allowing for evaluation on real-world programs with realistic workloads. While the webserver is run normally and exposed to untrusted inputs from the outside world, an intrusion-detection system (IDS) checks for anomalous behavior, and the system stores program state and each input while it is being processed. Our prototype system adopts an IDS that detects suspicious HTTP requests based on request features [19]. When the IDS detects an anomaly, the program is suspended, and GenProg is invoked. The negative test case is constructed from the IDS-flagged input. The positive tests consist of standard regression tests.

The efficacy of the proposed system depends on the anomaly detector's misclassification rates (false positives/negatives) and the efficacy of the repair method. The proposed system creates two new areas of particular concern. The first is the effect of an imperfect repair (e.g., one that degrades functionality not guaranteed by the positive tests) to a true vulnerability, which can potentially lead to the loss of legitimate requests or, in the worst case, new vulnerabilities. The second new concern is that a "repair" generated in response to an IDS false alarm could also degrade functionality, again losing legitimate requests. In both cases, the changed attack surface must improve the program, in that it meaningfully modifies it in the face of an attack, without introducing new vulnerabilities for attackers to exploit. The remainder of this section evaluates these concerns.

3.3.2 Benchmarks and workload

We focus these experiments on three benchmarks of our benchmarks from Section 3.2 that consist of three security vulnerabilities in long-running servers: **`lighttpd`**, **`nullhttpd`**, and **`php`**; these can be seen in Table 2. Note that we repair the **`php`** interpreter used by an unchanging **`apache`** webserver deployment, in **`libphp.so`**. The rest of this section outlines the vulnerabilities and repairs in more detail to provide context and to illustrate our claims regarding GenProg's efficacy.

The **`nullhttpd`** webserver is a lightweight multithreaded webserver that handles static content as well as CGI scripts. Version 0.5.0 contains a heap-based buffer overflow vulnerability that allows remote attackers to execute arbitrary code using POST requests. The problem arises because **`nullhttpd`** trusts the **`Content-Length`** value provided by the user in the HTTP header of POST requests; negative values cause **`nullhttpd`** to overflow a buffer. However, there is another location in the code that similarly but correctly processes POST-data. The GenProg-generated repair changes the faulty location such that it calls the other POST-processing code, and thus correctly bounds-checks the vulnerable value. The final, minimized repair is 5 lines long. Although the repair is not the one supplied in the next release by human developers—which inserts local bounds-checking directly—it both eliminates the vulnerability and retains desired functionality.

`lighttpd` is a webserver optimized for high-performance environments; it is used by **`YouTube`** and **`Wikimedia`**, among others. In version 1.4.17, the **`fastcgi`** module, which improves script performance, is vulnerable to a heap buffer overflow that allows remote attackers to overwrite arbitrary CGI variables (and thus control what is executed) on the server machine. The key problem is with the **`fcgi_env_add`** function, which uses **`memcpy`** to add data to a buffer without proper bounds checks. **`fcgi_env_add`** is called many times in a loop. The repair modifies this code such that the loop exits early on very long data allocations. However, the repaired server can still report all CGI and server environment variables and serve both static and dynamic content.

`php` is an interpreter for a popular web-application scripting language. Version 5.2.1 is vulnerable to an integer overflow attack that allows attackers to execute arbitrary code by exploiting the way the interpreter calculates and maintains bounds on string objects in single-character string replacements. Single-character string replacement replaces every instance of a character in a string with a larger string. This functionality is implemented by **`php_char_to_str_ex`**, which handles both single-character and multi-character replacements. The repair disables the single-character case, leaving multi-character replacements untouched (multi-character replacements are not vulnerable to the attack). We use this patch to evaluate the impact of a functionality-degrading patch in the context of the closed-loop system.

Workloads. We use indicative workloads taken from the University of Virginia Computer Science Department webserver to measure program throughput pre-, during, and post-repair. To evaluate repairs to the `nullhttpd` and `lighttpd` webservers, we used a workload of 138,226 HTTP requests spanning 12,743 distinct client IP addresses over a 14-hour period. To evaluate repairs to `php`, we obtained the room and resource reservation system used by the University of Virginia Computer Science Department. It totals 16,417 lines of PHP, including 28 uses of `str_replace` (the subject of the `php` repair). We also obtained 12,375 requests to this system.

We use two metrics to evaluate repair overhead and quality. The first metric is the number of successful requests a program processed before, during, and after a repair. We assume a worst-case scenario in which the same machine is used both for serving requests and repairing the program, and in which all incoming requests are dropped (i.e., not buffered) during the repair process. The second metric evaluates a program on held-out fuzz testing; comparing behavior pre- and post-repair can suggest whether a repair has introduced new errors, and whether the repair generalizes.

				Fuzz Test Failures			
	Repair	Requests Lost	Requests Lost to	Generic		Exploit	
Program	Made?	to Repair Time	Repair Quality	Before	After	Before	After
`nullhttpd`	Yes	$2.4\% \pm 0.83\%$	$0.0\% \pm 0.25\%$	0	0	10	0
`lighttpd`	Yes	$2.0\% \pm 0.37\%$	$0.0\% \pm 1.53\%$	1410	1410	9	0
`php`	Yes	$0.1\% \pm 0.00\%$	$0.0\% \pm 0.02\%$	3	3	5	0
Quasi False Pos. 1	Yes	$7.8\% \pm 0.49\%$	$0.0\% \pm 2.22\%$	0	0	—	
Quasi False Pos. 2	Yes	$3.0\% \pm 0.29\%$	$0.6\% \pm 3.91\%$	0	0	—	
Quasi False Pos. 3	No	$6.9\% \pm 0.09\%$	—	—			

Table 3 Closed-loop repair system evaluation. Each row represents a different repair scenario and is separately normalized so that the pre-repair daily throughput is 100%. The `nullhttpd` and `lighttpd` rows show results for true repairs. The `php` row shows the results for a repair that degrades functionality. The False Pos. rows show the effects of repairing three intrusion detection system false positives on `nullhttpd`. The number after ± indicates one standard deviation. "Lost to Repair Time" indicates the fraction of the daily workload lost while the server was offline generating the repair. "Lost to Repair Quality" indicates the fraction of the daily workload lost after the repair was deployed. "Generic Fuzz Test Failures" counts the number of held-out fuzz inputs failed before and after the repair. "Exploit Failures" measures the held-out fuzz exploit tests failed before and after the repair.

3.3.3 The Cost of Repair Time

The "Requests Lost To Repair Time" column of Table 3 shows the requests dropped during the repair as a fraction of the total number of successful requests served by the original program. The numbers have been normalized to the requests processed by the unmodified programs on a single day, assuming a single attack. Fewer than 8% of daily requests were lost while the system was offline for repairs. Buffering requests, repairing on a separate machine, or using techniques such as signature generation could reduce this overhead.

3.3.4 Cost of Repair Quality

The “Requests Lost to Repair Quality” column of Table 3 quantifies the effect of the generated repairs on program throughput. This row shows the difference in the number of requests that each benchmark could handle before and after the repair, as a percentage of total daily throughput. The repairs for `nullhttpd` and `lighttpd` do not noticeably affect their performance. Recall, however, that the `php` repair degrades functionality by disabling portions of the `str_replace` function. The `php` row of Table 3 shows that this low quality repair does not strongly affect system performance. Given the low-quality repair’s potential for harm, the low “Lost” percentage for `php` is worth examining. Of the reservation application’s 28 uses of `str_replace`, 11 involve replacements of multi-character substrings, such as replacing `'--'` with `'- -'`; the repair does not affect multi-character substring replacements. Many of the other uses of `str_replace` occur on rare paths. For example, many uses replace underscores with spaces in a form label field. If there are no underscores in the field, the result remains correct, since the repair causes single-character `str_replace` to return the input. Finally, a few of the remaining uses were for SQL sanitization; because the application also uses `mysql_real_escape_string`, it remains safe from such attacks.

3.3.5 Repair Generality and Fuzzing

Two additional concerns remain. First, repairs must not introduce new flaws or vulnerabilities, even when such behavior is not tested by the input test cases. To this end, Microsoft requires that security-critical changes be subject to 100,000 fuzz inputs [17] (i.e., randomly generated structured input strings). Similarly, we used the `SPIKE` black-box fuzzer from `immunitysec.com` to generate 100,000 held-out fuzz requests using its built-in handling of the HTTP protocol. The “Generic” column in Table 3 shows the results of supplying these requests to each program. Each program failed no additional tests post-repair. Second, a repair must do more than merely memorize and reject the exact attack input: it must address the underlying vulnerability. To evaluate whether the repairs generalize, we used the fuzzer to generate 10 held-out variants of each exploit input. The “Exploit” column shows the results. For example, `lighttpd` was vulnerable to nine of the variant exploits (plus the original exploit attack), while the repaired version defeated all of them (including the original). In no case did GenProg’s repairs introduce any errors that were detected by 100,000 fuzz tests, and in every case GenProg’s repairs defeated variant attacks based on the same exploit, showing that the repairs were not simply fragile memorizations of the input.

3.3.6 Cost of Intrusion Detection False Positives

Finally, we examine the effect of IDS false positives when used as a signal to GenProg. We randomly selected three of the lowest-scoring normal requests (closest to being incorrectly labeled anomalous) and attempted to “repair” **`nullhttpd`** against them; we call these requests quasi-false positives (QFPs). The “Quasi False Pos.” rows of Table 3 show the effect of time to repair and requests lost to repair when repairing these QFPs.

QFP #1 is a malformed HTTP request. The GenProg repair changed the error response behavior so that the response header confusingly includes **`HTTP/1.0 200 OK`** while the user-visible body retains the correct **`501 Not Implemented`** message, but with the color-coding stripped. The header inclusion is ignored by most clients; the second change affects the user-visible error message. Neither causes the webserver to drop additional legitimate requests, as Table 3 demonstrates.

QFP #2 is a **`HEAD`** request; such requests are rarer than **`GET`** requests and only return header information such as last modification time. They are used by clients to determine if a cached local copy suffices. The repair changes the processing of **`HEAD`** requests so that the **`Cache-Control: no-store`** line is omitted from the response. The **`no-store`** directive instructs the browser to store a response only as long as it is necessary to display it. The repair thus allows clients to cache pages longer than might be desired. It is worth noting that the **`Expires: <date>`** also included in the response header remains unchanged and correctly set to the same value as the **`Date: <date>`** header (also indicating that the page should not be cached), so a conforming browser is unlikely to behave differently. Table 3 indicates request loss.

QFP #3 is a relatively standard request, whichGenProg fails to “repair”. Since no repair is deployed, there is no subsequent loss to repair quality.

These experiments support the claim that GenProg repairs address given errors and without compromising functionality. It appears that the time taken to generate these repairs is reasonable and does not unduly influence real-world program performance. Finally, the danger from anomaly detection false positives is lower than that of low-quality repairs from inadequate test suites, but that both limitations are manageable. We conclude that integration of GenProg in the Helix framework viably modifies and reduces a program’s attack surface in response to detected vulnerabilities, and can thus improve program performance over time.

4 Conclusions and future work

We have described the Helix metamorphic shield, which (1) continuously shifts the program’s attack surface both spatially and temporally and (2), reduces the attack surface by automatically repairing existing vulnerabilities as they are detected. Taken together, these techniques allow software to change quickly enough to thwart a determined attacker and to improve over time by taking advantage of information revealed by such an attacker. Our results show that these approaches are cost-effective, applying to a wide variety of error types and at multiple layers of the software stack. We intend to continue exploring the benefits of randomization, such as by combining automatic exploit and test case generation for existing binaries to construct a closed-loop hardening system for existing binaries, enabling the proactive reduction and shifting of the program's attack surface without the need for attackers to reveal information about vulnerabilities.

5 Acknowledgements

This research is supported by National Science Foundation (NSF) grant CNS-0716446, the Army Research Office (ARO) grant W911-10-0131, the Air Force Research Laboratory (AFRL) contract FA8650-10-C-7025, and DoD AFOSR MURI grant FA9550-07-1-0532. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSF, AFRL, ARO, DoD, or the U.S. Government.

References

1. `http://httpd.apache.org/docs/2.2/programs/ab.html` (2010)
2. Al-Ekram, R., Adma, A., Baysal, O.: diffX: an algorithm to detect changes in multi-version XML documents. In: Conference of the Centre for Advanced Studies on Collaborative research, pp. 1–11. IBM Press (2005)
3. Anvik, J., Hiew, L., Murphy, G.C.: Coping with an open bug repository. In: OOPSLA Workshop on Eclipse Technology eXchange, pp. 35–39 (2005)
4. Barrantes, E.G., Ackley, D.H., Forrest, S., Palmer, T.S., Stefanović, D., Zovi, D.D.: Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In: Conference on Computer and Communications Security, pp. 281–289. ACM (2003)
5. Barrantes, E.G., Ackley, D.H., Forrest, S., Stefanovic, D.: Randomized instruction set emulation. ACM Transactions on Information System Security. **8**(1), 3–40 (2005). DOI http://doi.acm.org/10.1145/1053283.1053286
6. BBC News: Microsoft zune affected by 'bug'. In: `http://news.bbc.co.uk/2/hi/technology/7806683.stm` (2008)
7. `http://www.phpbb.com/community/faq.php?mode=bbcode`
8. Bernstein, D.J.: Cache-timing attacks on AES (2005). URL `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`
9. Brumley, D., Boneh, D.: Remote timing attacks are practical. In: Proceedings of the 12th USENIX Security Symposium, pp. 1–14 (2003)
10. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: DROP: Detecting return-oriented programming malicious code. Information Systems Security pp. 163–177 (2009)
11. Co, M., Coleman, C.L., Davidson, J.W., Ghosh, S., Hiser, J.D., Knight, J.C., Nguyen-Tuong, A.: A lightweight software control system for cyber awareness and security. Resilient Control Systems pp. 19–24 (2009)
12. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G.: Formatguard: Automatic protection from printf format string vulnerabilities. In: USENIX Security Symposium, (2001)

13. Evans, D., Nguyen-Tuong, A., Knight, J.C.: Effectiveness of moving target defenses. In: S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, X.S. Wang (eds.) Moving Target Defense, *Advances in Information Security*, vol. 54, pp. 29–48. Springer (2011)
14. Gustafson, S., Ekart, A., Burke, E., Kendall, G.: Problem difficulty and code growth in genetic programming. Genetic Programming and Evolvable Machines pp. 271–290 (2004)
15. Hiser, J.D., Coleman, C.L., Co, M., Davidson, J.W.: Meds: The memory error detection system. In: Symposium on Engineering Secure Software and Systems, pp. 164–179 (2009)
16. Hiser, J.D., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: ILR: Where'd my gadgets go? In: IEEE Symposium on Security and Privacy. IEEE (2012)
17. Howard, M., Lipner, S.: The Security Development Lifecycle. Microsoft Press (2006)
18. Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J.W., Evans, D., Knight, J.C., Nguyen-Tuong, A., Rowanhill, J.: Secure and practical defense against code-injection attacks using software dynamic translation. In: Virtual Execution Environments, pp. 2–12 (2006)
19. Ingham, K.L., Somayaji, A., Burge, J., Forrest, S.: Learning DFA representations of HTTP for protecting web applications. Computer Networks **51**(5), 1239–1255 (2007)
20. Jajodia, S., Ghosh, A.K., Swarup, V., Wang, C., Wang, X.S. (eds.): Moving Target Defense - Creating Asymmetric Uncertainty for Cyber Threats, *Advances in Information Security*, vol. 54. Springer (2011)
21. Jim, T., Swamy, N., Hicks, M.: Defeating Scripting Attacks with Browser-Enforced Embedded Policies. In: International World Wide Web Conference, pp. 601–610 (2007)
22. Jones, J.A., Harrold, M.J.: Empirical evaluation of the Tarantula automatic fault-localization technique. In: Automated Software Engineering, pp. 273–282 (2005)
23. Jorgensen, M., Shepperd, M.: A systematic review of software development cost estimation studies. IEEE Transactions on Software Engineering **33**(1), 33–53 (2007)
24. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering Code-Injection Attacks With Instruction-Set Randomization. In: Conference on Computer and Communications Security, pp. 272–280 (2003)
25. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: USENIX Security Symposium, pp. 191–206 (2002)
26. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
27. Lawton, K.P.: Bochs: A portable pc emulator for unix/x. Linux J. **1996**(29es), 7 (1996)
28. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Programming language design and implementation, pp. 141–154 (2003)
29. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Programming Language Design and Implementation, pp. 190–200 (2005)
30. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the Association for Computing Machinery **33**(12), 32–44 (1990)
31. Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: USENIX Security Symposium, pp. 67–82 (2009)
32. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Programming Language Design and Implementation, pp. 89–100 (2007)
33. Nguyen-Tuong, A., Wang, A., Hiser, J., Knight, J., Davidson, J.: On the effectiveness of the metamorphic shield. In: European Conference on Software Architecture: Companion Volume, pp. 170–174 (2010)
34. Pigoski, T.M.: Practical Software Maintenance: Best Practices for Managing Your Software Investment. John Wiley & Sons, Inc. (1996)
35. Portokalidis, G., Keromytis, A.D.: Fast and practical instruction-set randomization for commodity systems. In: Annual Computer Security Applications Conference, pp. 41–48 (2010)
36. Rajkumar, R., Wang, A., Hiser, J.D., Nguyen-Tuong, A., Davidson, J.W., Knight, J.C.: Component-oriented monitoring of binaries for security. In: Hawaii International Conference on System Sciences, pp. 1–10 (2011)
37. Ramamoothy, C.V., Tsai, W.T.: Advances in software engineering. IEEE Computer **29**(10), 47–58 (1996)

38. Rodes, B.: Stack layout transformation: Towards diversity for securing binary programs. In: Doctoral Symposium, International Conference of Software Engineering (2012)
39. Rodes, B., Nguyen-Tuong, A., Knight, J., Shepherd, J., Hiser, J.D., Co, M., Davidson, J.W.: Diversification of stack layout in binary programs using dynamic binary translation. Tech. rep. (2012)
40. RSnake: XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html` (2008)
41. Schulte, E., Forrest, S., Weimer, W.: Automatic program repair through the evolution of assembly code. In: Automated Software Engineering, pp. 33–36 (2010)
42. Scott, K., Davidson, J.: Strata: A software dynamic translation infrastructure. In: IEEE Workshop on Binary Translation (2001)
43. Scott, K., Davidson, J.: Safe virtual execution using software dynamic translation. In: Annual Computer Security Applications Conference (2002)
44. Scott, K., Kumar, N., Velusamy, S., Childers, B.R., Davidson, J.W., Soffa, M.L.: Retargetable and reconfigurable software dynamic translation. In: International Symposium on Code Generation and Optimization, pp. 36–47 (2003)
45. Seacord, R.C., Plakosh, D., Lewis, G.A.: Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices. Addison-Wesley Longman Publishing Co., Inc. (2003)
46. Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Computer and Communications Security, pp. 298–307 (2004)
47. Sovarel, N., Evans, D., Paul, N.: Where's the feeb? the effectiveness of instruction set randomization. In: USENIX Security Conference (2005)
48. Sridhar, S., Shapiro, J.S., Bungale, P.P.: Hdtrans: a low-overhead dynamic translator. SIGARCH Comput. Archit. News **35**(1), 135–140 (2007)
49. Sutherland, J.: Business objects in corporate information systems. ACM Comput. Surv. **27**(2), 274–276 (1995)
50. Symantec: Internet security threat report. In: `http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf` (2006)
51. Thimbleby, H.: Can viruses ever be useful? Computers and Security **10**(2), 111–114 (1991)
52. `http://info.tikiwiki.org/tiki-index.php` (2010)
53. Van Gundy, M., Chen, H.: Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In: Distributed System Security Symposium, pp. 55–67 (2009)
54. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: International Conference on Software Engineering, pp. 364–367 (2009)
55. Williams, D., Hu, W., Davidson, J.W., Hiser, J.D., Knight, J.C., Nguyen-Tuong, A.: Security through diversity: Leveraging virtual machine technology. IEEE Security and Privacy **7**(1), 26–33 (2009)
56. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2), 183–200 (2002)