To B or not to B: Blessing OS Commands with Software DNA Shotgun Sequencing

Anh Nguyen-Tuong, Jason D. Hiser, Michele Co,
Jack W. Davidson, John C. Knight
University of Virginia, Charlottesville, VA, USA
Contact author: an7s@virginia.edu

Nathan Kennedy, David Melski
GrammaTech, Inc.
Ithaca, NY, USA

William Ella, David Hyde
Raytheon
Arlington, VA, USA

***Abstract*—We introduce Software DNA Shotgun Sequencing (S^3), a novel, biologically-inspired approach to combat OS Injection Attacks, the #2 most dangerous software error as identified by MITRE. To thwart such attacks, researchers have advocated various forms of taint-tracking techniques. Despite promising results, e.g., few missed attacks and few false alarms, taint-tracking has not seen widespread adoption. Impediments to adoption include high overhead and difficulty of deployment. S^3 is based on a novel technique: *positive taint inference* which dynamically reassembles string fragments from a binary to infer *blessed*, i.e. trusted, parts of an OS command. S^3 incurs negligible performance overhead and is easy to deploy as it operates directly on binary programs.**

ACKNOWLEDGMENT

This material is based upon work supported by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

I. INTRODUCTION

Software weaknesses that lead to Operating System (OS) Command Injection Attacks are the #2 entry in MITRE's 2011 CWE/SANS list of Top 25 Most Dangerous Software Errors [1]. The high ranking makes intuitive sense: attackers that compromise an application can issue arbitrary commands to the underlying operating system, as if they were the owner of the application. The potential damages are especially catastrophic when the targeted applications are network-facing servers running with high privileges, e.g., file servers, mail servers, routers and even security appliances [2], [3], [4], [5].

In recent years, various taint-tracking techniques have been developed to thwart command injection attacks [6], [7], [8], [9], [10], [11], [12], [13], [14]. These techniques typically work by tracking the flow of data from an external source as it propagates through a program to a security-sensitive operation, such as network input flowing to a database command. Prior to issuing a security-sensitive operation, a command is first checked against its taint markings to ensure that critical parts of the command are not tainted. Modern taint trackers provide fine-grained resolution and keep track of taint markings at the level of individual characters. The resulting accuracy leads to few false alarms (false positives) and few undetected attacks (false negatives) [12], [11], [15], [16], [17], [18], [14].

Unfortunately, taint-tracking techniques are not practical for software binaries as keeping track of taint markings incurs high run-time overhead. Even state-of-the-art optimized taint trackers exhibit performance overhead between 50% and 200% [7]. In this paper, we focus on thwarting OS command injection attacks for *program binaries*, and seek a solution with the following characteristics[1]:

- **Operates on binaries**. The technique should operate directly on binaries, without requiring access to source code. In many deployment scenarios, source code will not be available, e.g., due to intellectual property protection measures, binary distribution, or use of legacy code.
- **Easy deployment**. The technique should be easy to apply and deploy. For example, it should not require the installation of a custom interpreter or significant changes in software development processes [12], [11], [15].
- **Low/no overhead**. A protected binary should incur very low overhead ($< 1\%$) during normal operation (i.e., while processing non-malicious commands).
- **Low rates of missed attacks while preserving normal functionality**. The technique should be effective at stopping attacks but it should not modify the functionality of protected binaries under normal operation.

The design landscape for run-time, taint-based defensive techniques is summarized in the following table:

	negative taint	**positive taint**
taint tracking	Haldar '05 [13] Newsome '05 [20] Nguyen-Tuong '05 [12] Pietraszek '06 [11] Futoransky '07 [15] Qin '06 [8], Xu '06 [16] Chin '09 [14] Bosman '11 [7] Papagiannis '11 [10]	Halfond '06 [17] Halfond '08 [18]
taint inference	Sekar '09 [21]	**S^3**

[1]These attributes were inspired by address space layout randomization, a security technique widely deployed across major commodity operating systems [19].

In one dimension, the focus is on keeping track of either untrusted data (*negative taint*) or trusted data (*positive taint*). In the other dimension, taint markings are derived either from tracking the flow of data through a program (*taint tracking*), or by inference (*taint inference*). S^3 investigates a combination of positive tainting and taint inference, the previously unexplored quadrant in this design space.

To meet our design goals, S^3 draws inspiration from *taint inference*, a technique described by Sekar [21]. Instead of instrumenting programs to keep track of the propagation of taint markings, Sekar's technique simply infers taint marking by correlating inputs to substrings in security-critical operations using an approximate string matching algorithm. By obviating the need to propagate taint, taint inference achieves low overhead. Sekar's taint inference technique relies on two main assumptions: (1) the accurate identification of external input data, and (2) that external data is mostly used verbatim when used in a command. These assumptions hold true for most web applications, but not for binary programs. For example, consider a server that uses various forms of data encoding or proprietary protocols to read input, and possibly uses shared memory to communicate with other programs. In this case, it is difficult and expensive to identify and monitor sources of input. Furthermore, if the input is encrypted or encoded, as is often the case with servers that use SSL, inferring taint markings based on the program's input becomes impractical.

S^3 captures the primary benefit of taint inference, i.e., low overhead, but uses positive tainting to obviate the needs of identifying sources of external data or relying on a readily observable correspondence between external input and critical commands.

The primary contributions of this paper are:

- We identify *positive taint inference*, a previously unexplored design point in the landscape of taint-based dynamic techniques.
- We demonstrate a realization of positive taint inference that we call *software DNA shotgun sequencing (S^3)*. S^3 forgoes in-depth and expensive program monitoring to infer taint markings.
- We highlight weaknesses in taint-based detection of OS command injection attacks, which motivates the need for better program specifications.
- We present and evaluate a working prototype of S^3, which effectively thwarts OS command injection attacks. S^3 has essentially no performance overhead and operates on binary programs, making it a practical, deployable solution to OS command injection attacks.

The remainder of the paper is organized as follows. Before presenting the details of our technique, we first present a threat model in Section II. Section III provides a high-level overview of S^3, using a simple, but realistic, working example. In Section IV we provide more details about each component of the S^3 architecture. We present a performance and security evaluation in Section V. We discuss security related issues with tainting and different aspects of S^3, including possible improvements, in Section VI. We discuss related work in Section VII and present concluding remarks in Section VIII.

II. Threat Model

Before describing S^3 in detail, it is necessary to understand the threat model of OS command injections that positive taint inference addresses.

The threat model assumes software is intended to be benign, but also that it likely contains flaws. The program, when run, reads untrusted user input possibly from many sources such as files, environment variables, shared memory, or network sockets. The input is used to create shell commands that are issued to the underlying operating system. Most inputs to the program are benign and cause the OS command to behave as intended by the programmer, but malicious inputs may exploit the program flaw to violate the security policy intended for the OS command. An OS command injection occurs when attacker-controlled inputs change the programmer-intended syntactic structure of a command [22], [21]. Further, the program may be performance sensitive, and cannot tolerate high run-time overhead.

This threat model includes the common "remote attacker" model where a server-type program receives malicious input over the network. However, it also includes privilege escalation attacks where a local user attempts to gain additional privileges (such as root access), by providing a malicious command line, environment variable, etc. to a program.

III. Software DNA Shotgun Sequencing: High-level Overview

Software DNA Shotgun Sequencing (S^3) is a technique inspired by genetic research [23]. In genetics, DNA shotgun sequencing breaks up very long DNA strands into short snippets, operates on (e.g., sequences) the snippets, and then recombines the results. Software DNA Shotgun Sequencing is similar in that we extract string fragments from a program, operate on them, and then later recombine them to validate some aspect of the program. Based on this idea, we invented the S^3 technique described in this paper. S^3 can thwart OS command injection attacks by matching the program's DNA fragments to the commands it attempts to issue. If commands cannot be matched, S^3 assumes that the DNA that has been injected into the program is potentionally dangerous.

S^3 differs from traditional taint-based techniques in two fundamental ways:

- S^3 does not seek to propagate taint markings as a program executes. Instead, it uses *taint inference*, a concept introduced by Sekar [21].
- However, in contrast to Sekar, S^3 infers taint markings for trusted data instead of untrusted data. The emphasis on trusted data is referred to as *positive tainting* and was developed by Halfond et al. [17], [18].

S^3 combines taint inference and positive tainting. We use the term ***positive taint inference*** to distinguish our work from Sekar's negative taint inference technique.

A. S^3 Architecture

S^3 consists of five major compoments. The goal of the **DNA Fragment Extraction** component (shown in Figure 1)

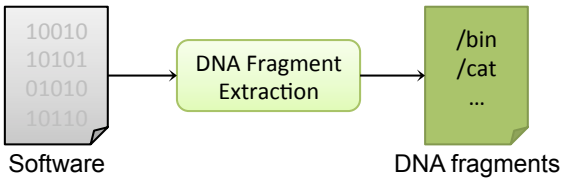

Fig. 1. DNA Fragment Extraction process.

is to extract string literals, i.e, DNA fragments, from the binary and its associated libraries. This analysis is done once, prior to program execution, and the analysis time is not counted against the run-time overhead.

The run-time components of S^3 are shown in Figure 2.

The **Command Interception** component intercepts security-critical commands so that they can be vetted.

The **Positive Taint Inference** component determines which characters in the intercepted command should be trusted by matching the command against the extracted DNA string fragments. Any unmatched character is deemed untrusted. Combining DNA fragments native to the protected binary to infer taint is a novel form of taint inference and one of the key contributions of the S^3 architecture.

The **Command Parsing** component parses the intercepted command to identify critical tokens and keywords.

The **Attack Detection** component combines the output of the Positive Taint Inference and Command Parsing component to determine whether an attack has occurred. A command is deemed an attack if a critical token or keyword is marked as untrusted.

Upon attack detection, S^3 either rejects the command outright and returns an error code, or it alters the command before passing it on to the operating system. The current prototype uses a simple form of error virtualization that simulates a failed command invocation by substituting an error code in place of the actual command [24], [25].

To illustrate how S^3 works, we use the following vulnerable program as a working example:

```
char *path = "/bin";
int main(int argc, char** argv) {
  char cmd[100];
  snprintf(cmd, 100,
           "%s/cat %s", path, argv[1]);
  system(cmd);
}
```

B. Example with Benign Input

When the program in the working example is passed a benign input such as `"README"`, the resulting command is shown in the first line of Figure 3a. The Positive Taint Inference component annotates each character in the command (`B` denotes that the character is trusted or *blessed*, `U` denotes untrusted), as denoted by the second line of the figure. In this case, `/bin/cat` is trusted as it matches the composition of the DNA fragments `"/bin"` and `"/cat"` extracted in the offline DNA Fragment Extraction process (see Figure 1). The Command Parsing component identifies critical tokens and keywords (`C` denotes critical), as shown in the third line of the figure. Lastly, the Attack Detection component takes as input the intercepted commands along with all annotations, and marks any critical command that is not blessed as untrusted. In this case, all critical tokens are marked as blessed, so the command is determined to be safe and is allowed to execute.

```
/bin/cat README
BBBBBBBBBUUUUUU
CCCCCCCC-------
```

(a)

```
/bin/cat README; rm -fr *
BBBBBBBBBUUUUUUUUUUUUUUU
CCCCCCCC-------C-CC-CCC--
                * ** ***
```

(b)

Fig. 3. Sample command, with S^3's "blessed" and "critical" markings for safe and malicious inputs.

C. Example with Attack Input

Consider the malicious input `README; rm -fr *` that seeks to recursively delete user files. The resulting command is shown in Figure 3b. Like the other example, the Positive Taint Inference component annotates each character in the command. Again, only `/bin/cat` matches the extracted fragments. The Command Parsing component identifies critical tokens and keywords, like before, except that this time the semicolon, `rm` command, and the `-fr` flags are also detected, as shown on line 3. Lastly, the Attack Detection component is invoked. It detects that there are critical command characters that are untrusted (shown with asterisks on line 4 of Figure 3b). Since S^3 has detected the attack, an appropriate remediation technique can be applied. The program can be shut down or the command can be blocked or sanitized before allowing it to be passed to the operating system.

IV. Software DNA Shotgun Sequencing: Detailed Overview

While the S^3 architecture is generic, e.g., it could be applied to web applications, we present details and discuss challenges encountered as we map S^3 into a practical instantiation to defeat OS command injection attacks for binary programs.

A. DNA Fragment Extraction

The accuracy of the fragment extraction process is crucial. If fragments are missed, valid commands might be flagged as injections. If spurious fragments are extracted, malicious command injections might not be flagged (See Section VI-B for further discussion).

1) String extraction: Extracting string fragments from binary programs is more difficult than it first appears. Our first attempt used the Linux program `strings`, which linearly scans a binary program and extracts null-terminated sequences of ASCII characters that have a length larger than a given

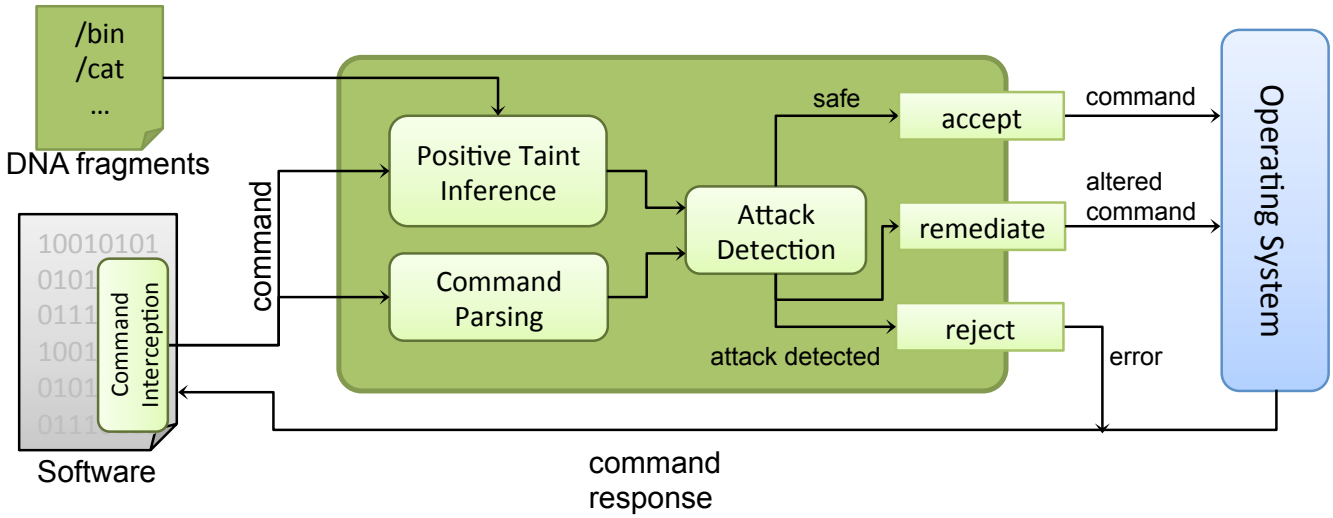

Fig. 2. Software DNA Shotgun Sequencing (S^3) Architecture.

threshold. Unfortunately, short strings are sometimes important. Consider this C++ snippet:

```
string q = "rm ";
q += "-f ";
q += filename;
system(q.c_str());
```

which creates and executes an OS command.

Using `strings`, the threshold needs to be sufficiently low to find short strings. Unfortunately, low thresholds tend to yield lots of garbage strings, which affects accuracy. Furthermore, compilers use many optimizations that can make strings harder to detect. For example, to initialize a string on the stack, a compiler might use a sequence of store instructions:

```
mov [esp+28], 0x2d206d72 # "rm -"
mov [esp+32], 0x00002066 # "f \0\0"
```

Each move stores four bytes onto the stack, ultimately creating the proper null-terminated string. Other compiler idioms may complicate accurately finding all strings, as well. We have seen examples of the compiler inlining some standard library functions that have constant operands, such as `memcpy(dst,"rm -f ", 6)`. This optimization yields inlined constants much like the previous string initialization example. Lastly, `strings` reports all strings in the executable file, which can include debug information, shared library names, compiler-version identifiers, etc. As these types of strings cannot be used to form OS commands, they should be excluded from consideration.

To deal with these issues, we use static analysis of the program to derive the string fragments. The static analysis starts by fully disassembling the program into a database which holds each instruction in the program, indexable by address, function, and control flow information. We use a hybrid linear-scan disassembler and recursive-descent disassembler to ensure we get good coverage of all instructions, as described by Hiser, et al. [26], [27].

After disassembly is complete, the instructions are scanned for accesses or creation of string values. We analyze each instruction's immediate operands and apply three heuristics to identify string fragments:

- Check if the immediate value holds the address of a program location and the location is the beginning of a sequence of printable characters or one printable character terminated by a null byte.
- Check immediate values to see if they contain a string fragment. Attempt to combine immediate values of sequential instructions to form one string fragment. This heuristic handles the case of strings constructed via sequential store instructions, as described in the previous example.
- For position-idependent code (PIC), check immediate values for PIC-relative addressing that might point to a string.

Finally, we check for other string fragments or string pointers in data sections.

2) Post-processing of DNA Fragments: Programs compiled from C or C++ often contain statements that use format specifiers, e.g. `%d`, `%f`, `%s`, `%x`. We split such fragments into their constituent sub-fragments using the format specifiers as delimiters. A fragment such as

```
"/bin/rm -f \%s; /bin/touch \%s"
```

would be split into the sub-fragments `"/bin/rm -f "`, and `"; /bin/touch"`. As the analysis cannot be sure that such fragments are used as format strings, the original fragment as well as the sub-fragments are retained in the list of signatures.

Figure 4 shows a representative sampling of the DNA fragments from the Spam Assassin program. The length of the fragments range from 1-111 characters. Because of the `%s` specifier, Fragment 11 expands into sub-fragments 34 and 173. Likewise, fragment 46 expands into fragment 57 and 314.

```
  3: Usage: spamass-milter -p socket
     [-b|-B bucket] [-d xx[,yy...]]
     [-D host]
 11: popen failed(%s). Will not send
           a copy to spambucket
 33: recipients; spamc gets
           default username
 34: ). Will not send a copy
           to spambucket
 46: Could not extract score from <%s>
 56: error. could not replace body.
 57: Could not extract score from <
173: popen failed(
273: hX
274: hx
279: h8
281: h@
282: h(
287: ><
286: @+
288: Z
304: 0
305: 8
306: @
307: (
308: "
309: .
310: /
311: :
312: '
313: _
314: >
315: `
```

Fig. 4. Sample fragments manually extracted from SpamAssassin Milter Plugin [4] (28 shown out of 315 fragments total).

Fragments 273–282 are likely spurious and result from the inherent imprecision of static analysis on binaries.

Astute readers will notice the short fragments that contain potentially dangerous shell metacharacters (fragments 306-315), or short fragments that could be composed in an attack (fragments 273-288). In Sections IV-E and VI, we discuss how the S^3 policies deal with short and potentially dangerous fragments.

B. Command Interception

For binaries derived from C/C++ programs, commands are typically encapsulated in an Application Programming Interface (API) and accessed via dynamically-linked shared libraries.

The S^3 prototype leverages standard library interposition facilities (`LD_PRELOAD`) to transparently intercept and wrap function calls to the underlying operating system. S^3 intercepts the `system`, `popen`, `rcmd`, and `exec` family of functions. Other functions could obviously be intercepted as well, but we have identified these as the primary candidates for OS command injection.

C. Command Parsing

This component is responsible for identifying the security-critical parts of a command. For OS commands, the critical parts consist of command names, options, delimiters, and the setting of environment variables.

The S^3 prototype uses a simple, combined lexical analyzer and parser. The parser is careful to identify special characters which could indicate the start of a new command (such as the semicolon character), match quotation marks and parentheses, etc. Ideally, one would use a full, formally-verified lexical analyzer and distinct parser to detect keywords, etc. However, it is impossible due to the nature of the shell language (`bash` in our case). Consider this command:

```
echo Touching ${file}; touch `foobar`
```

What are the "correct" lexical analysis and parse for this command? The answer depends on the value of the `file` variable and the output of the `foobar` executable. If `file` is set to a single quote character and `foobar` returns the same thing, then there is exactly one command, `echo`. Since variables are expanded and sub-processes are executed before the command is parsed, the correct parse cannot be determined *a priori*. Under most circumstances, though, such odd substitutions are not the case.

For the purposes of detecting OS command injections, we need to know the possible places where a command could be invoked. Our simple parser assumes that the structure of the command is not changed by the results of executing sub-commands. In the case of our simple example, the parser marks the command like so:

```
echo Touching ${file}; touch `foobar`
CCCC          CCCCCCCC CCCCC CCCCCCCC
```

where `C` indicates that a critical command character exists at the given location.

D. Positive Taint Inference

Conceptually, the Positive Taint Inference component infers which portions of the command come from within the program, and which ones come from external sources. To accomplish this step, it checks each substring in the command to determine if that location is within the set of DNA fragments. This pseudocode illustrates the process:

```
for each DNA fragment, f
  for each position, i, in the command
    l=len(f)
    if f==command[i .. i+l-1]
      mark_blessed(command[i .. i+l-1]);
```

Conceptually, this algorithm could be quite expensive, $O(n^3)$ where $n = max(len(sig), \#sigs, len(command))$. In practice, though, we use a move-to-front heuristic to organize the DNA fragments required to trust commands and exit the outermost loop when all critical parts of the command are marked as blessed. Further, each command and each DNA fragment is typically short, on the orders of tens or hundreds

<commandName>
;[\s]<commandName>
$(<commandName>
||[\s]<commandName>
&&[\s]<commandName>
`<commandName>
<environmentVar>[\s]=
-<optionFlags>
--<optionFlags>

Fig. 5. Attack detection policies using the *same fragment origin policy* ([\s] denotes an optional whitespace).

```
/bin/cat README; rm -fr *
BBBBBBBBBUUUUUUUUUUUUUUU
CCCCCCCCC       C CC CCC
```

Fig. 6. Overlapping policies to detect attacks.

of characters. These simple observations and adjustments dramatically reduce the time necessary to make the inference. Section V-D empirically evaluates the overhead associated with inferring trust markings.

E. Attack Detection

Attack detection consists of scanning the command for any character that has been marked as untrusted by the Positive Taint Inference component and critical by the Command Parsing component. In addition, we impose the constraints shown in Figure 5 that command names, shell metacharacters used for starting subcommands and their associated command names, option flags, and environment variable names must come from a single DNA fragment (*same fragment origin policy*).

This policy helps to compensate for the case when a short, critical token, such as a semi-colon or a quotation mark, is present in the set of DNA fragments. Such fragments allow attackers great latitude to create strings that append new commands, as in "`; rm -rf`". Unfortunately, these DNA fragments cannot simply be discarded, because many programs do use such fragments to terminate their commands. However, it appears uncommon for a program to use such fragments to introduce a new command, so we disallow this behavior entirely.

Figure 6 illustrates how these policies provide overlapping means to detect attacks. The core policy of checking for untrusted critical characters (shown in boldface red) is augmented with the same fragment origin policy (shown with rectangles). Note that `rm` is covered by three separate policies. Thus, even if `;` and `rm` were somehow both extracted as fragments, the attack would still be detected.

When no attack is detected, S^3 passes the command to the operating system to execute. However, if an attack is detected, S^3 does not pass through the command, but can enact any one of a variety of remediation responses, such as shutting down the program, warning the user and asking for permission to continue, or logging the attack. For the prototype described in this paper, we chose to return an error code as if the library call had failed. This policy makes sense in many cases, as well-written programs are designed to gracefully handle error conditions.

Section VI discusses potential sources of false negatives and false positives, and their implications in further detail.

V. Evaluation

To evaluate the security and performance of S^3, we have built a prototype and applied it to a variety of engineered and real-world benchmarks. The following sections describe the Experimental Setup, Benchmarks, Performance Evaluation and Security Evaluation in more detail.

A. Experimental Setup

For our evaluation, we used a 32-bit VirtualBox virtual machine running Ubuntu 12.04 with 4 GB of RAM and a 2 GHz Xeon E5-2620 processor.

B. Benchmarks

To evaluate the performance and security of S^3, we have collected a variety of benchmarks. For real-world benchmarks with CVE reports, we used the SpamAssassin Milter Plug-in [28], an email filter interface for detecting spam, and cbrPager [29] version 0.9.16, a program to decompress and view high-resolution images. We configured SpamAssassin Milter version 0.3.1 with SpamAssassin version 3.3.2 and Postfix version 2.9.6. Both of these programs have real-world OS command injection vulnerabilities.

We also used a set of vulnerable programs independently developed by MITRE Corporation from real-world, open-source software. Each program was seeded with a command injection vulnerability. This process was repeated to create many variants with the vulnerability at many locations. Each variant has inputs that represent normal program input, as well as exploit inputs.

Finally, we used a set of small exploitable programs, most less than 100 lines, that were developed by Raytheon. Like the MITRE programs, normal and exploit inputs are provided.

Lastly, to help evaluate performance, we developed a series of microbenchmarks. These benchmarks create an OS command from command line input, and use a tight loop to execute that command as frequently as possible, doing no other work. There are two dimensions of variation in the micro benchmarks: 1) the command to be executed and 2) the primitive used to invoke the command. There are two possible commands to be executed. The command to be executed in one case is `echo hello`, and in the second case is `bzip2 dickens.txt` [30], [31]. The two cases represent a fast command and a somewhat more reasonable workload that compresses a 775 KB file. Each microbenchmark uses one of the following primitives to invoke the command: `execv`, `popen`, or `system`.

C. Security Evaluation

We used a combination of programs with reported real-world vulnerabilities, synthetic test programs, and real-world programs seeded with vulnerabilities by an independent testing team to evaluate the strength of the S^3 approach.

Benchmark Type	Benchmark	Native (ms $\pm$ 95%CI)	S^3 (ms $\pm$ 95%CI)	Absolute Diff. (ms)	% difference
MITRE Seeded	C-C078-NGIN-04-DT03-02	10.6 ± 1.5	10.8 ± 1.8	0.2*	2.1%*
MITRE Seeded	C-C078-CHER-04-DF09-02	11.7 ± 3.3	11.5 ± 1.4	-0.2*	-1.5%*
Real world	spamass-milter 0.3.1	87 ± 70	82 ± 43	-4.4*	-5%*
Real world	cbrPager 0.9.16	121.3 ± 11.6	121.2 ± 9.0	-0.1*	-0.1%*
Micro	echo (system)	$1,126 \pm 18$	$1,222 \pm 5$	96	8.4%
Micro	echo (popen)	$1,236 \pm 12$	$1,240 \pm 6$	4	0.32%
Micro	echo (execv)	$1,243 \pm 4$	$1,514 \pm 11$	271	22%
Micro	bzip2 (system)	$1,377 \pm 13$	$1,380 \pm 14$	2.8	0.2%
Micro	bzip2 (popen)	$1,379 \pm 12$	$1,381 \pm 12$	3.0	0.2%
Micro	bzip2 (execv)	$1,366 \pm 14$	$1,373 \pm 13$	7.2	0.5%

TABLE I. PERFORMANCE OVERHEAD IN MILLISECONDS. ASTERISKS INDICATE THE DIFFERENCES ARE NOT STATISTICALLY SIGNIFICANT FOR THE 50 TRIAL RUNS PERFORMED.

1) Real-World Attacks: We evaluated S^3 against two reported command-injection vulnerabilities that we were able to reproduce in open-source binaries.

The first attack, based on CVE-2008-2575, is a command injection in `cbrPager` [32]. To extract images, `cbrPager` invokes the `system` library call to execute unzip or unrar on the archive, without sanitizing the filename. By crafting an input such as `";rm -rf *;".cbr` and providing it where a filename is expected, `cbrPager` is tricked into executing a malicious command when it attempts to open the putative file. S^3 is able to detect attempts to open a malicious filename and return an error from the `system` library call. These actions result in the program displaying a message that the file cannot be opened, and exiting harmlessly.

The second attack, based on CVE-2010-1132, is a remote exploit in the SpamAssassin Milter Plugin [4] (spamass-milter), which integrates the SpamAssassin spam filter with either sendmail or Postfix. The vulnerability occurs when the milter is invoked with the `-x` "expand" option, to pass the email address through alias and virtusertable expansion to allow emails to be redirected to other accounts. In this case, the `popen` function is invoked on sendmail with the email address provided from SMTP as an argument, without properly sanitizing the email address, which can contain a pipe character. With an SMTP command such as `RCPT TO:<username+:"|rm /var/spool/mail">`, arbitrary commands can be executed; with careful crafting, these may be sufficient to open a remote shell. Our technology was able to harmlessly block any command injections. The signatures extracted from spamass-milter do not include the vertical bar (pipe) character, foiling any attempt to exploit this weakness. Moreover, the Milter plugin properly error-checks the `popen` function call, so it continues to function without loss of service in the face of an attempted exploit.

2) Synthetic Attacks: We evaluated S^3 against engineered test suites developed by Raytheon and independently by MITRE. The Raytheon engineered suite consists of 18 microtests demonstrating command injections with 22 good inputs and 35 bad inputs, using 9 different function calls (`[f]exec[l,le,lp,v,ve,vp]`, `system` and `popen`) and a variety of input-processing techniques. S^3 mitigates all of the bad inputs while breaking none of the good inputs in this test suite.

The MITRE test suite includes 477 OS command injection (based on CWE-78) and 516 OS argument injection (based on CWE-88) test cases [33]. These test cases are based on inserting vulnerabilities into seven base programs: `Cherokee`, `grep`, `nginx`, `tcpdump`, `wget`, `w3c` (from `libwww`), and `zsh`. Each test case involves inserting a vulnerable call to `popen` at various locations in the base program. For the CWE-78 test cases, this call invokes `nslookup` with an unsanitized argument specified from an environment variable or untrusted file. For the CWE-88, the program builds the command using the format string "`find / -iname %s`." Semicolon characters are properly sanitized when constructing the command, but the user can still include input that has a `-exec` argument that is ultimately passed to `find`. Consequently, they could use an input such as "`* -exec rm {} \;`" to remove files or execute other commands. For each test case, ten good inputs and two bad inputs are provided. In each case, S^3 was able to intercept the bad inputs without altering behavior on the good inputs.

D. Performance Evaluation

Table I shows the performance overhead of S^3. The columns show the type of benchmark, benchmark name, performance timing without and with S^3 and finally an absolute and percentage difference, indicating the slowdown S^3 introduces. A 95% confidence interval is shown where appropriate.

We selected two of the benchmarks from the MITRE suite where the seeded vulnerability was in the main loop of a server; most vulnerabilities were injected into startup or shutdown code, and there was no significant performance difference. The seeded vulnerability was set to execute only once, but for timing purposes we modified the code slightly so that it executed on every request to the server. The benchmarks are based on `Cherokee` (C-C078-CHER-04-DF09-02) and `nginx` (C-C078-NGIN-04-DT03-02), two production-quality web servers. We performed 50 timings, each consisting of downloading a small HTML file (574 bytes). Even with the seeded vulnerability in the main loop and the small download size, no statistically significant difference in timing was observed with S^3.

For SpamAssassin Milter, we wrote a simple client that uses `gettimeofday` to measure the time spent in processing an email transaction. We also modified `cbrPager` to measure the time to render the first page of a 49 MB input file. Like the MITRE benchmarks, these benchmarks show no statistically significant overhead.

Unfortunately, the server applications have relatively high variance due to network latencies, disk caching, etc. To deal with this issue and benchmark worst-case overhead, we use the

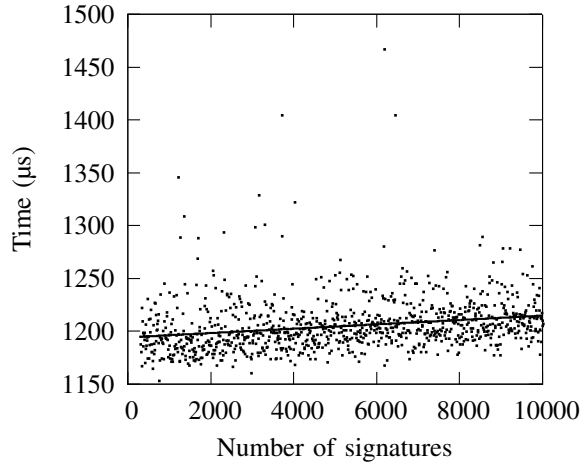

Fig. 7. Average time to invoke `system` versus number of signatures.

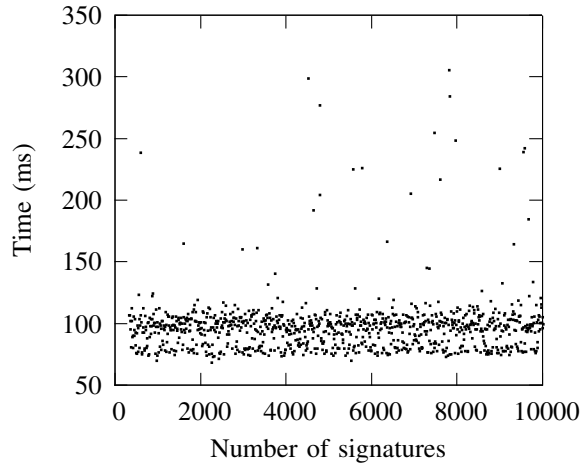

Fig. 8. Average time for email transaction versus number of milter signatures.

microbenchmarks described in Section V-C2. For these benchmarks, we perform 50 timings, where each timing invokes 10 or 1,000 OS commands for the `bzip2` or `echo` microbenchmark, respectively. The microbenchmarks that invoke `bzip2` to compress a file show that S^3 causes practically no overhead, only 0.2%. The true worst-case performance overhead is when the program does nothing but issue OS commands, and each OS command invocation completes extremely quickly. This case is represented by the microbenchmark that issues the `echo hello` command. These benchmarks show that the absolute worst case overhead might be as high as 22%. However, in practice the actual work performed by the program and by executing the OS command clearly dominates the overall runtime. Only our worst-case microbenchmarks demonstrate that S^3 generates any measurable overhead.

To verify the move-to-front heuristic was working properly, we measured the overhead of the echo microbenchmark that uses the `system` function to invoke OS commands as we vary the number of DNA fragments. We automatically added randomly generated strings to the program's DNA fragments. Figure 7 shows the average time in microseconds for the microbenchmark to execute the `system` call over 100 trials,

Benchmark	Size (KiB)	Total (s)	Extraction (s)
spamass-milter 0.3.1	142	17.3	0.60
cbrPager 0.9.16	198	25.8	0.69
C-C078-NGIN-04-DT03-02	708	238.6	8.3
C-C078-CHER-04-DF09-02	548	224.2	4.8

TABLE II. ANALYSIS TIME IN SECONDS.

using from 300 to 10,000 signatures (timing starts after steady state has been reached). There is a very slight positive correlation as shown by the line of best fit $y = 0.002x + 1194$. Our investigation indicates that the command processing and matching time after initialization was fixed across the differing number of signatures, but that as there are more signatures in the process's address space, the `fork` system call (used to implement `system`) takes longer. We suspect this behavior is a result of taking slightly longer to copy additional page table entries for the new process.

In practice, this additional overhead is negligible since most programs have few signatures. For example, SpamAssassin Milter has 316 signatures and `nginx` has 2,017. Figure 8 shows the average time in milliseconds to process an email transaction over 50 trials applying S^3 with from 320 to 10,000 signatures to SpamAssassin Milter, which shows no trend. We would not have expected to see any correlation, given an expected increase of only 20 microseconds based on our microbenchmark and the higher time variance of the email benchmark.

Based on these microbenchmark and real-world benchmark performance results, we believe that in practice the S^3 system would introduce no measurable overhead, and is the fastest OS command injection detector to date.

E. Analysis Time

We measured the time for offline analysis (i.e., DNA Fragment Extraction) of the real-world benchmarks. The results are shown in Table II. This table shows the size of the analyzed executables and libraries, the entire static analysis time, and the portion of that time spent in fragment extraction and processing. This analysis needs to be performed only once. Our results show the analysis taking up to four minutes for `nginx`. The time is dominated by the disassembly and IR recovery steps that can be shared by other binary analyses and protections. The actual time devoted to string extraction and post-processing amounts to about 2% of the analysis, completing in between 0.5 to 8 seconds on our benchmarks.

VI. SECURITY DISCUSSION

A. Spurious Attack Detection (False Positives)

The current S^3 prototype makes the assumption that command words originate from static strings in the binary or dependent libraries. If this assumption is violated, e.g., commands are stored in configuration files, S^3 may incorrectly flag a benign command as an attack, thereby breaking programs. As observed by Halfond et al., these situations can often be detected easily during a pre-deployment testing phase [18]. Since the S^3 architecture cleanly separates the specification of fragments from their use at run-time, additional fragments can be incorporated simply by appending to the DNA fragment list

(Figure 1). This step could be done manually or automatically through further tool support.

Another possibility is that the strings in the program that form commands cannot be detected via static analysis of the binary program. Such a situation might occur if the program is encrypted or highly obfuscated. Such situations are expected to be uncommon. However, we do not have experimental evaluation of the prevalence of obfuscated binaries and our evaluation suite may not be representative in this respect.

B. Missed Attack Detection (False Negatives)

The extraction heuristic introduced in Section IV-A may lead to spurious fragments, i.e., string fragments that do not actually exist in the binary's source code. Spurious fragments may also be introduced as a result of separating fragments containing format string specifiers into sub-fragments (Section IV-A2). For example, the fragment `"echo '%s'"` would be sub-divided into the fragments `"echo '"` and `"'"`. These spurious fragments may inadvertently match special shell operators or command words that might be useful in an attack, e.g., `'`, `` ` ``, `<`, `>`, `$(`, `|`, `;`, `&`.

Another issue is that a binary may truly contain dangerous string fragments. Consider the following code snippet used for filtering out dangerous characters:

```
if (strstr(s, "'") ||
    strstr(s, "`") ||
    strstr(s, ">")) ...
```

Or code that uses concatenation to assemble strings:

```
string s = "echo '";
s += name;
s += "'";
```

Such strings may thwart the ability of S^3 to detect all attacks. The current S^3 system handles this situation by using the policies described in Section IV-E. These policies enforce the constraints that critical command names, along with shell characters that start sub-command, must come from a single signature. These policies also handle the case where there may be fragments for each character in the alphabet, which could be reassembled to form any command name. Such a situation might be common for some programs that utilize many different command line options: programs often contain each command line option as a single character string! Despite this, S^3 has been able to detect all command injections in practice using the policies described previously.

1) Future Work: Reducing the Attack Surface: We plan on investigating simple data flow analyses methods to prune the fragments to just the set which might reach an OS command site. We would omit a fragment if we could prove that it never flowed into a critical command, as is the case with the `strstr` example above. Furthermore, we plan on refining our fragment matching algorithm to allow for regular expressions. Instead of breaking up fragments when they contain a format string specifier, we would instead substitute the corresponding regular expression pattern specifier, e.g., `[0-9]+` for `%d`.

C. Subtle Injections

Some "command injections" are particularly difficult to detect using tainting information. Consider this program snippet:

```
system("make");
```

The parameter to the command execution library is constant. However, if the program has root privileges and the user has the ability to control the executable search path, they can modify the `make` command to do as they wish, and avoid the programmer's intended security policies. Likewise, consider this program snippet:

```
sprintf(buf,"cat %s", argv[1]);
system(buf);
```

The user is presumably allowed to specify a file to be displayed by the program. However, if the user specifies a file with a relative or absolute pathname, a security policy may be violated. Perhaps even trickier is if the user specifies no file at all, and the command becomes simply `cat` with no parameter. Terminal input is then used instead of a file on the file system. Again, no command is "injected" into the program.

Lastly, if the program specifies that an external intepreter should be used to interpret commands, detection may be challenging. Consider this program snippet:

```
sprintf(buf,"echo %s | bc", argv[1]);
system(buf);
```

The program snippet indicates that the user should be able to specify input to the `bc` program. However, `bc` accepts a large variety of commands, which may have effects that were not anticipated by the programmer. Many programs in a standard Linux install have this characteristic, e.g. `bash -c`, `zsh`, `find -exec`, `psql -c`, `printf`, etc., all have flags that allow them to interpret arbitrary commands. Furthermore, there are many non-standard interpreters. There is no *a priori* way to establish whether a program is an interpreter or not, which language it might accept, and which parts of the language are intended by the programmer.

While each of these cases is hard to detect with S^3, they also represent the most challenging command injections to handle automatically. They represent the fundamental problem that programmer intent is not typically available. Without clear, correct, and formally represented programmer intent, no tool can detect all OS command injections. Even expensive taint propagation systems, which are considered largely effective, would be ineffective against the attacks shown.

VII. Related Work

We focus our discussion on software-based, run-time defensive techniques.

A. Taint Tracking

1) Taint Tracking in Managed Runtimes: Livshits provides an extensive review of dynamic taint tracking projects [6]. Most projects use a form of negative taint tracking, i.e., these projects keep track of external (untrusted) data as it flows through a program, and check whether such data is used in a security-sensitive operation [13], [12], [11], [16], [17], [18], [14]. The notable exception is the WASP project by Halfond et al. which uses positive taint tracking to keep track of internal

(trusted) data [18]. The primary trade-off is that positive taint tracking favors false positives (breaking application functionality) whereas negative taint tracking favors false negatives (missing attacks). Halfond advocates the use of positive taint tracking as it provides a more conservative security posture.

Unfortunately, both positive and negative taint tracking are seldom used in practice. Perl and Ruby are the only two major languages that we are aware of that provide support for dynamic taint tracking out of the box [34], [35]. Several projects modified the PHP run-time engine to support taint tracking at the level of individual characters [12], [11], [15]. To avoid modifying the PHP run-time engine, PHP Aspis applies source code transformations selectively to only parts of a web application. This scheme maps well to extensible applications whose core is well-maintained but where the quality of third-party plugins is unknown [10], [36]. Despite the selective application of taint markings, Aspis incurs high overhead of 2.2X on Wordpress. Haldar et al. provided coarse-grained taint tracking for Java strings [13]. Chin and Wagner implemented taint tracking at the level of characters for Java [14]. In general, fine-grained approaches to taint tracking result in higher precision and fewer false positives.

2) Taint Tracking for Binaries: The overhead numbers reported for the systems highlighted below are illustrative of the rapid rate of progress in reducing the overhead of taint-tracking techniques on binaries. However, they should not be directly compared to one another as the benchmarks and hardware used vary across these projects.

TaintCheck, one of the early pioneering projects for using taint tracking to detect memory-overwriting attacks on binaries incurred overhead as high as 37X for CPU-bound applications, and from 2.5X to 25X for I/O bound workloads for a typical web server [20]. TaintCheck was built on top of Valgrind, a flexible but relatively slow dynamic binary rewriter [37]. The LIFT project achieved overhead of 3.6X on several SPEC INT2000 benchmarks and 6.2% on server applications [8]. The order of magnitude improvement resulted from several optimizations, including using a more efficient binary rewriter, eliminating instrumentation on provably safe code paths, coalescing checks and reducing the overhead of context switching between the application code and the dynamic binary rewriter. Bosman et al. reported overhead of 2.4X for SPEC INT2006 and 1.5X-3X for real-world applications using an emulator custom-built for taint analysis [7]. Unlike the previous approaches, Saxena et al. use static rewriting as the mechanism for instrumenting binary code [38]. They reported average overhead of 1.95X on several CPU-intensive SPEC95 INT benchmarks. Dytan [39] and libdft [9] incorporate years of experiences with taint tracking to provide easily customizable and generic taint analysis frameworks.

Despite steady and impressive progress in improving the performance of taint-tracking techniques, our stringent overhead requirements ($< 1\%$ on binaries [40]) led us to bypass taint-tracking techniques altogether.

B. Taint Inference

S^3 was heavily influenced by Sekar's taint inference technique for protecting web applications against command injection attacks [21]. Sekar's insight of establishing taint markings by correlating inputs to observable commands obviated the need for taint tracking and was the key to enabling practical performance. Instead of inferring taint markings for untrusted data, S^3 seeks to infer trusted data used in OS commands. To highlight this fundamental difference, we view S^3 as an embodiment of *positive* taint inference, in contrast to Sekar's use of *negative* taint inference.

C. Model-based Approaches

Christensen et al. perform static analysis to model possible string values at any point in a Java program [41]. The model extracted represents an over-approximation of the programmers' intended specification for benign commands. The AMNESIA project leverages these models to detect and prevent SQL injection attacks [42]. We believe that AMNESIA can be extended to cover OS command injections. The overhead reported on a set of Java web applications was negligible.

String analysis for binaries is much more challenging as binary code does not retain as much type information as Java byte code. Christodorescu et al. modeled strings for x86 binaries, though the precision of the analysis is limited by the lack of interprocedural analysis [43]. Sophisticated memory-analysis techniques such as Value-Set Analysis (VSA) could also be applied to string extraction [44]. However, it seems likely that string extraction requires abstract domains that are designed for reasoning about strings. VSA uses an abstract domain based on reduced interval congruences which is excellent for reasoning about (strided increments of) pointer values, but likely to lead to imprecise representation of string values.

By extracting and allowing for the arbitrary combination of string fragments, S^3 makes a conscious trade-off between model complexity and model accuracy. S^3 combines a very simple (but over-approximated) string model with additional policies based on the origin of string fragments for its attack detection policies.

VIII. Conclusion

This paper has described Software DNA Shotgun Sequencing (S^3), the first look at a new, efficient, approach for detecting taint markings based on positive taint inference. Our findings indicate that S^3 can be effectively used to detect OS command injection attacks on binary programs. Furthermore, S^3 has demonstrated that it can be used in many real-world situations because it has negligible performance overhead and can be applied directly to binary programs without need for source code or compiler support. Future work consists of refining the string DNA extraction process, expanding the attack classes covered to include SQL injections, LDAP injections, XML injections and cross-site scripting attacks (XSS), and adapting S^3 to mobile and web applications.

Acknowledgments: This research is supported by 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 AFRL, ARO, DoD, or the U.S. Government.

REFERENCES

[1] The MITRE Corporation, "2011 CWE/SANS top 25 most dangerous software errors." [Online]. Available: http://cwe.mitre.org/top25/

[2] "CVE-2003-0041: MIT kerberos FTP client remote shell commands execution," 2003. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0041

[3] "CVE-2007-3572: Yoggie Pico Pro remote code execution," 2007. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3572

[4] "CVE-2010-1132: SpamAssassin mail filter: Arbitrary shell command injection," 2010. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1132

[5] "CVE-2013-3568: Linksys CSRF + root command injection," 2013. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3568

[6] B. Livshits, "Dynamic taint tracking in managed runtimes," *Microsoft Research Technical Report*, 2012.

[7] E. Bosman, A. Slowinska, and H. Bos, "Minemu: the world's fastest taint tracker," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–20. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23644-0_1

[8] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 135–148. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.29

[9] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: ACM, 2012, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/2151024.2151042

[10] I. Papagiannis, M. Migliavacca, and P. Pietzuch, "PHP Aspis: using partial taint tracking to protect against injection attacks," in *2nd USENIX Conference on Web Application Development*, 2011, p. 13.

[11] T. Pietraszek and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Recent Advances in Intrusion Detection*. Springer, 2006, pp. 124–145.

[12] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *20th IFIP International Information Security Conference*. Springer, 2005, pp. 372–382.

[13] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for java," in *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005, pp. 303–311.

[14] E. Chin and D. Wagner, "Efficient character-level taint tracking for java," in *Proceedings of the 2009 ACM Workshop on Secure Web Services*, ser. SWS '09. New York, NY, USA: ACM, 2009, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/1655121.1655125

[15] A. Futoransky, E. Gutesman, and A. Waissbein, "A dynamic technique for enhancing the security and privacy of web applications," *Proc. Black Hat USA*, 2007.

[16] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *Proceedings of the 15th USENIX Security Symposium*, 2006, pp. 121–136.

[17] W. G. J. Halfond, A. Orso, and P. Manolios, "Using positive tainting and syntax-aware evaluation to counter SQL injection attacks," in *Proceedings of the 14th ACM SIGSOFT International symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 175–185. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181797

[18] ——, "WASP: Protecting web applications using positive tainting and syntax-aware evaluation," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 65–81, 2008.

[19] "Address space layout randomization." [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[20] J. Newsome, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *12th Annual Network and Distributed System Security Symposium*, 2005.

[21] R. Sekar, "An efficient black-box technique for defeating web application attacks," in *Network and Distributed System Security Symposium*, 2009.

[22] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 372–382.

[23] Wikipedia, "Shotgun sequencing." [Online]. Available: http://en.wikipedia.org/wiki/Shotgun_sequencing

[24] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis, "Building a reactive immune system for software services," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247371

[25] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis, "A dynamic mechanism for recovering from buffer overflow attacks," in *Information Security*. Springer, 2005, pp. 1–15.

[26] 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. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.39

[27] J. D. Hiser, C. L. Coleman, M. Co, and J. W. Davidson, "MEDS: The memory error detection system," in *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, ser. ESSoS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 164–179.

[28] G. C. F. Greve, M. Brown, and D. Nelson, "SpamAssassin milter plugin," 2013. [Online]. Available: http://savannah.nongnu.org/projects/spamass-milt/

[29] J. Coppens, "cbrPager," 2013. [Online]. Available: http://jcoppens.com/soft/cbrpager/index.en.php

[30] J. Seward, "bzip2," 2013. [Online]. Available: http://www.bzip.org

[31] C. Dickens, *A Tale of Two Cities*. Project Gutenberg, 1859. [Online]. Available: http://www.gutenberg.org/files/98/98.txt

[32] "CVE-2008-2575: cbrPager: Arbitrary command execution via shell metacharacters," 2008. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2575

[33] The MITRE Corporation, "Common weakness enumeration." [Online]. Available: http://cwe.mitre.org/

[34] "The perl programming language." [Online]. Available: http://www.perl.org

[35] "Ruby." [Online]. Available: http://www.ruby-lang.org

[36] "WordPress." [Online]. Available: http://www.wordpress.org

[37] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proc. of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[38] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proc. of the 6th Annual IEEE/ACM Intl. Symposium on Code Generation and Optimization*, ser. CGO '08. New York, NY, USA: ACM, 2008, pp. 74–83.

[39] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proc. of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 196–206.

[40] M. Co, J. W. Davidson, J. D. Hiser, J. C. Knight, A. Nguyen-Tuong, D. Cok, D. Gopan, D. Melski, W. Lee, C. Song, T. Bracewell, D. Hyde, and B. Mastropietro, "PEASOUP: Preventing exploits against software of uncertain provenance (position paper)," in *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, ser. SESS '11. New York, NY, USA: ACM, 2011, pp. 43–49. [Online]. Available: http://doi.acm.org/10.1145/1988630.1988639

[41] A. S. Christensen, A. Møller, and M. I. Schwartzbach, *Precise analysis of string expressions*. Springer, 2003.

[42] W. G. J. Halfond and A. Orso, "AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 174–183. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101935

[43] M. Christodorescu, N. Kidd, and W.-H. Goh, "String analysis for x86 binaries," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '05. New York, NY, USA: ACM, 2005, pp. 88–95. [Online]. Available: http://doi.acm.org/10.1145/1108792.1108814

[44] T. Reps and G. Balakrishnan, "Improved memory-access analysis for x86 executables," in *Proc. of the Joint European Conferences on Theory and Practice of Software 17th intl. conference on Compiler construction*, ser. CC'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 16–35.