Taint-Enhanced Anomaly Detection*

Lorenzo Cavallaro[1] and R. Sekar[2]

[1] Department of Computer Science, Vrije Universiteit Amsterdam, The Netherlands
[2] Department of Computer Science, Stony Brook University, USA

Abstract. Anomaly detection has been popular for a long time due to its ability to detect novel attacks. However, its practical deployment has been limited due to false positives. Taint-based techniques, on the other hand, can avoid false positives for many common exploits (e.g., code or script injection), but their applicability to a broader range of attacks (non-control data attacks, path traversals, race condition attacks, and other unknown attacks) is limited by the need for accurate policies on the use of tainted data. In this paper, we develop a new approach that combines the strengths of these approaches. Our combination is very effective, detecting attack types that have been problematic for taint-based techniques, while significantly cutting down the false positives experienced by anomaly detection. The intuitive justification for this result is that a successful attack involves unusual program behaviors that are exercised by an attacker. Anomaly detection identifies unusual behaviors, while fine-grained taint can filter out behaviors that do not seem controlled by attacker-provided data.

1 Introduction

System-call based anomaly detection has been popular among researchers due to its effectiveness in detecting novel application-layer attacks [1, 4, 7, 8, 10, 13, 26, 29, 30, 32]. These techniques typically learn a model of an application's behavior during a training phase, which is then compared with behaviors observed during a detection phase. Deviations are flagged as potential intrusions. A key benefit of these techniques is that they require no policy specification. They are thus ideal for detecting unknown attacks.

The key assumption behind anomaly detection techniques is that attacks manifest unusual program behaviors. While experience to date supports this assumption, the converse does not hold: not all unusual behaviors are attacks. As a result, anomaly detection suffers from a high rate of false positives that impacts its practical deployment.

Recently, taint-tracking approaches [28, 19, 22, 20, 33] have become popular for defending against common software exploits. Their strength stems from their ability to accurately reason about the use of untrusted data (i.e., data that may be coming from an attacker) in security-critical operations. By using policies to distinguish between safe and unsafe uses of "tainted" data, these techniques can detect many common software vulnerability exploits, including those based on memory corruption [28, 19, 33], and SQL, command or script injection [20, 22, 27, 33, 25].

The main advantage of taint-tracking approaches is that accurate, application-independent policies can be developed for the above attacks. These policies express the general principle that *tainted data should be limited to non-control purposes;* and *control-data, such as code pointers, scripts, or commands, should be untainted.* Since attackers prize their ability to take control over a victim application, taint policy enforcement has

* This work was supported in part by an NSF grant CNS-0831298, an ONR grant N000140710928, and an AFOSR grant FA9550-09-1-0539.

proved to be very effective against the most popular attacks prevalent today. At the same time, due to the difficulty of developing accurate policies, many less popular (but still very dangerous) data attacks are not addressed by taint-based techniques, e.g., memory corruption attacks on non-control data [3], path traversals, race conditions. Anomaly detection remains the best option for such attacks.

We present a new technique, called *taint-enhanced anomaly detection* (TEAD), that combines the strengths of system-call-based anomaly detection with fine-grained taint-tracking. This combination is effective, detecting attack types that have been problematic for taint-based techniques, while significantly cutting down the false positives experienced by anomaly detection. The intuitive justification for this result is that a successful attack involves unusual program behaviors that are exercised by an attacker. Anomaly detection identifies unusual behaviors, while fine-grained taint can reduce false positives by filtering out behaviors that are not dictated by attacker-provided data.

As with any other taint-based technique, TEAD begins with a specification of the set of taint-sources and taint-sinks. Currently, our taint-sinks include all system calls and a few other functions such as `printf`, and functions used for communicating with external entities such as database servers or command interpreters. Like many previous techniques [26, 4, 8, 7], our models rely on the contexts in which sink functions are invoked. This model is augmented with information about taintedness of arguments. In the simplest case, this information will indicate whether each argument of a sink function is tainted. More generally, the model captures information such as the components of aggregate data (e.g., fields of a C-structure, or components of a path name) that can be tainted, or the lexical structure of tainted data (e.g., whether tainted data should be alphanumeric or can contain various special characters). Attacks are flagged when there is a significant change from tainting patterns observed during training. Some of the advantages that TEAD can offer are:

1. Since tainted events correspond to a subset of events observed at runtime, TEAD's scope for false positives (FPs) is correspondingly reduced. In particular, TEAD can work with limited training data on untainted events since it triggers alarms only on tainted events. Note that a reduction in false positives can have an indirect effect on reducing false negatives (FNs), since a lower FP can allow the detection threshold to be lowered.
2. TEAD can be combined with more inclusive notions of taint, including those that account for control-flows. Previous taint-based vulnerability defenses have largely ignored control-dependencies, while also limiting taint propagation via pointers. This was done in order to reduce false positives. With TEAD, the training phase can help reduce these false positives by discarding those control dependences that were also observed during training.
3. TEAD is deployed in conjunction with taint policies to guard against most common exploits such as control-flow hijacks. This combination can mitigate problems faced by learning-based techniques due to attacks in training data—the most common attack types can be filtered out by removing event traces that violate taint policies. Likewise, it can also improve resistance to sophisticated mimicry attacks [31, 12], since these attacks rely on control-flow hijacks. Indeed, since taint-tracking involves reasoning about input/output data, TEAD does not suffer from a versatile

mimicry attack [21] that can defeat previous system-call anomaly detection techniques that all ignored arguments to operations such as `read` and `write` [1, 4, 7, 8, 10, 13, 26, 29, 30, 32].

2 Approach Description

2.1 Fine-grained taint-tracking

TEAD relies on fine-grained taint information. In principle, it could be based on many of the existing techniques for this purpose, but clearly, performance is an important factor. For this reason, our implementation relies on DIVA [33], which is implemented using a source-to-source transformation of C programs. On server programs—often the focus of intrusion detection—DIVA has reported an overhead of about 5%. Binary taint-trackers exist as well [24], although they tend to be less mature and/or more expensive.

Like all taint-tracking techniques, DIVA relies on a specification of taint sources such as network read operations. In particular, on return from a system call that is a taint source, DIVA uses taint source specifications to mark the data returned by the system call as tainted. DIVA is a byte-granularity taint-tracker, and hence will mark each byte of data read from an untrusted source as tainted. This taint information is maintained in a global bit-array called `tagmap`. In particular, the taint associated with a byte of memory located at an address `A` is given by `tagmap[A]`. DIVA's source-to-source transformation ensures that data reads, arithmetic operations, and data writes are all augmented so as to propagate taint. DIVA can transform arbitrary C-programs, including various libraries, when their source-code is available. If source is unavailable, it can make use of summarization functions that capture taint propagation. In particular, after a call to a function f whose source code is unavailable, DIVA's transformation will introduce a call to f's summarization function in order to correctly propagate taint. (DIVA's taint-source marking specifications are a special case of summarization functions.)

Although DIVA's focus is on capturing direct data flows, it does incorporate features to track some control flows. These features enable it to correctly handle certain frequently encountered constructs such as the use of translation tables. This factor reduced the need for using full control dependency tracking in our implementation of TEAD. Even though DIVA operates on C-programs, it is applicable to many interpreted languages such as PHP and shell scripts. This applicability has been achieved by transforming the interpreters themselves to perform taint-tracking. As discussed before, TEAD enforces policies that block control-flow hijack, SQL injection and command injection attacks during its operation. This is done using the policies incorporated into DIVA as described in [33].

2.2 Taint-Enhanced Behavior Models

As with any other taint-based technique, TEAD begins with a specification of the set of taint-sources and taint-sinks. Currently, our taint-sinks include all system calls and a few other functions such as `main`, and `printf`, and functions used for communicating with external entities such as database servers or command interpreters. Unlike policy enforcement techniques such as DIVA that require policies to be associated with each sink, TEAD simply requires the identification of sinks. As a result, TEAD can handle numerous sinks without any significant specification efforts.

Since taint is a property of data, we focus primarily on learning properties of system call arguments. Let Σ be the set of all the sinks, and $s(a_1, a_2, \cdots, a_n) \in \Sigma$ be a generic sink, where $a_1, a_2, \cdots, a_n$ denote the sink's arguments. Rather than learning properties common to all invocations of s, our approach learns properties that are specific to each context in which s is invoked. In our prototype, the context is simply the calling location, except that calls made within shared libraries are traced back to program locations from which these library functions were invoked [26] (more refined notions of contexts, such as those of [4, 8, 7] could be used as well). The use of calling contexts increases the accuracy of models. For instance, a context-sensitive model can distinguish between `open` system calls made from two different parts of a program. If one of these is used to open a configuration file and the other one is used to open a data file, then it would be possible for the model to capture the intuitive property that a configuration file name cannot be controlled by the attacker, but a data file name may be.

As with other anomaly-based approaches, TEAD builds models during a *learning* phase. Deviations from this model are identified and reported as anomalies during a *detection* phase. Below, we detail the types of information embedded into TEAD models.

2.3 Coarse-grained Taint Properties

For each argument a_i of each sink s, TEAD learns if any of its bytes are tainted. More generally, TEAD could learn whether a_i has control dependence, data dependence, or no dependence on tainted inputs, but control dependence is not tracked in our current prototype. For pointer arguments, taintedness of pointers could be learned, as well as the taintedness of the objects pointed by the pointer. At detection time, an alarm is raised if an argument that was not tainted during training is now found to be tainted. This approach can detect many buffer overflow attacks that modify system call arguments, as opposed to modifying control flows. For instance, an attack may overwrite a filename that is supposed to represent a file containing public data with `/etc/shadow`. This may allow the attacker to obtain the contents of `/etc/shadow` that he may subsequently use for an offline dictionary attack.

2.4 Fine-grained Taint Properties

For aggregate data, the above approach may lose too much information by combining taint across the entire data structure. To improve precision, TEAD refines taint properties to capture more details regarding different parts of the data that may be tainted. The simplest case to handle in this regard are C-structures. For them, TEAD associates a taint with each field of a `struct`. This is particularly useful for some system calls, e.g., `sendmsg`, `writev`, etc. A more complex case concerns non-struct data, such as strings. String data, such as file names, database queries, scripts and commands are frequently targeted in data attacks. TEAD includes several algorithms for learning fine-grained taint properties that is motivated by such use of string data. We organize these algorithms based on whether a sink argument a_i is *fully* or *partially* tainted.

Properties of fully tainted arguments

Maximum length (MaxTaintLen). This property is an approximation of the maximum permissible length l_{max} of a tainted argument. This helps to detect attacks that overflow buffers with the intent to overwrite security sensitive data.

Structural inference (StructInf). Often, an attacker may not try to overflow any buffers. Instead, he may try to modify the normal structure of an argument to bypass some security checks. To this end, the structure of a_i is inferred so that each byte is clustered in proper class. Currently, our model classifies (or maps) uppercase letters (A-Z) to the class represented by `A`, lowercase letters (a-z) to `a`, numbers (0-9) to `0`, and so on. Each other byte belongs to a class on its own. For instance, if the model sees an `open("/etc/passwd", ...)` system call invocation, the finite state automaton (FSA) which is generated for the string `/etc/passwd` will recognize the language `/a*/a*`. We further simplify the obtained FSA by removing byte repetition, as we are not concerned about learning lengths with this model. The final FSA will thus recognize the simplified language `/a/a`. If during detection the structure of the considered argument is different from the one learned, an alarm will be raised.

It can be noted that for particular sinks, trying to infer their (tainted) argument structure can lead to FPs if the structure for that sink is highly variable, e.g., arbitrary binary data read from the network. For this reason, our prototype limits fine-grained learning to those sinks and arguments where it is explicitly specified. An alternative possibility is to limit it to string data (i.e., `char *`).

Properties of partially tainted arguments

In this case, a tainted argument consists of a combination of tainted and untainted bytes. The tainted portion is subjected to the learning of the aforementioned properties, while the following learning rules are considered for the untainted part.

Untainted common prefix (UCP). It is often the case for string-valued data that the interpretation of the trailing components is determined by the leading components. Examples include filenames which are interpreted within the directory specified by the leading components of the string; and command arguments that are interpreted by a leading command name. TEAD learns the longest common untainted prefix for every sink argument that is partially tainted. This algorithm can be easily generalized to learn a small number of common prefixes, rather than a single prefix.

Allowable set of tainted characters (ATC). Many applications expect tainted data will be limited to a subset of the alphabet. For instance, tainted components of filenames may be expected to be free of "`/`" and "`.`" characters in some contexts. In other contexts, they may be expected to be free of special characters such as "`;`" or whitespace characters. To capture such constraints, we can learn the set of characters that cannot appear in tainted context. To improve convergence, we utilize character classes as before, e.g., upper-case and lower-case characters and numbers. But most punctuation and whitespace characters form a class of their own.

3 Implementation

As mentioned before, our TEAD prototype relies on the DIVA implementation, which takes a program P as input and produces P_T, a semantically-equivalent taint-enhanced version of it. In particular, for *every* taint sink (or source) f, DIVA allows us to associate a wrapper f_w that will be called before (or after) it. Our prototype uses these wrappers to learn properties of sink arguments, as well as for marking source arguments as tainted (e.g., those coming from the network). We enable DIVA policies that protect against control-flow hijack and command injection attacks during detection phase, and to filter out attack-containing traces during training.

P_T is monitored during the *training* phase and a log file is created. The log file includes sink names and their context information (e.g., calling site), sink arguments and, for each argument, byte-granularity taint information. For instance, a typical log entry looks like the following:

```
read@0x8048f5c 3 arg0={ A:U } arg1={ A:U V[0-98]:T C:99:0:ls -la } arg2={ A:U }
```

The meaning is as follows. The sink name (`read`) is followed by its calling site (`0x8048f5c`). Next, the number of arguments follows (`3`) and details about these arguments are recorded. For instance, the entry for the second argument (`arg1`) tells us that the address (`A`) where the sink buffer of size 99 (`V[0-98]`) is stored is untainted (`A:U`), while the buffer content is tainted (`V[0-98]:T`). The content of the tainted buffer, which starts at offset 0, is `ls -la`[3]. This information will be used by the next step.

The log file is analyzed *off-line* to build a profile $\mathcal{M}$ of the behavior of P_T by using the aforementioned information. In particular, (i) identical events, that is events whose names and call sites are identical, are merged into a single event instance, and (ii) *untainted* events are inserted into the model just for evaluation reason.

For instance, considering the previous example, the tainted sink `read` invoked at the calling site `0x8048f5c` has its first and third argument untainted, while the second argument a_1 is tainted. Moreover, l_{max}, the maximum length for a_2 is 99 while, its structure is given by `a -a`. The profile created during this step is serialized and reloaded during the next step. P_T is then monitored during the *detection* phase. Deviations from the model learned in the previous step are reported.

4 Evaluation

4.1 Effectiveness in Detecting Attacks

TEAD main focus is on non-control data attacks. This section considers attacks taken from [3] and other sources. Where necessary, we slightly modify the example to show that our approach is effective even when some of the specifics of an attack are changed.

[3] To avoid noise into the log file we actually base64 encode buffer contents which are decoded by the off-line log analyzer to create the application profile.

Format String Attack against User Identity Data A version of `WU-FTPD` is vulnerable to a format string vulnerability in the `SITE EXEC` command. The non-control data exploit of this vulnerability, described by Chen *et al.* [3], is based on the following code snippet that is used to restore the effective userid of the ftp server to that of the user logged in. (This restoration follows operations where the server briefly needed to utilize root privilege to perform `setsockopt` operation.)

```
1 FILE *getdatasock(...) {
2     ...
3     seteuid(0);
4     setsockopt(...);
5     ...
6     seteuid(pw->pw_uid);
7     ...
```

The attack aims to overwrite the `pw_uid` field to zero, so that the restoration code will leave the process with root-privileges. This would enable the current user, a non-privileged user, to assume root privileges on the system hosting the server.

Our approach detects this attacks in two different ways. It either considers whether the `seteuid`'s argument is tainted, or it detects structural divergence in the tainted arguments of the `printf`-like function used to exploit the format string vulnerability. The latter method relies on the presence of a particular memory error vulnerability, and can be detected using taint policies as well. For this reason, we focus on the former method. In particular, our approach learns that the `seteuid` argument `pw->pw_uid` at line 6 was always *untainted* during training. During an attack, `pw->pw_uid` is marked as tainted, since it was overwritten by copying some attacker provided data. This causes the argument of `seteuid` to become tainted, thus raising an alarm.

It is worth pointing out that, in this context, taint-based learning seems to provide better results than what could be achieved with a conventional anomaly detection technique, even if the latter relies on very comprehensive training data. For instance, a conventional anomaly detector observing a limited number of authenticated users may be vulnerable to an attack where attackers are able to impersonate one of such users.

Heap Corruption Attacks against Configuration Data We report on two heap-based memory error vulnerabilities and attacks as described by Chen *et al.* [3].

Null HTTPD This attack aims to overwrite the `CGI-BIN` configuration string. Note that the name of every CGI program invoked by a client will be prefixed with this string. Thus, by changing it from its default value of `/usr/local/httpd/cgi-bin` to the string `/bin`, a malicious client would be able to execute programs such as the shell interpreter on the server.

In this scenario, it can be observed that the available options for the attacker are mainly two: (a) to either completely overwrite the original `CGI-BIN` configuration string, or (b) partially overwrite the configuration string. In this latter case, the goal would typically be to perform a path traversal to ascend above the original `CGI-BIN` directory and then to descend into a directory such as `/bin` or `/usr/bin`. For simplicity, let us consider that the sink of interest here is the `open` system call.

(a) During training, our approach would learn that the first argument of `open` system call invoked at a context $\mathcal{C}$ is a combination of untainted data (i.e., the original `CGI-BIN` configuration string) and tainted data (i.e., the command derived from untrusted input); and has an *untainted prefix* that includes all of the original `CGI-BIN` string. Thus, the UCP model will detect this attack since the common untainted prefix observed during training is no longer present.
(b) In this case, the untainted prefix property may still hold. However, the set of allowable tainted characters (i.e., the ATC model) are different due to the presence of tainted characters such as "`.`" and "`/`". In addition, structural inference would have learned previously that the tainted component of `open` argument consisted of alphanumeric characters, whereas now it has a different structure that consists of a long, alternating sequence of alphanumeric and special characters.

Netkit Telnetd The attack described in [3] exploits a heap-based buffer overflow vulnerability. It aims to corrupt the program name which is invoked upon login request by referencing the `loginprg` variable as shown by the following code snippet.

```
void start_login(char *host, ...) {
    addarg(&argv, loginprg);
    addarg(&argv, "-h");
    addarg(&argv, host);
    addarg(&argv, "-p");
    execve(loginprg, argv);
}
```

As a result of a successful attack, the application invokes the program interpreter `/bin/sh -h -p -p` (underlined characters are tainted). This raises an alarm in the UCP model: during detection, the untainted prefix contained the entire command name, which should be longer than the current untainted prefix `/bin`.

SquirrelMail Command Injection Shell command injections can be somewhat tricky for policy-based techniques. In particular, a typical policy would be one that prevents tainted whitespace or shell metacharacters except inside quoted strings. Unfortunately, such a policy is susceptible to false positives as well as false negatives. False positives arise with applications that permit untrusted users to specify multiple command arguments. (SquirrelMail itself is one such application.) These applications may, in fact, be incorporating checks to ensure that the arguments are safe. On the other hand, false negatives may arise because a string literal may be passed down to a command that further parses the string into components that are handed down to another command interpreter, e.g., consider `bash -c 'ls -l xyz; rm *'`. For these reasons, we believe command injections are better handled by a technique such as TEAD that can utilize learning to fine-tune the characters that can legitimately appear within arguments.

Specifically, SquirrelMail version 1.4.0 suffered from a shell command injection vulnerability in version 1.1 of its GPG plugin. The vulnerability involves the use of a shell command to invoke `gpg` to encrypt email contents. `gpg` program needs to access the public key of the recipient, so the recipient name should be provided as a command-line argument. SquirrelMail retrieves the name of the recipient from the "to" field on the email composition form. An attacker can provide a malicious value for this email field, such as "`nobody; rm -rf *`" that causes SquirrelMail to delete files. This attack can easily be detected by the ATC model: in the absence of attacks, tainted characters in the `execve` argument will never include shell delimiters such as semicolons.

Stack Buffer Overflow Attack against User Input Data The exploitation of this stack-based buffer overflow vulnerability was somewhat tricky but the authors of [3] were able to bypass the directory traversal sanity check enforced by the application. In summary, after the directory traversal check and before the input usage, a data pointer is changed so that it points to a second string which is not subjected to the application-specific sanity check anymore, thus it can contain the attack pattern (similar to a TOCTTOU). Other than this TOCTTOU aspect, this attack is very similar to case (b) of the Null HTTPD attack, and hence is detected in the same way.

Straight Overflow on Tainted Data The following example is from Mutz *et al.* [18]. The memory error attack is simple. The `user_filename` array obtained at line 7 (`gets` function) is guarded by a security check (`privileged_file` function at line 9) which checks whether `user_filename` specifies a name of a privileged file or not. In the affirmative case, the program prints an error message and quits. Otherwise (i.e., a non privileged file), more data is read into the array `user_data` using the function `gets` at line 14, and the file name specified by `user_filename` is opened at line 15. Instead of corrupting `write_user_data` return address, an attacker can overwrite past the end of `user_data` and overflow into `user_filename`. As the overflow happens after the security check performed at line 9, an attacker can specify a privileged file name for `user_filename` that will be replaced subsequently by the overflow attack.

```
 1 void write_user_data(void) {
 2
 3     FILE * fp;
 4     char user_filename[256];
 5     char user_data[256];
 6
 7     gets(user_filename);
 8
 9     if (privileged_file(user_filename)) {
10       fprintf(stderr, "Illegal␣filename.␣Exiting.\n");
11       exit(1);
12     }
13     else {
14       gets(user_data);          // overflow into user_filename
15       fp = fopen(user_filename, "w");
16       if (fp) { fprintf(fp, "%s", user_data); fclose(fp); }
17     }
18 }
```

Our approach detects this data attack by learning the maximum length l_{max} of the tainted arguments of the `gets` invoked at line 7, and 14, during the learning phase.

Format Bug to Bypass Authentication The following example has been described by Kong *et al.* in [11]. Normally, the variable `auth` is set to 1 or 0 depending on the fact that the right authentication credential is given as input or not (line 5). An attacker, however, can exploit the format string vulnerability at line 11 and overwrite `auth` with a non-null value so that the subsequent check of the credential at line 12 will grant an access to the system.

```
 1 void do_auth(char *passwd) {
 2     char buf[40];
 3     int auth;
 4 
 5     if (!strcmp("encrypted_passwd", passwd))
 6         auth = 1;
 7     else
 8         auth = 0;
 9 
10     scanf("%39s", buf);
11     printf(buf);        // format string
12     if (auth) access_granted();
13 }
```

This attack can be stopped by modeling tainted format string directives. By modeling the *tainted* format string of the `printf` function invoked at line 11 our approach learns whether tainted format directives have been used during the training step, along with their structure (structural inference on tainted arguments). If no tainted formatting directives are learned during the learning phase, then no tainted formatting directives can be subsequently encountered during detection phase without raising an alarm.

4.2 False Positives

Table 1 shows the false positives rate we obtained by conducting experiments on the `proftpd` ftp server and `apache` web server. Table 2 attributes these false positives to each of the models used.

Program	# Traces (Learning)	# Traces (Detection)	Overall FP rate
`proftpd`	$68,851$	$983,740$	1.7×10^{-4}
`apache`	$58,868$	$688,100$	2.5×10^{-3}

Table 1. Overall False Positives.

As shown by Table 2, the majority of false positives were caused by violation of structural inference models. We expected a relatively high number of false positives as the model proposed in § 2 is a simple non-probabilistic model. The main focus of this paper is to show the benefits of using taint information to improve the false positive rates of anomaly detection, so we have not emphasized the use of sophisticated learning techniques. The use of more sophisticated learning techniques (e.g., [17, 1]) is orthogonal to our technique, and can further reduce false positives.

Program	Tainted Events	UCP	StructInf	MaxTaintLen	Overall FP Rate
proftpd	3.0×10^{-5}	0	1.4×10^{-4}	0	1.7×10^{-4}
apache	0	0	2.4×10^{-3}	0	2.5×10^{-3}

Table 2. False Positives Breakdown.

To assess the effectiveness of taint information in reducing the false positives of learning-based anomaly detection, we carried out the following test. We computed the false positive rate of the anomaly detection techniques underlying TEAD, when taint information is ignored. In particular, an alarm is counted each time a system call s is invoked in a context different from those observed during training. This led to false positive rates of 2.4×10^{-4} and 4.3×10^{-4} for `proftpd` and `apache` respectively. We then compute the false positive rate that would be observed when taintedness of the argument was taken into account. In particular, an alarm was raised only if the anomalous system call was also associated with anomalous taint, i.e., a previously untainted argument was now tainted. This reduced the false positives on these programs to 3.0×10^{-5} and zero respectively. Thus, the use of taint information reduces the false positive rate by about an order of magnitude or more.

As a second way to assess the impact of taint-tracking on false positives, we compared the fraction of system calls that were tainted during the learning and detection phase. As Table 3 depicts, half of the traces of `apache` have been considered during detection, while only a small fraction of them have been considered for `proftpd`. By omitting the rest, TEAD can avoid FPs that may arise due to them.

Program	# Traces (Learning)	# Tainted (%)	# Traces (Detection)	# Tainted (%)
proftpd	68,851	2,986 (4.3%)	983,740	7,120 (0.72%)
apache	58,868	46,059 (82.1%)	688,100	354,371 (51.5%)

Table 3. Fraction of tainted system calls.

4.3 Performance overheads

The dominant source of overhead in TEAD is that of taint-tracking. Overhead due to DIVA has been reported in [33] to be about 5% for I/O-intensive applications such as `apache`, and about 50% for CPU-intensive applications. Most real-world applications experience overheads in between these two figures.

Our preliminary results indicate that the additional overhead introduced by TEAD is relatively low. So far, we have measured only the overheads due to `proftpd`, but we hope to measure `apache` overhead in the near-future. In particular, for `proftpd`, we experienced slowdowns of 3.10% due to taint-tracking only, 5.90% due to taint-tracking and model profiling during the learning phase, and 9.30% due to taint-tracking and model matching during the detection phase. `proftpd` overheads were measured when the program was subjected to a variety of operations, including changing of directories, file uploads and downloads, and recursive directory listing.

Note that taint-learning overhead includes only the overhead of logging, and omits the cost of offline learning that uses these logs. In contrast, detection is performed online, and hence its overheads are higher.

We point out that unlike DIVA, whose overhead increases for CPU-intensive computation, TEAD's anomaly detectors experience higher overheads for I/O-intensive computations, e.g., computations characterized by a high rate of system calls.

5 Related Work

Anomaly detection based on system calls. Forrest *et al.* [6, 9] first introduced anomaly detection techniques based on system calls made by applications. This system is built following the intuition that the "normal" behavior of a program P can be characterized by the sequences of system calls it invokes during its executions in an attack-free environment. In the original model, the characteristic patterns of such sequences, known as N-grams, are placed in a database and they represent the language L characterizing the normal behavior of P. To detect intrusions, sequences of system calls of a fixed length are collected during a detection phase, and compared against the contents of the database. This technique was subsequently generalized in [32] to support variable-length system-call sequences.

The N-gram model is simple and efficient but it is associated with a relatively high false alarm rate, mainly because some *correlations* among system calls are not captured in the model. Furthermore, it is susceptible to two types of attacks, namely *mimicry* [31] and *impossible path execution* (IPE). Newer algorithms have since been developed to address these drawbacks, primarily by associating additional context with each system call. Reference [26] uses the location of system call invocation as the calling context, while References [4] and [7] can potentially use the entire list of return addresses on the call stack at the point of system-call invocation. Techniques have also been developed that rely on static analysis for building models [30, 8], as opposed to learning.

These newer models make mimicry and IPE attacks harder, but they still remain possible. In particular, the use of calling contexts make mimicry attacks difficult: although an attacker may be able to make one system call using an exploit, the attack code will not be able to resume control after the execution of this system call. This is because the IDS will otherwise observe a return address on the stack that resides in attacker-provided code, and hence would not be in the IDS model. Kruegel *et al.* then devised a clever approach [12] that relied on corrupting data items such as saved register contents or local variables in order to reacquire control after making a system call. Moreover, Chen et al [3] demonstrated powerful attacks that don't modify control flows at all — instead, they only change non-control data, yet achieve the same end goals achieved by (the more popular) control-flow hijack attacks.

The above developments focused more research efforts on techniques that incorporate system call argument data into IDS models [13, 29, 1, 17, 18, 15]. Unfortunately, since most of these techniques do not reason about bulk data arguments such as the data read from (or written to) files or the network, they remain vulnerable to a class of mimicry attacks [21]. This attack works on any *I/O-data-oblivious IDS,* i.e., IDS that may possess perfect knowledge about system calls, their sequencing, and their argument values, with the singular exception of data buffer arguments to read and write operations. Since TEAD examines read buffers and write buffers to check for anomalous taint, it is *not* I/O-data-oblivious, and hence this attack is not applicable to TEAD.

Dependency and taint-based techniques. The core idea behind TEAD was described in an abstract in [2]. This paper represents the full development of those core ideas.

Dataflow anomaly detection [1] provides an interesting contrast with TEAD. In particular, dataflow anomaly detection uses learning to infer information flows. For this reason, it focuses on so-called binary relations that capture relationships between the arguments of different system calls. In contrast, TEAD relies on actual information flows present in a program. Since it has access to information flow already, it uses only *unary relations* in the terminology of dataflow anomaly detection, and does not consider relationships between arguments of different system calls. Key benefits of dataflow anomaly detection over TEAD are: (a) it does not require access to source code, and (b) it has much lower overheads. On the other hand, taint-tracking is much more reliable as compared to dataflow inference, which should lead to a lower false positive rate for TEAD.

Whereas dataflow anomaly detection focuses on security-sensitive data such as file names and file descriptors, dataflow inference [25] is concerned with inferring more general dataflows, e.g., between the data read and written by an application. This necessitates the use of more powerful matching algorithms as compared to the simpler (exact or prefix-matching) algorithms used in dataflow anomaly detection. While dataflow inference can provide low overheads and avoids the need for heavy-weight instrumentation, it is limited to applications that do not perform complex transformations on inputs.

SwitchBlade [5] has some similarity with TEAD in combining taint-tracking with system call anomaly detection. However, the similarity is only superficial since our goals as well as the techniques are quite different. In particular, TEAD uses taint information to improve attack detection and false positives of a typical system call anomaly detector. SwitchBlade's main goal is not one of improving anomaly detection, but instead, to reduce the overhead of taint-based policy enforcement techniques [28, 19, 33]. In particular, they aim to stop control-flow hijacks that are prevented by [28, 19, 33], but do so without the overheads of runtime taint-tracking. They achieve this by using system call anomaly detection as a low-overhead filter to screen out potential exploits from normal behaviors. These potential exploits are then verified by replaying them on a taint-tracked version of the victim process. If a taint policy violation is observed during replay, an attack is reported. Otherwise, the new behavior is added to the IDS model. To reduce the likelihood of missing exploits, SwitchBlade develops new techniques that personalize system-call IDS models to each deployment site, and injects random system calls into the model.

Sarrouy et al [23] also observe that attacks result from tainted data that feeds into system calls. However, their technical approach is quite different from ours. In particular, their approach does not rely on system call models, but instead, captures invariants involving a program's internal state that may hold at various points during execution.

Ming et al [16] are concerned with the problem of improving the accuracy of so-called gray-box anomaly detectors that focus on data [13, 29, 1, 14]. In particular, these techniques may end up learning properties that were observed during training but do not necessarily hold in general. Ming et al show that taint-tracking can resolve such questions, and establish whether training observations truly support the rules learned by an anomaly detector. In contrast, our work shows that by leveraging taint information, we can extend the class of attacks that can be reliably detected by an anomaly detector.

In summary, although numerous works have studied learning-based anomaly detection and information-flow tracking, none of them have considered our approach of augmenting anomaly detection models with taint data in order to reliably detect non-control data attacks that have been challenging for all previous intrusion detection techniques.

6 Conclusion

In this paper, we presented a new approach which combines fine-grained taint-tracking and learning-based anomaly detection techniques. By exploiting the information provided by the taint-tracking component, our approach was able to detect a variety of non-control data attacks that have proved to be challenging for previous intrusion detection or policy enforcement techniques. False positives, one of the main drawbacks of learning-based approaches, are caused due to the fact that training can never be exhaustive. Our approach limits this drawback as it considers only tainted traces, which usually are a small percentage of the whole traces executed by an application. Our evaluation results show that the false positive rate of a learning-based approach is reduced by about a factor of ten due to the use of taint-tracking.

References

1. Bhatkar, S., Chaturvedi, A., Sekar, R.: Dataflow anomaly detection. In: IEEE Security and Privacy (2006)
2. Cavallaro, L., Sekar, R.: Anomalous taint detection (extended abstract). In: RAID (2008)
3. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-Control-Data Attacks Are Realistic Threats. In: USENIX Security Symposium (2005)
4. Feng, H., Kolesnikov, O., Fogla, P., Lee, W., Gong, W.: Anomaly Detection using Call Stack Information. IEEE Symposium on Security and Privacy (2003)
5. Fetzer, C., Susskraut, M.: Switchblade: enforcing dynamic personalized system call models. In: EuroSys (2008)
6. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: IEEE Symposium on Security and Privacy (1996)
7. Gao, D., Reiter, M.K., Song, D.: Gray-box extraction of execution graphs for anomaly detection. In: ACM CCS (October 2004)
8. Giffin, J.T., Jha, S., Miller, B.P.: Efficient context-sensitive intrusion detection. In: NDSS (2004)
9. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion Detection Using Sequences of System Calls. Journal of Computer Security (1998)
10. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security (JCS) 6(3), 151–180 (1998)

11. Kong, J., Zou, C.C., Zhou, H.: Improving Software Security via Runtime Instruction-level Taint Checking. In: Workshop on Architectural and System Support for Improving Software Dependability (2006)
12. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating Mimicry Attacks Using Static Binary Analysis. In: USENIX Security Symposium (2005)
13. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: ESORICS. Gjøvik, Norway (October 2003)
14. Li, P., Park, H., Gao, D., Fu, J.: Bridging the gap between data-flow and control-flow analysis for anomaly detection. In: Annual Computer Security Applications Conference (2008)
15. Liu, A., Jiang, X., Jin, J., Mao, F., Chen, J.: Enhancing System-Called-Based Intrusion Detection with Protocol Context. In: IARIA SECURWARE (August 2011)
16. Ming, J., Zhang, H., Gao, D.: Towards Ground Truthing Observations in Gray-Box Anomaly Detection. In: International Conference on Network and System Security (2011)
17. Mutz, D., Valeur, F., Kruegel, C., Vigna, G.: Anomalous System Call Detection. ACM Transactions on Information and System Security 9(1), 61–93 (February 2006)
18. Mutz, D., Robertson, W., Vigna, G., Kemmerer, R.: Exploiting Execution Context for the Detection of Anomalous System Calls. In: RAID (2007)
19. Newsome, J., Song, D.X.: Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In: NDSS (2005)
20. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: Automatically hardening web applications using precise tainting (2005)
21. Parampalli, C., Sekar, R., Johnson, R.: A practical mimicry attack against powerful system-call monitors. In: AsiaCCS (2008)
22. Pietraszek, T., Berghe, C.V.: Defending against injection attacks through context-sensitive string evaluation. In: RAID (2005)
23. Sarrouy, O., Totel, E., Jouga, B.: Building an Application Data Behavior Model for Intrusion Detection. In: IFIP Data and Applications Security (2009)
24. Saxena, P., Sekar, R., Puranik, V.: Efficient fine-grained binary instrumentation with applications to taint-tracking. In: CGO (April 2008)
25. Sekar, R.: An efficient black-box technique for defeating web application attacks. NDSS (2009)
26. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In: IEEE Symposium on Security and Privacy (2001)
27. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: POPL (2006)
28. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure Program Execution via Dynamic Information Flow Tracking. In: ASPLOS (2004)
29. Tandon, G., Chan, P.: Learning rules from system call arguments and sequences for anomaly detection. In: ICDM Workshop on Data Mining for Computer Security (2003)
30. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In: IEEE Symposium on Security and Privacy (2001)
31. Wagner, D., Soto, P.: Mimicry Attacks on Host Based Intrusion Detection Systems. ACM CCS (2002)
32. Wespi, A., Dacier, M., Debar, H.: Intrusion detection using variable-length audit trail patterns. In: RAID (October 2000)
33. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced Policy Enforcement: a Practical Approach to Defeat a Wide Range of Attacks. In: USENIX Security Symposium (2006)