Defeating Denial-of-Service Attacks in a Self-Managing N-Variant System

Jessica Jones*, Jason D. Hiser†, Jack W. Davidson†, and Stephanie Forrest*
*Biodesign Center for Biocomputation, Security and Society
Arizona State University, Tempe, Arizona 85287-7801
Email: {Jessica.Jones.3,Steph}@asu.edu
†Department of Computer Science
University of Virginia, Charlottesville, Virginia 22904
Email: {hiser,jwd}@virginia.edu

***Abstract*—N-variant systems protect software from attack by executing multiple variants of a single program in parallel, checking regularly that they are behaving consistently. The variants are designed to behave identically during normal operation and differently during an attack. When different behavior (divergence) is detected, N-variant systems self-heal by either rolling back to a safe state or restarting. Unfortunately, an attacker can create a denial-of-service (DoS) attack from a diverging input by using it to force an N-variant system into an endless diverge/restart cycle. This paper describes a defense, CRISPR-Inspired Program Resiliency (Crispy), that automatically protects N-variant systems from such DoS attacks. Crispy mitigates DoS attacks against N-variant systems using an automatic signature generation technique modeled on CRISPR/Cas, the bacterial adaptive immune system. Experiments on two webservers using exploits developed by an independent Red Team showed Crispy protected against 87.5% of DoS attacks with zero false positives. Overhead was minimal and varied according to the number of signatures maintained, which can be tailored to the threat model and performance requirements.**

***Index Terms*—adaptive systems, security, software systems**

I. Introduction

An N-variant system consists of multiple versions of a program which execute in parallel. These program variants behave identically on normal (benign) inputs but, with high probability, they behave differently—diverge—during attack [1], [2]. After divergence, the system responds automatically, either by restarting the variants or rolling back to a previously saved state. Thus N-variant systems can both self-protect and self-heal [3], [4].

N-variant systems can provide effective protection for high-availability high-assurance systems such as critical webservers, where the need for uninterrupted service and security justifies the overhead incurred by running multiple variants and monitoring them for consistency. The variant set can be constructed so that divergence is guaranteed for certain classes of attacks, although N-variant system designs rarely include such guarantees. An exception is the Double Helix architecture, which provides a mathematical assurance case for its transforms [5].

N-variant systems protect against and recover from many exploits, especially those related to memory errors, but their design leaves the system vulnerable to Denial-of-Service (DoS) attacks. Once an attacker observes that an input causes divergence, it is straightforward to bombard the system with repeated identical inputs. This tactic forces the system into an endless cycle of divergence followed by restart or rollback, preventing it from resuming meaningful execution. This vulnerability is especially severe for N-variant systems because each recovery from divergence involves significant overhead. Given the prevalence and impact of DoS attacks, this weakness must be addressed before N-variant systems can be deployed as a practical self-managing cybersecurity solution.

This paper describes a technique for protecting N-variant systems from DoS attacks. We leverage divergence as a signal to quickly retrieve attack signatures, which can then be deployed automatically and in near real-time in a running system. This approach ensures minimal downtime when compared to, for instance, waiting for an intrusion-detection system to identify the attack and synthesize a signature. Our system detects divergence with the first system call that is executed anomalously, providing rapid and localized information for generating accurate signatures. Our DoS defense saves inputs that cause divergence on a blacklist and blocks any subsequent inputs that match a signature on the blacklist. Like N-variant systems, this process is inspired by biology: specifically, the CRISPR/Cas adaptive immune system in bacteria, which saves genetic snippets from infections and uses them to neutralize pathogens [6]. Due to the parallels with CRISPR/Cas, we named our DoS protection technique *Crispy*, or CRISPR-Inspired Program Resiliency.

The main contributions of the paper include:

- A new approach to protecting N-variant systems from DoS attacks inspired by bacteria's CRISPR/Cas immune system.
- Demonstration of feasibility in a working system, called Crispy, deployed as part of the DoubleHelix variant generation architecture with the RAVEN N-variant system [5].

The authors gratefully acknowledge the support of DARPA (FA8750-15-C-0118). SF acknowledges the partial support of NSF (1518878), AFRL (FA8750-15-2-0075), DARPA (FA8750-19C-0003) and the Santa Fe Institute. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

- A third-party Red Team evaluation of Crispy's ability to tolerate DoS attacks on a wide variety of vulnerabilities in two webservers. They find that Crispy prevents repeated divergences for 87.5% of the Red-Team designed attacks, with no false positives.
- Evaluation of Crispy's effect on server performance. Crispy slightly increases response time in proportion to the size of the blacklist it maintains. Throughput is not affected.

The remainder of the paper is organized as follows. Section II reviews N-variant systems briefly, focusing on the RAVEN implementation for concreteness. Section III outlines how CRISPR/Cas works in bacteria and describes the implementation of these ideas in Crispy. A Red Team evaluated Crispy against a set of attacks, and Section IV reports results and performance data. Section V describes related work. Limitations, implications, and future work are discussed in Section VI, and Section VII concludes the paper.

II. N-Variant Systems

N-variant systems are one solution to the problem of software monoculture, in which many computers run identical versions of a program [7], [8], leaving them vulnerable to replicated attacks. N-variant systems, as introduced by Cox [1], consist of a single system running multiple unique variants of the same program in parallel. The program variants behave identically from a user's perspective, but they are designed to react differently to an attack. For instance, two variants might use disjoint memory spaces. If an attacker tries to access a particular memory address, the access will be illegal in at least one of the variants. Even if the attacker knows which memory space each variant is using, that knowledge will not allow her to exploit both variants simultaneously.

The DoubleHelix binary transformations are designed to provide guarantees and mathematical evidence based on how the variants are constructed. The approach is to create both "structured" and "probabilistic" variants. Structured variants are constructed so that a property needed to effect an attack is not satisfied by two or more executing variants. For example, arc-injection attacks require the attacker to know the location of the target code and effect a transfer to that code by exploiting a vulnerability. Consider the following invariant where PC denotes the program counter and $Valid_i(PC)$ is true if PC_i is valid executable instruction for $variant_i$:

$$\forall PC, \neg Valid_1(PC) \vee \neg Valid_2(PC) \tag{1}$$

That is, every PC must be invalid in one or more of the variants. If this invariant can be proved for two variants in an N-variant system, arc-injection attacks are not possible. The invariant can trivially be proved if the two variants are constructed with disjoint code address spaces.

Probabilistic variants are constructed by randomizing various features of the code using unique randomizations for each variant. Techniques include: randomizing the stack layout [9], randomizing the heap [10], randomizing data [11], etc. Although these techniques do not support mathematical proofs of non-exploitability, statistical evidence can quantify the difficulty of performing an attack that will behave identically on all running variants. For example, the probability that a critical global variable will have the same location or offset in all variants can be made vanishingly small with sufficient entropy.

DoubleHelix uses binary rewriting techniques to create both structured and probabilistic variants. These diversifed binaries are run on an N-variant system known as RAVEN [5]. Together, DoubleHelix and RAVEN can defeat many attack classes including stack-, heap-, and gadget-based attacks as well as memory-leak attacks [9], [10], [12].

N-variant systems have proven effective against many attack classes. Given an optimal combination of variants, a system running just two variants can defeat return to libc, function pointer overwrite, and stack-smashing attacks [13]. N-variant systems have also thwarted information leakage [14], partial overwrite [15], code injection [1], and code reuse attacks [16]. The attacker must devise a way to exploit vulnerabilities in each variant, and the exploit must compromise all the variants simultaneously or compromise them in a way that does not affect their behavior in order to evade detection.

In addition to the program variants, an N-variant system also contains a monitor that compares the variants' behavior and detects when they diverge. Many N-variant systems, including RAVEN, define divergence as either a crash affecting some, but not all variants, or in terms of system calls. A variant diverges when it invokes a different system call from the other variants, or invokes the same system call with different arguments. Post-divergence, one or more variants may have crashed, and others may be compromised. The monitor terminates any remaining variants to halt the detected attack and initiate the recovery process.

After an attack, RAVEN has two different recovery mechanisms. It restarts the program (possibly with a different set of variants), or it rolls back to a previously saved safe state and resumes execution. In either case, the program resumes normal operation after a brief delay. However, an observant attacker can detect the delay and exploit the divergence by submitting the divergence-inducing input repeatedly, leading to a DoS. This inherent vulnerability limits the deployability of N-variant systems, given the prevalence and cost of DoS attacks. A survey of 254 companies across seven countries found DoS attacks were among the most expensive cyber attacks to resolve, second only to those caused by malicious insiders [17]. In a 2018 international poll, 65% of senior information technology professionals said distributed DoS attacks were currently a very frequent threat, and 69% predicted they would be very frequent in the next three years [18].

DoS attacks can take several forms. Volumetric or flooding DoS attacks are the most common and perhaps best known type [19]. As the name suggests, these attacks degrade service by overwhelming a network with a large volume of requests or traffic. In contrast, DoS attacks at the application layer of the Internet exploit software vulnerabilities to slow or crash an application. For example, two recently discovered bugs in Apache's webserver allow an attacker to cause a DoS by sending specially crafted HTTP requests [20], [21]. Existing DoS detection tools focus on spikes in traffic at the network level, and therefore application-level DoS attacks can elude them. Crispy protects against these DoS attacks, which are increasing in frequency [19], [22].

To resist a DoS, N-variant systems need the ability both to recover from divergence and to acquire immunity to the input that caused it. In the following section, we outline Crispy's implementation and how it varies depending on the N-variant system's recovery strategy. We also show how Crispy immunizes variants against malicious inputs in a way that resembles the CRISPR/Cas immune system.

III. Crispy

Crispy addresses the problem of DoS in N-variant systems using ideas based on the bacterial immune system, which detects viral infections and creates signatures from viral DNA.[1] These single-celled organisms contain a CRISPR/Cas system for adaptive immunity: learning to recognize novel pathogens and disabling them quickly on subsequent encounters. From a computational perspective, CRISPR/Cas systems detect intrusions (viruses), record intrusion signatures in their native DNA, and use those signatures to recognize and neutralize future intrusions.

CRISPR stands for *Clustered Regularly Interspaced Palindromic Repeats*, referring to distinct regions of DNA within bacterial genomes. A CRISPR array within the genome consists of palindromic sequences of bases, such as **GTTATTG**, and spacers. Each distinct spacer sequence is a signature captured from the genome of a virus. The cell saves signatures from infections by splicing them into its own DNA, using the palindromic repeats to separate them. CRISPR arrays persist through cell division, so the learned patterns are passed on to daughter cells. Over time, a single cell can accumulate as many as a few hundred such signatures, although fewer than 50 is thought to be typical [6].

Adjacent genes code for CRISPR-associated (Cas) proteins, which transport information to and from the CRISPR array. Some Cas proteins cut signatures out of viral DNA and paste them into the CRISPR array. Other Cas proteins carry the information encoded in the spacers and compare it to strands of DNA found within the cell. If a DNA strand contains a match to the viral signature from the spacer, the proteins sever the DNA, preventing it from hijacking the cell to replicate itself.

Figure 1 illustrates how CRISPR/Cas records information about infections and uses it to defend the cell. In CRISPR/Cas, the process can be abstracted into three phases: *Adaptation*, when the signature is saved from the virus and spliced into the bacterial (host) DNA; *Processing*, when strands of DNA in the bacterial cell are compared to the known signatures; and *Interference*, when a viral intrusion is detected and its

[1]For simplicity, we refer to CRISPR/Cas as a bacterial immune system. CRISPR/Cas systems have also been found in archaea; they respond to infections by both viruses and plasmids, and some CRISPR/Cas systems attack parasite RNA rather than DNA.

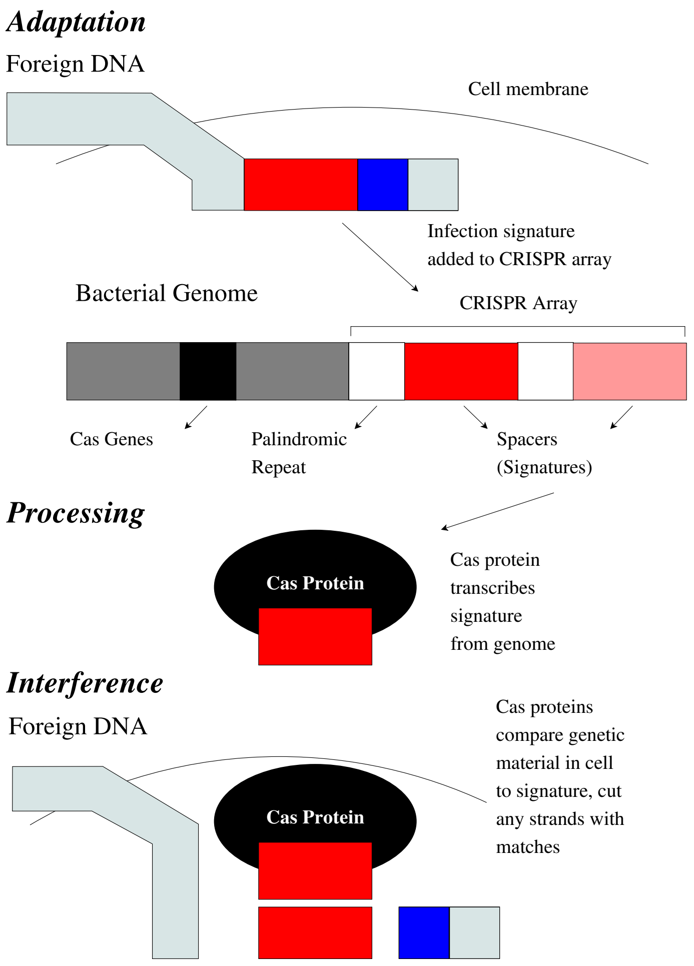


Fig. 1: CRISPR/Cas adaptive immune system. When the cell successfully defeats a novel viral infection, it saves a signature from the viral DNA within its own genome (Adaptation). Cas proteins copy these signatures and compare them to genetic material within the cell (Processing). If the Cas proteins find matching DNA, they sever it, preventing subsequent infections by the same virus (Interference).

DNA is cut. Similarly, our Crispy implementation contains these three phases.

A. Crispy Implementation

Crispy prevents DoS attacks using the three phases found in the CRISPR/Cas system. Table I outlines the correspondences. Crispy's implementation consists of two functions: `crispy_read()` and `retrieve_signature()`. Pseudo-code for these two functions is presented in Algorithm 1 and Algorithm 2. Using library interposition, during execution calls to `read()` are diverted to `crispy_read()`, which implements Crispy's Processing and Interference phases (see Figure 2).

When divergence is detected, RAVEN invokes Crispy's `retrieve_signature()` function as part of its recovery procedure. Function `retrieve_signature()` implements the Adaptation phase (see Figure 2). Rather than inserting the signatures directly into the variant's code, Crispy maintains them in a blacklist. We chose this implementation strategy because new variants are created from the original program, rather than through cell division, and we wanted the blacklists to be shared easily among variants if needed.

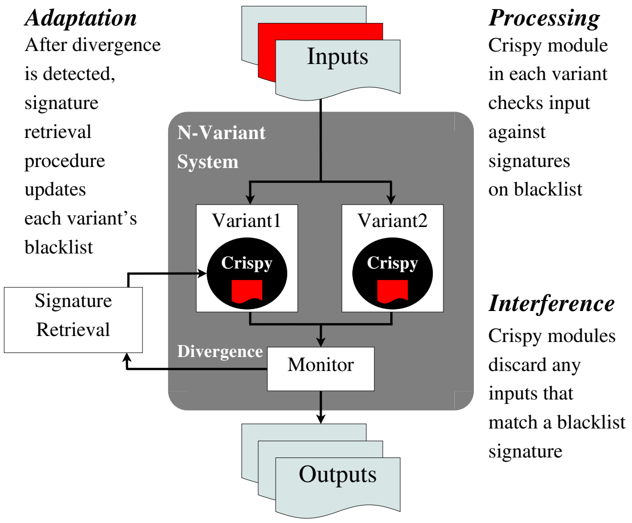
A simple running example illustrates how Crispy self-adapts to prevent DoS attacks. The example assumes RAVEN is configured to take periodic snapshots of the system state using Checkpoint/Restore in Userspace (CRIU) and to rollback to a saved state after divergence [5], [23].

Consider the "codepointer disclosure 1" Apache attack designed by the Red Team (attack number 3 in Table II). In this attack, the attacker logs into a debugging console and inputs the string "pointer\n" to disclose a code location. Apache calls `read()` to copy the input from the attacker's socket (represented by its file descriptor) to a buffer. The call to `read()` is diverted to `crispy_read()`. As shown in Algorithm 1, `crispy_read()` then calls `read()` itself (line 1). The input is compared to the blacklist (lines 2-3). Initially, the blacklist does not contain "pointer\n" and `crispy_read()` simply records the input and returns the number of bytes read (lines 7-8) to the application as if a normal read occurred.

Algorithm 1 crispy_read()

```
Input: $file, buffer, size$
Output: $number\ bytes\ read$
1: $bytes \leftarrow read(file, buffer, size)$
2: $input \leftarrow get\_contents(buffer)$
3: if $input$ in $blacklist$ then
4:     $buffer \leftarrow 0 \times size$
5:     return 0
6: else
7:     record $input$
8:     return $bytes$
9: end if
```

After `crispy_read()` returns, the attacker's input causes Apache to leak information. However, RAVEN detects the information leak because the value leaked is different in each variant as the variants' memory layout has been randomized. When divergence is detected, RAVEN takes a snapshot of the system's state, invokes its recovery routine, and loads a pre-divergence snapshot. It then calls Crispy's `retrieve_signature()` function (Algorithm 2), which locates the most recent input in RAVEN's post-divergence snapshot (lines 1-2). In our example, the input is "pointer\n," which is added to the blacklist (line 3). When RAVEN resumes execution, the variants have the pre-divergence state and the updated blacklist (*Adaptation*) (see Figure 2).



Adaptation
After divergence is detected, signature retrieval procedure updates each variant's blacklist
Inputs
N-Variant System
Variant1
Variant2
Crispy
Crispy
Signature Retrieval
Divergence
Monitor
Outputs
Processing
Crispy module in each variant checks input against signatures on blacklist
Interference
Crispy modules discard any inputs that match a blacklist signature

Fig. 2: Crispy architecture. Each variant has a Crispy module (a shared library containing a blacklist and `crispy_read()` functionality). When the variants diverge, the N-variant system's monitor invokes the signature retrieval procedure, which adds the most recent malicious input to each variant's blacklist (Adaptation). In each variant, `crispy_read()` compares incoming inputs to the blacklist (Processing), discarding inputs that match; `crispy_read()` passes an empty buffer to the variant (Interference).

Algorithm 2 retrieve_signature()

```
Input: $post-divergence\ snapshot, saved\ state$
1: $inputs \leftarrow get\_inputs(post-divergence\ snapshot)$
2: $signature \leftarrow get\_most\_recent(inputs)$
3: $update\_blacklist(saved\ state, signature)$
```

After recovery, if the attacker repeats the attack, `crispy_read()` intercepts the call, finds the input string on the blacklist (Algorithm 1, line 3), and responds by zeroing out the destination buffer, effectively erasing the input (line 4). The function returns zero (zero bytes read), line 5. Apache interprets the return value as an error, closes the attacker's connection, and continues processing requests. Because the variants never receive the divergence-inducing input, the input fails to trigger a second divergence. This resembles CRISPR/Cas's *Interference* phase, in which parasite DNA is disabled.

The simple approach of considering only the most recent input preceding divergence was effective in our evaluation, but could be extended, e.g., if divergence is triggered by a sequence of inputs (see Section VI).

B. Signature generalization

A final component of the CRISPR/Cas system concerns signature generalization, an issue that has been studied previously for computational signature generation systems [24]–[26]. Cas proteins tolerate slight mismatches in the spacer [27]. Equivalent functionality could be added to Crispy by

TABLE I: CRISPR/Cas and Crispy Processes for Intercepting Malicious Inputs

Step	CRISPR/Cas	Crispy
Adaptation	Cas proteins cut and paste a signature from viral DNA into cell DNA	Interposed `read()` function stores inputs to variants. When a malicious input causes the N-variant system to diverge, an input signature is generated and stored.
Processing	Cas proteins compare DNA found within the cell to viral signatures.	Interposed `read()` function checks inputs against signatures from prior divergences.
Interference	If DNA matches a viral signature, Cas proteins sever the DNA.	If an input matches a signature, the interposed function ignores the input.

generalizing the divergence-inducing bytestring. For our high-assurance application, which generates signatures in near real-time, the Red Team evaluation did not highlight signature generalization as a high-priority feature. Because nearly all of the DoS attacks were defeated (Section IV-A) without signature generalization, we opted not to implement it. Crispy's design is thus somewhat conservative.

Our reasoning for Crispy's approach to signature representation is as follows. Crispy is designed to enhance availability [4] by avoiding DoS attacks. Crispy, however, could introduce new availability issues if it accidentally interfered with a legitimate (non-diverging) input. This could happen if a generalized signature introduced a false positive. Put simply, generalizing signatures introduces a new risk that legitimate non-diverging inputs might be blocked. Thus, we designed Crispy to eliminate most but not all DoS attacks. A second consideration is efficiency: many signature generalization schemes add significant computation overhead, and we did not want to impede RAVEN's automated restore/restart process. For other settings, a Crispy-like architecture with signature generalization might be more appropriate.

C. Restart vs. Recovery

For webservers, restart is a good choice because the server is stateless and restarting does not lose critical information. For restart after divergence, `crispy_read()` records inputs in shared memory. After the monitor detects a divergence, the signature generator obtains the most recent input from shared memory, creates a new signature, and adds it to a blacklist file. When the system restarts, Crispy reads the file to initialize the blacklist. Although some particulars may change depending on the N-variant system and its capabilities, the basic Crispy workflow remains the same.

For stateful applications, recovery may be more approriate. When configured for recovery, RAVEN regularly records system state in a CRIU snapshot [23]. After divergence, RAVEN rolls back the variants to match the state at the last valid snapshot. Crispy adds a few steps to this process, as outlined in Section III-A. For some applications, forward error recovery may be possible and desirable. In these cases, a similar mechanism could be used to install signatures to prevent future attacks. We generated the results in Section IV using Crispy on recovery, showing Crispy can protect both stateful and stateless applications.

IV. Results

A. Does Crispy deter DoS attacks?

We tested Crispy's effectiveness using a third-party Red Team and the RAVEN N-variant system. The Red Team customized two webservers, Apache and thttpd, with statically linked modules containing common security weaknesses, as defined by the Common Weakness Enumeration [28]. Then, the Red Team created tests to simulate an attacker exploiting one or more of the seeded weaknesses. These attacks are described in Tables II and III. The attacker requests a persistent connection to the server, then submits inputs that execute code in the added modules. If the variants diverge, RAVEN has detected the attack and stops all executing variants. RAVEN launches the recovery process, and Crispy adds the malicious input's signature to each variant's blacklist. After the variants' states have been updated, RAVEN resumes execution.

They evaluated Crispy's performance on the 24 attacks shown in Tables II and III: 18 for Apache and 6 for thttpd. They first confirmed that each variant set used in the experiment would, without Crispy, always diverge on the attack inputs. They then reran the attacks with Crispy enabled, finding that Crispy prevented repeated divergences on 21 of the 24 attacks or 87.5%. Figure 3 shows the performance of Apache, with and without Crispy protection, under a simulated DoS attack using one of these tests.

For two tests (number 16 in Table II and number 5 in Table III), Crispy failed to avoid the DoS because it does not generalize attack signatures. Crispy generates signatures from these attacks which differ slightly with each divergence: for example, **AC844452aaaaaaaaaaaa** and **2C6F1FFDaaaaaaaaaaaa**. Crispy failed to prevent another DoS attack because the exploit used (number 4 in Table III) does not trigger divergence immediately. The attack sets the stage for a buffer overflow but does not directly cause an overflow, relying on a subsequent arbitrary input to overflow the buffer. In our testing, this occurred several minutes after the initial input. Our current implementation assumes no lag between an attack and the ensuing divergence. A simple n-gram signature representation [29] could address this problem by representing sequences of inputs as potential signatures (see Section VI).

Finally, the Red Team provided functionality tests for the customized Apache modules. These tests assess whether the

TABLE II: Red Team's Apache Attacks

	Attack	Description	Vulnerabilities Targeted by Common Weakness Enumeration Number	Protected?
1	apr pool disclosure	Leaks a pointer to the Apache memory pool	CWE-125: Out-of-bounds Read, CWE-129: Improper Validation of Array Index, CWE-200: Information Exposure, CWE-704: Incorrect Type Conversion or Cast	✓
2	code disclosure	Leaks code address space via a debug print statement	CWE-200: Information Exposure	✓
3	codepointer disclosure 1	Leaks pointers to code via a debug print statement	CWE-200: Information Exposure	✓
4	codepointer disclosure 2	Uses an Easter egg to leak pointers to code	CWE-200: Information Exposure	✓
5	global data disclosure	Reads value from an array using global offsets found through static analysis	CWE-125: Out-of-bounds Read, CWE-129: Improper Validation of Array Index, CWE-200: Information Exposure, CWE-704: Incorrect Type Conversion or Cast	✓
6	global disclosure	Leaks value of global variables through a debug print statement	CWE-200: Information Exposure	✓
7	global offset	Overwrites global variable using global offsets found through static analysis	CWE-123: Write-what-where Condition, CWE-129: Improper Validation of Array Index, CWE-704: Incorrect Type Conversion or Cast	✓
8	global overrun	Uses a buffer overflow to set the value of a block of global memory	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer, CWE-704: Incorrect Type Conversion or Cast, CWE-787: Out-of-bounds Write	✓
9	global partial pointer increment sidechannel	Increments a vtable pointer by 8 so each pointer is to the next function in the vtable	CWE-123: Write-what-where Condition, CWE-129: Improper Validation of Array Index, CWE-704: Incorrect Type Conversion or Cast	✓
10	global scope	Overwrites a global variable by overflowing a global array	CWE-129: Improper Validation of Array Index	✓
11	global scope format string sidechannel	Corrupts a global variable using an externally controlled format string; sidechannel leaks address of global	CWE-134: Use of Externally-Controlled Format String	✓
12	global write-what-where	Exploits write-what-where condition to overwrite "userType" variable, thereby gaining root privileges	CWE-123: Write-what-where Condition	✓
13	heap data sidechannel disclosure	Uses information from sidechannel to disclose heap data values; similar to effect of heartbleed vulnerability	CWE-125: Out-of-bounds Read, CWE-129: Improper Validation of Array Index, CWE-200: Information Exposure, CWE-704: Incorrect Type Conversion or Cast	✓
14	heap pointer disclosure	Leaks pointer to the heap	CWE-125: Out-of-bounds Read, CWE-129: Improper Validation of Array Index, CWE-200: Information Exposure, CWE-704: Incorrect Type Conversion or Cast	✓
15	heap to global partial pointer increment 1	Changes a vtable pointer so that it points to another vtable; uses 4-byte offset write-what-where	CWE-123: Write-what-where Condition, CWE-129: Improper Validation of Array Index, CWE-704: Incorrect Type Conversion or Cast	✓
16	heap to global partial pointer increment 2	Changes a vtable pointer so that it points to another vtable; uses 8-byte offset write-what-where	CWE-123: Write-what-where Condition, CWE-129: Improper Validation of Array Index	No
17	inter-variant communication	Uses differences in the amount of time to execute a loop to leak addresses between variants	CWE-208: Information Exposure Through Timing Discrepancy	✓
18	uninitialized stack	Uses an uninitialized variable to elevate privileges by spraying the stack with non-zero values	CWE-457: Use of Uninitialized Variable	✓

TABLE III: Red Team's thttpd Attacks

	Attack	Description	Vulnerabilities Targeted by Common Weakness Enumeration Number	Protected?
1	code disclosure	Leaks code address space via a debug print statement	CWE-200: Information Exposure	✓
2	codepointer disclosure 1	Leaks pointers to code via a debug print statement	CWE-200: Information Exposure	✓
3	global disclosure	Leaks value of global variables through a debug print statement	CWE-200: Information Exposure	✓
4	global overrun	Uses a buffer overflow to set the value of a block of global memory	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer, CWE-704: Incorrect Type Conversion or Cast, CWE-787: Out-of-bounds Write	No
5	global scope	Uses a buffer overflow to set the value of a block of global memory	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer, CWE-704: Incorrect Type Conversion or Cast, CWE-787: Out-of-bounds Write	No
6	heap flagrant disclosure	Exploits an index overflow to leak a block of heap memory	CWE-129: Improper Validation of Array Index, CWE-200: Information Exposure, CWE-704: Incorrect Type Conversion or Cast	✓

modules return correct output in response to benign inputs. Running these functionality tests with Crispy enabled, using the signatures generated by Crispy for all of the Apache attacks, no false positives were observed.

B. *Efficiency*

Crispy incurs overhead at two points:

- Recovery: After divergence, Crispy must access the recorded inputs and update the blacklist.
- Runtime: Whenever the variants call `read()`, the custom `read()` implementation checks the input against a blacklist and, assuming the input does not match a known malicious signature, records the input before returning.

Figure 3 shows Crispy's performance during a DoS attack. The DoS attack consists of the Red Team's codepointer disclosure 1 attack launched 200 times in succession; it lasts approximately 12 minutes. When the DoS attack begins, performance declines immediately, both with and without Crispy. However, as soon as the Crispy-protected RAVEN N-variant system adds the signature to its blacklist and restores the variants to their checkpointed state, it resumes serving requests (blue line in figure). The recovery process with Crispy lasts 73 seconds. Although recovery from divergence without Crispy is significantly faster (only a few seconds), we note that during a DoS attack an N-variant system will diverge almost immediately after each recovery. Thus, the standard RAVEN N-variant system's performance remains poor until the DoS attack ends (red line in figure). This result motivated our decision to defer more sophisticated signature representations until testing demonstrates that the additional overhead is justified.

Our testing found that Crispy's runtime overhead was affected to some extent by the size of the blacklist. Figure 4 shows the response time, latency, and throughput of Apache variants running in RAVEN with and without Crispy enabled. We tested Crispy with 10 signatures in the blacklist and with 100 signatures. The 10-signature blacklist increased response time by about 7 milliseconds, and the 100-signature blacklist by about 13 milliseconds. We have not optimized the signature lookup algorithm, and a more efficient data structure would reduce this overhead. Implementation choices such as this or selecting the blacklist's size involve trade-offs, and the right choice will depend on the nature of the application to be protected and the threat model. A Crispy-enabled N-variant system with a smaller blacklist will respond more quickly to

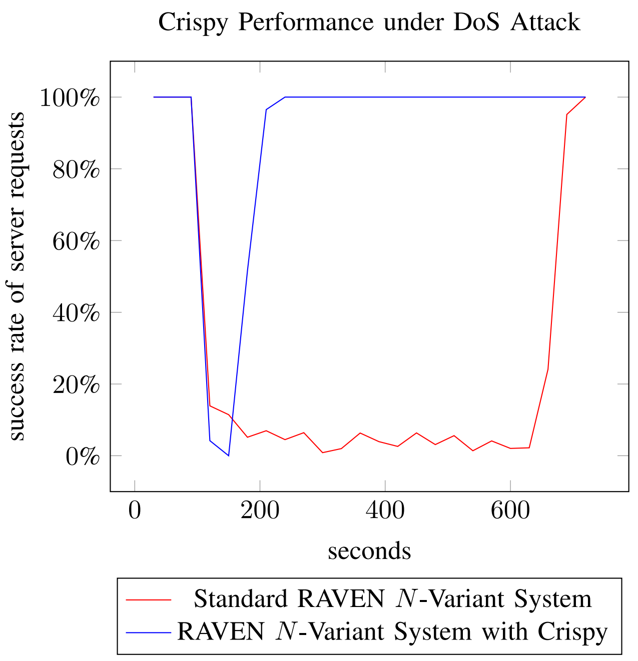


Fig. 3: Performance of the RAVEN N-variant system running Apache with and without Crispy enabled during a DoS attack. The Red Team's codepointer disclosure 1 attack was launched 200 times in succession. After Crispy added the attack signature to the blacklist during recovery, RAVEN resisted the DoS attack. Without Crispy, performance remained low until the DoS attack ended.

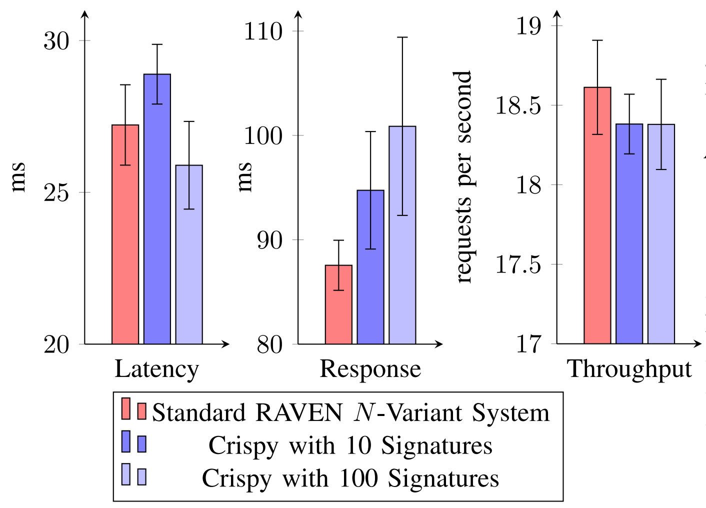


Fig. 4: Performance overhead of RAVEN with and without Crispy enabled while running Apache. Each bar represents 10 tests conducted with the Apache JMeter load testing tool; each test consisted of 898 requests. Error bars indicate one standard deviation from the mean response time, latency and throughput.

benign inputs, but if the server experiences diverse attacks, it will also experience more service interruptions. Mean throughput with Crispy, regardless of blacklist size, was within one standard deviation of the standard RAVEN N-variant system's. Although the time required to serve individual requests slightly increased, Crispy did not significantly affect the number of requests the server could handle.

Adaptive defenses in the biological world require considerable overhead, motivating research into how and why their benefits justify their costs. Adaptive or inducible defenses, like human T cells, activate only in response to pathogens but organisms must devote resources to maintaining them. In contrast, constitutive defenses (such as human skin) provide cheap, constant protection regardless of whether a threat is present. Experiments and mathematical modeling have shown an optimal immune strategy consists of low-cost constitutive defenses for everyday pathogens and more expensive adaptive defenses for rarer and more virulent infections [30], [31]. In software systems, passwords, user groups, and file permissions serve as constitutive defenses, continuously deterring incursions [32]. Adaptive defenses—like CRISPR/Cas in bacteria or Crispy in self-managing software systems—bolster lower-overhead defenses to combat particularly dangerous or persistent threats [33].

V. Related Work

Crispy incorporates concepts from several existing research areas beyond the N-variant systems work discussed earlier. We briefly review the most related work on automated signature generators and Moving Target Defenses (MTD). MTD techniques change a system or program's properties over time or space with the goal of increasing an attacker's workload [34], [35]. MTDs have been applied at different levels of granularity, from individual binaries to systems comprising multiple machines, each of which we review briefly.

A. Automated Signature Generation

At a high level, Crispy's design is closest to Liang and Sekar's seminal work on fast generation of attack signatures [36], although most of the details are different. Both Crispy and their system (known as COVERS) filter inputs using library interposition, dropping any inputs that match known attack signatures. Other inputs are forwarded to the application and logged in memory. If an intrusion is detected, both create a new signature based on the input that caused the attack.

COVERS relies on Address Space Layout Randomization (ASLR) for intrusion detection [36]–[38]. After a memory exception, COVERS's memory analyzer correlates the contents of the affected buffer with recent inputs. It then forwards the relevant inputs to the signature generator, assuming "that an exact copy of some part of the attack input would be found around the affected pointer" [36]. This byte-by-byte matching needs to be efficient, but if the input has been transformed in some way complexities arise. By contrast, Crispy considers only the most recent input, avoiding this correlation step. Crispy also benefits from its use of an N-variant system. Information leak attacks, such as Heartbleed, evade ASLR, but are immediately detected by N-variant systems due to differences in the variants' system calls [14]. More than half of the attacks against which we tested Crispy involved information leaks and would likely not be detected by COVERS using ASLR.

Kaur and Singh provide a good survey of other approaches to signature generation [39]. To summarize, some signature generators create signatures tailored to the underlying vulnerability rather than the input that exploits it. For example, Sweeper replays attacks using instrumentation that enables detailed analysis [40]. Sweeper then generates an "antibody," which includes the malicious input and a set of instruction addresses to monitor. If analysis identifies an instruction that can cause a buffer overflow, Sweeper inserts a check to ensure the input is within bounds. Sweeper can distribute antibodies among different hosts, effectively vaccinating the population. Sweeper incorporates ideas previously presented in [41]–[44]. Vigilante and Bouncer involve the same general steps as Sweeper: attack replay, signature generation, and binary rewriting [45], [46]. By contrast, Crispy was designed to be simpler and more efficient because it targets DoS in N-variant systems. This context provides immediate and local information at the system call level, but it also imposes performance constraints.

B. Binary rewriting

Some low-level MTDs have been widely adopted. StackGuard, for example, detects buffer overflows with a randomly generated canary value [47]. All major operating systems incorporate ASLR, which randomizes a program's layout in memory each time it executes [37], [48]. However, ASLR implementations often offer limited randomization; they commonly randomize the base addresses of memory segments but not relative addresses within the segments. Thus attackers can easily defeat ASLR via information leaks [34]. While ASLR obfuscates a program's layout, instruction set randomization (ISR) obfuscates the program itself. A random key scrambles the program's bytes each time it executes, and a binary-to-binary translator decodes the program during execution [49], [50]. The overhead associated with ISR has prevented its widespread adoption, but hardware support may make these methods more efficient and practical [35].

Newer MTD techniques aim to change the binary while it executes so that the attack surface constantly changes. The DoubleHelix toolchain incorporates some of these techniques via Zipr, a tool for binary analysis, rewriting, and transformation [51]. For example, Zipr can create variants in which the address space layout rerandomizes during execution [52]. Other methods for introducing diversity via binary rewriting include STIR [53] and CCFIR [54].

C. Source code rewriting

Diversifying binaries directly changes properties of a program that attackers use to their advantage, such as positioning of objects within a memory segment. However, MTD and N-variant systems can operate at higher levels of abstraction. Taguinod et al. propose automatically translating the source code of web applications from one language to another (e.g., from Python to PHP) [55]. Injected code intended for the original application would be incompatible with the translated version and would defeat zero-day vulnerabilities in the program's original language. A more aggressive approach is proposed in [56], where semantically distinct versions of a program are created by varying aspects of the program not constrained by the specification.

Other methods for automatically rewriting software target bug repair, e.g., [57]–[59]. Of these, Code Phage [59] draws on concepts from microbiology, just as Crispy does, although the algorithms differ significantly. Code Phage creates patches by transferring code between applications, which the authors argue resembles the process by which bacteria share genetic material.

D. Other moving target approaches

Diversification of a program's environment can increase its security, even if the program itself remains unchanged. Symbiote Embedded Machines (SEM) inject code for intrusion detection into a host program [60]. The SEM code executes with and tracks the state of the host program (just as the monitor does in an N-variant system). Tampering with the SEM causes the host program to crash, and the injected code is randomly distributed throughout the host program so attackers cannot easily locate it. Platform diversity techniques transfer execution between different program instances running on different platforms [61]. For example, Blink reconfigures a user's web browsing platform—changing the operating system, browser, fonts, and plugins—in order to mitigate browser fingerprint tracking [62]. An evaluation of platform diversity defenses highlighted the significance of the threat model [63]. If the time spent running on a given platform exceeds the time required for an attacker to achieve her goal, platform diversity may not enhance security. Some MTDs permute network properties, such as nodes' IP addresses [64], in order to deflect attackers. Finally, Rainbow changes a system's defenses (such as enabling and disabling Captcha) depending on whether observations of the system's behavior suggest malicious users are present [65].

VI. Discussion

Incorporating lightweight automated signature generation in an N-variant system addresses a significant systematic vulnerability in their design, namely susceptibility to DoS attacks. By leveraging divergence as a signal to generate new signatures immediately and locally after a diverging system call, Crispy provides efficient and effective signatures in the context of an executing system. By working in tandem with an N-variant system, Crispy avoids the complications of some prior signature generation methods. For example, Crispy can assume that the N-variant system detects divergence at the first system call involving malicious input, while other signature generators often rely on taint analysis or similar tools to analyze and replay attacks.

Although simple conceptually, Crispy's implementation relies on two key features of the RAVEN and DoubleHelix architecture: (1) Diversified variants that run in parallel and are checked for consistency at each system call; (2) Access to recent inputs after divergence, either from shared memory (if the system is restarted) or from checkpointed state (if the system recovers and continues processing). With these features in place, a Crispy implementation could be deployed on any N-variant system.

Crispy might also be adapted for cyber physical systems (CPS) with a similar architecture. For example, Demirbas and Song describe a wireless sensor network in which monitoring nodes record information about other nodes in the system, then send that information to a detector node [66]. The detector node finds intruders by comparing the information received from different nodes, just as an N-variant system's monitor checks the variants' system calls for consistency. The detector node or the central server for such a system could formulate a signature for the intrusion and broadcast it to the individual nodes. Each node could then filter its inputs accordingly. In addition, CPS components could receive signatures for newly discovered vulnerabilities in their firmware (e.g., [67]).

Some types of input pose special challenges for Crispy. Encrypted data, e.g., processing `https` connections that use `ssl` encryption, prevent direct signature generation from the raw input because the bytes are not decrypted when Crispy encounters them. Although not included in the current implementation, Crispy could mitigate this issue by hooking library calls to common libraries, such as `libssl`, to handle such situations. Similar challenges arise when an application uses shared memory to read input, which can be addressed through manual or automatic internal hooking. Finally, in threaded applications, the thread that reads the input can pass the data to other threads. In these situations, the most recent input to the program may not be the input that actually causes a security violation. Instead, Crispy needs access to the most recent input to the individual thread, for example using a per-thread input buffer with internal hooks at the program point(s) where data are transferred to another thread. This would be a natural extension to Crispy, although it was not required for our evaluation, where the applications were configured to run in a single thread.

Red Team testing demonstrated that Crispy effectively deters most DoS attacks with minimal performance penalty. Each of the Red Team tests used for evaluation replicates a single attack repeatedly. An observant attacker could learn from her failed attempts and adapt attacks accordingly, providing motivation for implementing generalization and multi-part signatures. As mentioned earlier, however, generalizing signatures introduces a new risk that legitimate non-diverging inputs might be blocked, which is why Crispy adopts a conservative strategy.

Developing multi-part attack signatures is another avenue for future extensions, which would follow strategies used by biology. In Crispy, the multiple parts would contain information about the essential characteristics of an attack, impeding the attacker from avoiding Crispy's defenses through trivial changes, but would also have to be implemented carefully to avoid denial of legitimate requests. CRISPR/Cas avoids false positives by using two-part signatures: the spacer (the saved signature) plus an adjacent sequence, the Proto-Spacer Adjacent Motif (PAM). Because spacers are spliced into the host cell's genome, Cas proteins could easily target the host even when a virus is not present. To prevent this form of false positive, interference is not triggered unless the Cas protein finds both the spacer and the PAM; Mohanraju et al. call this double check a "quality control step" [27]. Two-part signatures in CRISPR/Cas systems are hypothesized to prevent viruses from using random mutation to evade the stored signatures, and also to improve the chances that such mutations are deleterious to the virus' fitness [68].

We looked to biology for solutions to N-variant systems' DoS vulnerability because at least part of the inspiration for diversity defenses and N-variant systems came from biology [7]. As an example of how closely our engineered design parallels the biological design, consider how the signatures are stored. Crispy stores the blacklist in a circular buffer, which mirrors the organization of CRISPR arrays. Cas proteins are believed to obey a first-in, first-out rule when adding spacers; evolution favors retaining information about recently encountered viruses which may still be present in the environment [6]. Identical logic applies to countering a DoS attack and was adopted for Crispy.

The biological CRISPR/Cas system must contend with false positives, similar to those faced by N-variant systems. Bacteria easily acquire new DNA through Horizontal Gene Transfer (HGT), integrating new genes from multiple sources, even across species [69]. HGT can help bacteria, for example, to acquire antibiotic resistance, but CRISPR/Cas can also block these new beneficial genes. Successful antibiotic resistance has been correlated with a defective CRISPR/Cas system [70]; the disabled CRISPR/Cas effectively overrides the false positives in these strains. Thus, bacteria apparently face trade-offs between false positives and false negatives, similar to Crispy. CRISPR/Cas can prevent beneficial HGT, but a weakened CRISPR/Cas defense leaves the cell vulnerable to viruses [71].

Genomic data suggests that the trade-off between CRISPR/Cas and HGT is more complex than outlined above; a bacterial population's adaptive immunity may itself adapt to the environment. For example, HGT seems to continue despite CRISPR/Cas because the genes for CRISPR/Cas systems can themselves be passed through HGT [72]. Therefore they persist in the bacterial genetic library, becoming more or less prevalent in a population based on selective pressures (such as antibiotic exposure). This resembles some intrusion detection systems (such as Rainbow [65]), which tune themselves dynamically as the threat model changes. N-variant or multiagent systems could benefit from a similar strategy, where each individual either is or is not endowed with a given defense (such as Crispy) and the overall composition of defenses changes in response to environmental feedback.

VII. Conclusions

Crispy addresses self-managing N-variant systems' vulnerability to DoS attacks by leveraging divergence signals to provide efficient and effective fully automated signature generation. Crispy uses information from past divergences to filter malicious inputs and thereby stop a DoS attack, similar to how bacteria prevent re-infection by viral DNA. Tested on the RAVEN N-variant system, Crispy prevented repeated divergences for 87.5% of attacks designed by a Red Team. By providing a defense against DoS, Crispy addresses a systematic weakness of N-variant systems, and the parallels between Crispy and its biological counterpart reinforce the relevance of biological defenses to cybersecurity engineering.

More generally, there are broad parallels between bacteria and self-managing systems, as illustrated by the tension between HGT and CRISPR/Cas. Over billions of years, bacteria have evolved mechanisms for distinguishing between helpful and harmful DNA. Self-managing systems face a similar challenge when identifying inputs (whether data or code) as malicious or benign. Crispy shows how concepts from bacterial immune systems can help defend self-managing systems and the analogy suggests several interesting avenues for future work. Biologists have much left to learn about bacterial immune systems [73], [74], and bacterial immune systems may have much more to teach us about cybersecurity.

Acknowledgments

Thanks to Andrew Milligan, Bruno Duterte, Cari Martinez, Joe Renzullo, Kirtus Leyba, Padraic Cashin, Petr Šulc, and Shannon Gallagher for their comments and suggestions.

References

[1] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006.

[2] S. Volckaert, B. De Sutter, T. De Baets, and K. De Bosschere, "Ghumvee: Efficient, effective, and flexible replication," in *Foundations and Practice of Security*, J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, A. Miri, and N. Tawbi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 261–277.

[3] J. Magee and T. Maibaum, "Towards specification, modelling and analysis of fault tolerance in self managed systems," in *Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems*, ser. SEAMS '06. New York, NY, USA: ACM, 2006, pp. 30–36.

[4] E. Yuan and S. Malek, "A taxonomy and survey of self-protecting software systems," in *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 109–118.

[5] M. Co, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong, W. Weimer, J. Burket, G. L. Frazier, T. M. Frazier, B. Dutertre, I. Mason, N. Shankar, and S. Forrest, "Double Helix and RAVEN: A system for cyber fault tolerance and recovery," in *Proceedings of the 11th Annual Cyber and Information Security Research Conference*, ser. CISRC '16. New York, NY, USA: ACM, 2016, pp. 17:1–17:4. [Online]. Available: http://doi.acm.org/10.1145/2897795.2897805

[6] P. Horvath and R. Barrangou, "CRISPR/Cas, the immune system of bacteria and archaea," *Science*, vol. 327, no. 5962, pp. 167–170, 2010. [Online]. Available: http://science.sciencemag.org/content/327/5962/167

[7] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Sixth Workshop on Hot Topics in Operating Systems*, 1998.

[8] D. Geer, C. P. Pfleeger, B. Schneier, J. S. Quarterman, P. Metzger, R. Bace, and P. Gutmann, "Cyberinsecurity: The cost of monopoly," Computer & Communications Industry Association, Tech. Rep., September 2003.

[9] B. D. Rodes, A. Nguyen-Tuong, J. D. Hiser, J. C. Knight, M. Co, and J. W. Davidson, "Defense against stack-based attacks using speculative stack layout transformation," in *Runtime Verification*, S. Qadeer and S. Tasiran, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 308–313.

[10] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 158–168. [Online]. Available: http://doi.acm.org/10.1145/1133981.1134000

[11] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson, "Security through redundant data diversity," in *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, June 2008, pp. 187–196.

[12] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 571–585.

[13] T. Jackson, B. Salamat, G. Wagner, C. Wimmer, and M. Franz, "On the effectiveness of multi-variant program execution for vulnerability detection and prevention," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, ser. MetriSec '10. New York, NY, USA: ACM, 2010, pp. 7:1–7:8.

[14] K. Koning, H. Bos, and C. Giuffrida, "Secure and efficient multi-variant execution using hardware-assisted process virtualization," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016, pp. 431–442.

[15] D. Bruschi, L. Cavallaro, and A. Lanzi, "Diversified process replicæ for defeating memory error exploits," in *2007 IEEE International Performance, Computing, and Communications Conference*, April 2007, pp. 434–441.

[16] S. Volckaert, B. Coppens, B. De Sutter, K. De Bosschere, P. Larsen, and M. Franz, "Taming parallelism in a multi-variant execution environment," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 270–285.

[17] K. Richards, R. LaSalle, F. van den Dool, and J. Kennedy-White, "2017 cost of cyber crime study," Accenture, Tech. Rep., 2017, https://www.accenture.com/us-en/insight-cost-of-cybercrime-2017.

[18] Ponemon Institute LLC, "2018 study on global megatrends in cybersecurity," Raytheon, Tech. Rep., 2018, https://www.raytheon.com/sites/default/files/2018-02/2018_Global_Cyber_Megatrends.pdf.

[19] P. Alcoy, S. Bjarnason, P. Bowen, C. F. Chui, K. Kasavchenko, and G. Sockrider, "13th annual worldwide infrastructure security report," NETSCOUT, Tech. Rep., 2018, https://resources.arbornetworks.com/.

[20] "CVE-2018-1303," National Vulnerability Database, March 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-1303

[21] "CVE-2018-8011," National Vulnerability Database, July 2018. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-8011

[22] J. Ali, "The new DDoS landscape," https://blog.cloudflare.com/the-new-ddos-landscape/, 2017.

[23] "Checkpoint/restore in userspace." [Online]. Available: https://criu.org

[24] J. Caballero, Z. Liang, P. Poosankam, and D. Song, "Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration," in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 161–181.

[25] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto, "ShieldGen: Automatic data patch generation for unknown vulnerabilities with informed probing," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, May 2007, pp. 252–266.

[26] J. C. Reynolds, J. Just, L. Clough, and R. Maglich, "On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization," in *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9 - Volume 9*, ser. HICSS '03. Washington, DC, USA: IEEE Computer Society, 2003.

[27] P. Mohanraju, K. S. Makarova, B. Zetsche, F. Zhang, E. V. Koonin, and J. van der Oost, "Diverse evolutionary roots and mechanistic variations of the CRISPR-Cas systems," *Science*, vol. 353, no. 6299, 2016. [Online]. Available: http://science.sciencemag.org/content/353/6299/aad5147

[28] MITRE Corporation, "Common weakness enumeration," 2018. [Online]. Available: https://cwe.mitre.org/

[29] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *IEEE Symposium on Computer Security and Privacy*. IEEE Computer Society Press, 1996, pp. 120–128.

[30] A. Mayer, T. Mora, O. Rivoire, and A. M. Walczak, "Diversity of immune strategies explained by adaptation to pathogen statistics," *Proceedings of the National Academy of Sciences*, 2016. [Online]. Available: https://www.pnas.org/content/early/2016/07/15/1600663113

[31] E. Westra, S. vanHoute, S. Oyesiku-Blakemore, B. Makin, J. Broniewski, A. Best, J. Bondy-Denomy, A. Davidson, M. Boots, and A. Buckling, "Parasite exposure drives selective evolution of constitutive versus inducible defense," *Current Biology*, vol. 25, no. 8, pp. 1043 – 1049, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0960982215001293

[32] A. Somayaji, S. Hofmeyr, and S. Forrest, "Principles of a computer immune system," in *Proceedings of the 1997 Workshop on New Security Paradigms*, ser. NSPW '97. New York, NY, USA: ACM, 1997, pp. 75–82. [Online]. Available: http://doi.acm.org/10.1145/283699.283742

[33] A. Graham, "Evolution: Parasite pressure favors fortress-like defence," *Current Biology*, vol. 25, no. 8, pp. R335 – R337, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0960982215002122

[34] P. Larsen, S. Brunthaler, and M. Franz, "Security through diversity: Are we there yet?" *IEEE Security & Privacy*, vol. 12, no. 2, pp. 28–35, Mar 2014.

[35] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *IEEE Security & Privacy*, vol. 12, no. 2, pp. 16–26, Mar 2014.

[36] Z. Liang and R. Sekar, "Fast and automated generation of attack signatures: A basis for building self-protecting servers," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 213–222. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102150

[37] "PaX documentation," 2003. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[38] 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 Conference on USENIX Security Symposium - Volume 12*, ser. USENIX-SS'03. Berkeley, CA, USA: USENIX Association, 2003. [Online]. Available: https://www.usenix.org/legacy/event/sec03/tech/full_papers/bhatkar/bhatkar_html/

[39] S. Kaur and M. Singh, "Automatic attack signature generation systems: A review," *IEEE Security & Privacy*, vol. 11, no. 6, pp. 54–61, Nov 2013.

[40] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song, "Sweeper: A lightweight end-to-end system for defending against fast worms," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 115–128.

[41] D. Brumley, D. Song, J. Chamcham, and X. Kovah, "Vulnerability-specific execution filtering for exploit prevention on commodity software," in *NDSS Symposium*, 2006.

[42] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *2006 IEEE Symposium on Security and Privacy (S P'06)*, May 2006, pp. 15 pp.–16.

[43] J. Newsome, B. Karp, and D. Song, "Polygraph: automatically generating signatures for polymorphic worms," in *2005 IEEE Symposium on Security and Privacy (S P'05)*, May 2005, pp. 226–241.

[44] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *NDSS Symposium*, 2005.

[45] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 133–147.

[46] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing software by blocking bad input," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 117–130.

[47] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267549.1267554

[48] A.-R. Sadeghi, L. Davi, and P. Larsen, "Securing legacy software against real-world code-reuse exploits: Utopia, alchemy, or possible future?" in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 55–61.

[49] G. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[50] 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*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 272–280. [Online]. Available: http://doi.acm.org/10.1145/948109.948146

[51] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson, "Zipr: Efficient static binary rewriting for security," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017, pp. 559–566.

[52] W. Hawkins, A. Nguyen-Tuong, J. D. Hiser, M. Co, and J. W. Davidson, "Mixr: Flexible runtime rerandomization for binaries," in *Proceedings of the 2017 Workshop on Moving Target Defense*, ser. MTD '17. New York, NY, USA: ACM, 2017, pp. 27–37.

[53] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 157–168.

[54] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 559–573.

[55] M. Taguinod, A. Doupé, Z. Zhao, and G.-J. Ahn, "Toward a moving target defense for web applications," in *Proceedings of the 2015 IEEE International Conference on Information Reuse and Integration*, ser. IRI '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 510–517.

[56] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genetic Programming and Evolvable Machines*, vol. 15, no. 3, pp. 281–312, Sep. 2014.

[57] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan 2012.

[58] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "ASSURE: Automatic software self-healing using rescue points," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 37–48.

[59] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 43–54.

[60] A. Cui and S. J. Stolfo, "Defending embedded systems with software symbiotes," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 358–377.

[61] H. Okhravi, A. Comella, E. Robinson, and J. Haines, "Creating a cyber moving target for critical infrastructure applications using platform diversity," *Elsevier International Journal of Critical Infrastructure Protection (IJCIP)*, vol. 5, no. 1, pp. 30–39, 2012.

[62] P. Laperdrix, W. Rudametkin, and B. Baudry, "Mitigating browser fingerprint tracking: Multi-level reconfiguration and diversification," in *Proceedings of the 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 98–108.

[63] H. Okhravi, J. Riordan, and K. Carter, "Quantitative Evaluation of Dynamic Platform Techniques as a Defensive Mechanism," in *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'14)*, ser. Lecture Notes in Computer Science (LNCS), Sep 2014.

[64] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis, "Defending against hitlist worms using network address space randomization," in *Proceedings of the 2005 ACM Workshop on Rapid Malcode*, ser. WORM '05. New York, NY, USA: ACM, 2005, pp. 30–40.

[65] B. Schmerl, J. Camara, G. Moreno, D. Garlan, and A. Mellinger, "Architecture-based self-adaptation for moving target defense," Carnegie Mellon University, Tech. Rep. CMU-ISR-14-109, 2014, https://apps.dtic.mil/dtic/tr/fulltext/u2/1019560.pdf.

[66] M. Demirbas and Y. Song, "An RSSI-based scheme for sybil attack detection in wireless sensor networks," in *Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, ser. WOWMOM '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 564–570. [Online]. Available: http://dx.doi.org/10.1109/WOWMOM.2006.27

[67] "CVE-2019-9590," National Vulnerability Database, March 2019. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-9590

[68] C. L. Sun, R. Barrangou, B. C. Thomas, P. Horvath, C. Fremaux, and J. F. Banfield, "Phage mutations in response to CRISPR diversification in a bacterial population," *Environmental Microbiology*, vol. 15, no. 2, pp. 463–470, February 2013.

[69] J. P. J. Hall, M. A. Brockhurst, and E. Harrison, "Sampling the mobile gene pool: innovation via horizontal gene transfer in bacteria." *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, vol. 372, no. 1735, 2017. [Online]. Available: http://search.proquest.com/docview/1955059867/

[70] K. Palmer and M. Gilmore, "Multidrug-resistant enterococci lack CRISPR-*cas*," *mBio*, vol. 1, no. 4, pp. e00 227–e00 227, 2010. [Online]. Available: http://search.proquest.com/docview/839680828/

[71] E. R. Westra, D. C. Swarts, R. H. Staals, M. M. Jore, S. J. Brouns, and J. van der Oost, "The CRISPRs, they are a-changin': How prokaryotes generate adaptive immunity," *Annual Review of Genetics*, vol. 46, no. 1, pp. 311–339, 2012, pMID: 23145983. [Online]. Available: https://doi.org/10.1146/annurev-genet-110711-155447

[72] U. Gophna, D. M. Kristensen, Y. I. Wolf, O. Popa, C. Drevet, and E. V. Koonin, "No evidence of inhibition of horizontal gene transfer by CRISPR-Cas on evolutionary timescales," *The ISME Journal*, vol. 9, no. 9, 2015.

[73] S. Doron, S. Melamed, G. Ofir, A. Leavitt, A. Lopatina, M. Keren, G. Amitai, and R. Sorek, "Systematic discovery of antiphage defense systems in the microbial pangenome," *Science*, vol. 359, no. 6379, 2018. [Online]. Available: http://science.sciencemag.org/content/359/6379/eaar4120

[74] E. V. Koonin, "Open questions: CRISPR biology," *BMC Biology*, vol. 16, no. 1, p. 95, Sep 2018. [Online]. Available: https://doi.org/10.1186/s12915-018-0565-9