Anomalous Taint Detection

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

[1] Department of Computer Science
University of California at Santa Barbara, USA
<sullivan@cs.ucsb.edu>[⋆]
[2] Department of Computer Science
Stony Brook University, USA
<sekar@seclab.cs.sunysb.edu>

Abstract. Software security has become an increasing necessity for guaranteeing, as much as possible, the correctness of computer systems. A number of techniques have been developed over the past two decades to mitigate software vulnerabilities. Learning-based anomaly detection techniques have been pursued for many years due to their ability to detect a broad range of attacks, including novel attacks. More recently, taint-tracking techniques (also known as information-flow techniques) have become popular due to their high accuracy and the ability to detect a broad range of attacks.
We believe that the discriminating power of anomaly detectors can be improved by combining them with fine-grained taint analysis. To this end, we propose *anomalous taint detection*, an approach which couples taint analysis and learning-based anomaly detection approaches to automatically infer taint-enhanced security policies, while keeping false positives low and increasing the accuracy of the underlying models. The intuitive justification for this is that an attack involves a combination of a vulnerability, and an attackers ability to exercise this vulnerability. Anomaly detection techniques detect behavioral deviations that occur when a vulnerability (targeted by an attack) is exercised. Fine-grained taint information provides clues about the ability of the attacker to exercise this vulnerability.
We developed a prototype implementation of our approach which showed that is effective to provide protection from data attacks as well as memory errors which corrupt code pointers. False positives rate are discussed as well.

1 Introduction

A number of techniques have been developed over the past two decades to mitigate software vulnerabilities. Learning-based anomaly detection techniques [37,35,9,15,29,21,20,2] have been pursued for many years due to their ability to detect a broad range of attacks, including novel attacks. More recently, taint-tracking techniques (also known as information-flow techniques) [30,23,5,26,24,38] have become popular due to their high accuracy and the ability to detect a broad range of attacks.

It can be observed that an attack involves a combination of a vulnerability, and an attackers ability to exercise this vulnerability. Anomaly detection techniques detect behavioral deviations that occur when a vulnerability (targeted by an attack) is exercised. Fine-grained taint information, on the other hand, can provide information about the ability of the attacker to exercise this vulnerability, significantly increasing the odds that an attack is in progress. One of the advantages of anomaly detection or,

[⋆] This work has been carried out while the author was a PhD student from Università degli Studi di Milano, Italy visiting the Department of Computer Science of SUNY at Stony Brook, USA.

to be more precise of dynamic learning-based techniques, is that they are able to automatically infer security policies by observing and characterizing applications' events, as successfully shown in recent and past research efforts [37,35,9,15,29,21,20,2]. For instance, sequences of system calls can be used as a starting point to define a possible behavior of an application [15], statistical models can help to characterize system calls arguments usage [21,20], and machine learning techniques can be adopted to infer relationships among the arguments of different system calls [2]. Unfortunately, all these techniques suffer of false positives (FPs) and attempts to limit them generally increase false negatives (FNs) as well. Moreover, by building profiles of legitimate behaviors, anomaly detection approaches are generally victim of mimicry attacks [36,35,17,33,32], where an attacker tries to stick his attack to be compliant to the inferred profile.

Although we do not attempt to solve all the dynamic learning issues, we believe its drawbacks can be mitigated by enhancing anomaly-based techniques with information provided by taint analysis. More precisely, by combining taint analysis and anomaly detection, only *tainted traces* – that is events such as system calls that have at least one tainted argument – which are normally a small percentage of the total number of the observed events, are further characterized. This provides two main benefits. First, FPs can be lowered and model accuracy increased, which, in turn, decreases FNs as taint is a property of data but it does not depend on the observed data itself. For instance, an untainted system call argument a observed during a learning phase will most likely remain untainted during a normal detection run, regardless of the data characterization actually observed for a. An anomaly would occur if a became tainted during detection as this would be considered as a manifestation of an attack. The second benefit is that the approach focuses its analysis on tainted events. Therefore, untainted events are not characterized anymore and thus, for these events, the incompleteness of dynamic learning approaches with respect to observed data and exercised paths does not represent an issue anymore. In fact, any unseen/unknown untainted events will not be considered during detection phase further constraining FPs.

Unfortunately, as protection mechanisms improve, so do the attacks. For instance, recently Chen *et al.* [6] pointed out that is no longer necessary to exploit a memory error vulnerability with the goal to hijack a legal process' control-flow to cause harm. In fact, damage can also be caused by overwriting arbitrary security sensitive data and data pointers. Even if this seems to be a strict restriction, Chen *et al.* showed that these attacks can be as powerful as the classic ones (i.e., which corrupt code pointers).

Motivated by these observations, in this paper we propose an approach to provide a more comprehensive protection against data attacks. In particular, we make the following contributions:

1. We propose *anomalous taint detection*, a technique which combines taint and anomaly detection-based approaches. As aforementioned, this permits to automatically infer a process behaviors by considering only tainted traces. Thus, FPs can be limited and attacks detection accuracy and resistance to mimicry attacks improved.
2. We developed a prototype implementation of the proposed approach which transforms a program P to P_T, a semantically-equivalent taint-enhanced version of it. Then, by leveraging on taint information, P_T is further enhanced to perform (i) an attack-free training phase (as in every learning-based anomaly detection approach) where properties of *tainted* sinks' arguments, that is, relevant events (e.g., system calls or security sensitive functions) are learnt and modeled to generate a behavioral profile $\mathcal{M}$ of P_T, and (ii) a detection phase during which single events of P_T are observed at run-time and checked one at a time to see whether they are consistent to the learnt behavioral profile $\mathcal{M}$. Should $\mathcal{M}$ be inconsistent with respect to these traces, an alarm will be raised.
3. We evaluated our approach in terms of effectiveness in detection accuracy of data attacks, FPs rate, and resistance to some form of mimicry attacks [36,35,17,33,32].

The rest of the paper is organized as follows. We describe how *anomalous taint detection*, the approach proposed in this paper, provides a more comprehensive protection from data corruption attacks [6] in § 2. Implementation overview is provided by § 3, while evaluation of the proposed approach in terms of effectiveness, and false positives rate is given in § 4. Discussion on the approach is provided by § 5, and related literature is described in § 6. We conclude the paper with § 7 which provides future direction as well.

2 Anomalous Taint Detection

An important benefit of taint analysis approach is that it can accurately detect many classes of attacks *without* requiring application-specific policy development. In fact, the enforced policies are generic and easy to specify. However, to provide more comprehensive protection against other general data attacks [6] or unknown types of attacks, it becomes necessary to develop application-specific policies that can tightly constrain the behavior of the protected application. Development of such policies can be time-consuming. Moreover, it could be hard to describe manually, what a policy should look like. On the other end, anomaly detection techniques have had an advantage in this regard: they do not require manual effort for developing behavior profiles; instead, profiles are automatically learnt using training data that is acquired during normal operation of an application.

The drawback of anomaly detection techniques is that in practice, they suffer from a high rate of false positives (FPs). This is because of the fact that training can never be exhaustive, and hence some unseen (but legitimate) behaviors will be classified as attacks. We believe that the discriminating power of anomaly detectors can be improved by combining them with fine-grained taint analysis. The intuitive justification for this is as follows. Note that an attack involves a combination of a vulnerability, and an attackers ability to exercise this vulnerability. Anomaly detection techniques detect behavioral deviations that occur when a vulnerability (targeted by an attack) is exercised. Now, fine-grained taint information can provide information about the ability of the attacker to exercise this vulnerability. More concretely, consider a system that detects anomalous system call arguments. Such a system may detect an anomalous argument to an `execve` system call, which raises a suspicion. If, in addition, this argument is tainted, then it significantly increases the odds that an attack is in progress. Based on this observation, we propose a mixed approach which combines anomaly detection and fine-grained taint-tracking.

Since taint is a property of data, our proposed approach will be focused on learning properties of system call arguments rather than their names. Let Σ be the set of all the sinks, and $s(a_1, a_2, \cdots, a_n) \in \Sigma$ a generic sink, where $a_1, a_2, \cdots, a_n$ are sink's arguments. As mentioned earlier, our approach uses taint analysis and anomaly detection using a learning-based approach to learn taint information of sinks' arguments. For instance, our model considers all the system calls and some function of interest (e.g., `print`-like functions used in format string attacks) as sinks.

As other approaches [2,29,21], our analysis is *context-sensitive*. That is, it considers contexts for each system call that can be utilized to refine argument learning. For instance, our approach can distinguish between `open` system calls made from two different locations, and can thus learn different properties for the arguments of the two calls. This increases accuracy of the model if, say, one of the `open`'s is used to open a configuration file, while the other is used to open a data file specified by a user. In fact, intuitively, the former system call uses an *untainted* file name argument, while the latter uses a *tainted* one. This improves the likelihood of detecting an attack, should the untainted file name argument be marked tainted during the detection of our approach.

As other anomaly-based approaches, our strategy roughly consists of two phases, namely a *learning* phase, and a *detection* one. During the learning phase our approach builds a profile $\mathcal{M}$ of a monitored application, based on the information we describe below. Afterwards, when the learning phase is terminated, the application is executed in detection mode and a new profile $\mathcal{M}^{'}$ is created incrementally. Should $\mathcal{M}$ deviate from $\mathcal{M}^{'}$, an alarm will be raised.

In the following, we describe what kind of taint information is learnt by our strategy, and how this can be used to detect memory error exploits.

Using Coarse-grain Taint Information

Taint information associated to a sink argument a_i are learnt. For aggregate data such as `struct`'s and arrays, the taint status of all bytes of the data will be combined into one. Multiple taint values, e.g., corresponding to data dependencies versus control dependencies, are learnt separately.

At detection time, an alarm can be 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. Examples of documented attacks that can be detected by this extension are (a) an attack on the popular `WU-FTPD` that corrupts a *userid* argument to a `seteuid` system call [6], and (b) an attack on `Netkit` telnet server that overwrites the name of a login program, which is subsequently used as an argument to `execve` system call [6] (see § 4.1).

Using Fine-grained Taint Information

For some aggregate data, the above approach may loose too much information by combining the taint values associated with the data. To improve precision, we can avoid this combination step. For instance, we can individually learn taint information associated with each field of a `struct`. This is particularly useful for some system calls, e.g., `recvmsg`, `sendmsg`, `readv`, `writev`, where a more detailed understanding of the involved data structure is required, in order to gather more meaningful taint-provided information). For arrays, user may select specific array elements for which taint information needs to tracked individually. Currently, our proof of concept implementation does not exploit this last feature.

Deriving Application-specific Taint-enhanced Policies

Policy-based and anomaly-based detection techniques possess complementary benefits: the main benefit of policy-based detection is a low rate of false positives, while their drawback is the effort required for policy development. In contrast, anomaly detection requires no such effort, but in practice, tends to suffer from a higher rate of false positives. Our strategy combines the strengths of the two approaches by using models built for taint-based anomaly detection to suggest (taint-enhanced) security policies.

Learning whether a sink argument a_i is tainted or not already improves the accuracy of our approach and, as aforementioned, allows the detection of some data corruption attacks which corrupt data pointers (see for instance `WU-FTPD`, and `Netkit` in § 4.1). However, a better characterization of the argument considered is needed when more general memory error attacks are involved (e.g., [21]). To this end, for every sink argument a_i, the adopted learning-rules characterize the following properties depending on whether a sink argument a_i is *fully* or *partially* tainted.

(a) a_i is *fully* tainted. That is, each byte of a_i is tainted. The following argument's properties are inferred by the underlying learning-rules:

Maximum length. Since the argument is tainted, this property inferred during training phase gives an approximation of its probable maximum length l_{max} which is expected during detection phase. This helps to detect memory error attacks that try to overflow buffers with the intent to overwrite security sensitive data used at sinks (see [21] for instance). In fact, during detection these tainted arguments will exhibit a length $l > l_{max}$ which, as a direct consequence, will violate the inferred security policy.

Structural inference. There are situations where characterizing arguments' lengths is not enough. An attacker might not try to overflow any buffers but, instead, he might try to modify the normal structure of the considered 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`, and numbers (0-9) to `0`. 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 learnt, an alarm will be raised.

It can be noted that for particular sinks, trying to infer their (tainted) argument structure can rise FPs if the structure for that sink is highly unpredictable during learning (i.e., it keeps changing frequently). For instance, when arbitrary data D are read from the network, they are marked as tainted as network input is considered untrusted. Let suppose that D is subjected to some application-specific transformation (e.g., encoding/decoding) or application-specific sanity/security check, subsequently. Let $D^{'}$ be the transformed data. When $D^{'}$ reaches other sinks, such as output sinks (e.g., `open`, `stat`, `execve`) its structure will be either different from the initial one (e.g., encoding/decoding), or its structure will have a "fixed shape" (e.g., sanity/security check) when that particular output sink is reached. Therefore, to try to constrain FPs due to an incorrect characterization of the analyzed argument, and to make the learning phase less data-dependent, it makes sense to enable only the learning of the argument maximum length for particular sinks (e.g., `gets` at proper context, for example [21])[1].

(b) a_i is *partially* tainted. That is, a_i has both 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 (a_i can have different tainted/untainted portions and all of them have to be considered. However, our current implementation considers only the first pair):

Minimum length. When an untainted portion of a tainted argument is considered, what is important to remember is its minimum length l_{min}. In fact, considering its maximum length would be misleading as the argument would likely not have the same length all the time and, as the argument portion is untainted (i.e., trusted), we are more concerned on the fact that the argument will *always* have a minimum number of untainted bytes. Intuitively, this is an indication that the attacker will not be able to overwrite the whole untainted argument portion. In most cases the attacker will not be able to overwrite not even a byte of the untainted portion as usually, l_{min} will be identical to l_{max} (e.g., sinks arguments which operate on the same untainted data). However, for those situations where this is not true, l_{min} provides a lower bound under which is not possible to underflow without raising an alarm.

[1] Although argument length depends on data, it is not data-dependent in the sense that it does not depend on any particular data value.

Longest common prefix (LCP). Untainted arguments (or portion of them) should have a more regular "structure" or shape than the tainted counterpart as they are not directly influenced by user input. Therefore, our approach learns the longest common prefix for every considered sink argument which is partially untainted. Should the learnt longest common prefix be different during detection, an alarm would be raised.

Further discussion about attacks on the proposed approach as well as arguments on the protection provided are given in § 5.

3 Implementation

As a first step, the approach proposed in this section takes a program P as input and produces P_T, a semantically-equivalent taint-enhanced version of it. In particular,

- For *every* taint source and sink, that is for every system call or function of interest invoked by P_T, a wrapper $\mathcal{W}$ is introduced by the transformation approach. This enables our approach to (i) learn properties of sinks' arguments, and (ii) mark some source arguments as tainted (e.g., those coming from the network). Following this reasoning, it is possible to automatically infer and enforce taint-enhanced security policies for every taint source and sink by using the information and learning rules described in § 2.
- Tracking of control dependency are fully enabled to be able to further enhance the inferred policies not only with taint information but also with information on *how* taint information is propagated throughout the application lifetime. This information can help to further thwart memory error attacks which corrupt arbitrary non-pointer data and represents an on-going research area we are exploring. Currently, the implementation of this feature in our proof of concept is not fully complete.

In the following we detail the aforementioned steps, the building blocks of anomalous taint detection.

1. P_T is monitored during the *training* phase and a log file is created. The log file includes sink's names and their context information (e.g., calling site), sink's arguments and, for each argument, taint information as well as further characterization, if needed, by using the model described in § 2. 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 considered. For instance, the entry for the second argument (`arg2`) tells us that the address (`A`) where the sink buffer of size 99 (`V[0-98]`) is stored at is untainted (`A:U`), while the buffer content is tainted (`V[0-98]:T`). Moreover, the content of the tainted buffer which starts at offset 0 is `ls -la`[2]. These information will be used by the next step.
   The learning phase is implemented by using a dynamic shared object transparently loaded into P_T address space which wraps and overrides the original sinks behavior, re-invoking them whenever necessary.

[2] 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.

2. The log file is analyzed *off-line* to build a profile $\mathcal{M}$ of the behavior of P_T by using the information provided by the previous step. 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 as part of $\mathcal{M}$ but no further information is gathered or subsequently considered for them (this is basically done for evaluation reason).
   For instance, considering the previous example, the tainted sink `read` invoked at the calling site `0x8048f5c` has the 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, accordingly to the learning rules described in § 2, its structure is `a  -a`.
   The profile created during this step is serialized and re-loaded during the next step. This permits to update the profile whenever there is the need to do so, without discarding the out-of-dated one.
3. P_T is monitored during a *detection* phase. An *on-the-fly* behavioral profile $\mathcal{M}^{'}$ of P_T is incrementally created. The information contained in $\mathcal{M}^{'}$ are consistent with the one considered in 1. Should $\mathcal{M}^{'}$ be inconsistent with $\mathcal{M}$ an alarm will be raised.
   The detection phase is implemented by using a dynamic shared object transparently loaded into P_T address space which wraps and overrides the original sinks behavior, re-invoking them whenever necessary.

The anomaly detection part of our proof of concept has been implemented by using the C, C++ and Python programming languages with approximately $15,000$ lines of code. The program transformation prototype which is in charge to taint-enhance a given program P, instead, leverages on the implementation developed in [38] which has been modified accordingly, where necessary, to suite our needs.

4 Evaluation

In the following, we present the evaluation of our anomalous taint detection approach. We evaluate our approach in terms of (i) effectiveness in detecting memory corruption attacks which target arbitrary data and data pointers [6], and (ii) false positives rate.

4.1 Effectiveness

It can first be observed that taint analysis by itself provides enough protection for memory error attacks which corrupt security sensitive code pointers, such as function return addresses saved on the stack or function pointers. Thus, it is possible to avoid considering these memory corruptions thanks to the taint analysis component adopted by our approach[3]. Therefore, in the following we consider several different examples of memory error attacks which corrupt data and data pointers. Moreover, where necessary, we slightly modify the example to show that our approach is effective regardless of the kind of memory error vulnerability (e.g., buffer overflow, format string, integer overflow) considered. For simplicity, we reproduced the vulnerable code snippet of vulnerable programs, as described next, and we verified that our approach detects the memory corruption attacks performed on these code snippet.

[3] Indeed, we could relax this restriction and enhancing our learning rules to consider whether a code pointer is allowed to be tainted or not. This is not the case generally, but when direct control dependencies are fully tracked, there might be the situation where a code pointer is marked as tainted based on control dependency taint propagation.

Format String Attack against User Identity Data A version of `WU-FTPD` is vulnerable to a format string vulnerability in the `SITE EXEC` command [6]. The attack proposed by Chen *et al.* aimed to keep the process' privilege level as high as possible (i.e., `root` privileges). In this way, a regular authenticated user could upload a custom `/etc/passwd` file which, subsequently, allowed him to log in as `root`.

By considering the following code snippet, part of the function `getdatasock`, it is clear that this goal can be achieved by overwriting the `pw->pw_uid` field which contains the cached credential of the current authenticated user[4].

```
1  FILE *getdatasock(...) {
2     ...
3     seteuid(0);
4     setsockopt(...);
5     ...
6     seteuid(pw->pw_uid);
7     ...
8  }
```

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. While the latter method relies on the presence of a particular memory error vulnerability, the former detects corruptions caused by untrusted input. In particular, our approach learns that the `seteuid` argument `pw->pw_uid` at line 6 is always *untainted* during the learning attack-free phase[5]. On normal situation, this also holds during detection phase, where no attacks are conveyed. On the other end, any corruption of the user credential represented by `pw->pw_uid` will affect the taintedness value of the `seteuid`'s argument of line 6, thus disclosing the attack attempt. It is worth pointing out that, in this context, our anomaly detection component leverages on the information provided by the underlying taint-tracking mechanism and not on the data actually encountered and characterized during the learning phase. As a result, our approach is less subjected to mimicry-like attacks than, for instance, learning-based anomaly detection approaches which rely *only* on statistical properties of the observed data. Depending on the context, an approach as the one proposed in [21], for instance, would likely exhibit FPs, or be more vulnerable to mimicry-like attacks. In fact, a learning phase which observes a limited number of authenticated users would open the possibility for an attacker to impersonate one of the observed legitimate users. On the other end, a more conservative learning phase which considers only one authenticated user would most likely constrain the application usability (for instance, the `root` user can never be used by the approach proposed in [21] neither during training nor during detection, otherwise the proposed attack would be effective).

[4] Although, it is not completely clear whether the attack proposed in [6] properly works or not (the used injection vector would have the effect to write 10 in the `pw->pw_uid` field, in the best case (for the attacker)), it should be possible to achieve what is claimed in the paper anyway, even if requires a slightly more complicated attack pattern.

[5] Even if `pw->pw_uid` is derived from user input, the result of this system call is marked as untainted. Moreover, also the result of the function `getpwnam`, which would likely be used to obtain information on the credential of a user, is marked as untainted. This holds for several other system calls/functions of interest, but not for others (e.g., the number of bytes read by `read` is marked as tainted as it can be used to influence loops or similar actions.

Heap Corruption Attacks against Configuration Data In the following we report two heap-based memory error vulnerabilities and attacks as described by Chen *et al.* in [6].

Null HTTPD The proposed attack exploits a heap-based buffer overflow vulnerability. It aims to overwrite the `CGI-BIN` configuration string prefix to change it from the value `/usr/local/httpd/cgi-bin` (default value) to the string `/bin`. Every CGI script/program invoked by the client will be searched in this new CGI directory, allowing an attacker to easily invoke the shell interpreter, for instance.

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) partial overwrite the configuration string. In this latter case, the goal could simply be to execute commands in a sub-directory of the original `CGI-BIN` or, alternatively, perform a directory traversal to reach the intended directory. Depending on the attacker choices, it is possible to observe different scenarios. For simplicity, let us consider that the sink of interest here is the `open` system call.

(a) During the training step our approach would learn that the first argument of the `open` system call invoked at a context $\mathcal{C}$ is possibly a combination of *untainted* data (i.e., the original `CGI-BIN` configuration string) and *tainted* one (i.e., the command derived from untrusted input), if any.
When an attempt to exploit the memory error is made, it is clear that during the detection phase the `open`'s first argument has no *untainted* component. Thereby, our system will raise an alarm for the anomalous event as, in this case, both the minimum untainted string length and the longest common prefix policies are violated.

(b) This case is similar to the previous one because even if we still do have some *untainted* data, the length l of these data observed during detection phase is less then the one learnt during the training phase (with $l_{min} > 0$).
However, it is worth noting that l_{min} could be less or equal to l (e.g., the `open` system call invoked at context $\mathcal{C}$ operates on different untainted strings). To be successful, an attack should perform a directory traversal attack in order to backward-traverse the original directory while keeping the length of the *untainted* data consistent to what has been learnt. Since a directory traversal attack exhibit clear patterns, and the injected bytes are tainted as they come from an untrusted source, our structural inference learning rule would discover the divergence with the structure learnt during the training phase (unless, of course, such a pattern would have been learnt, at the same context, for the considered sink and argument, during such a step).

Netkit Telnetd The attack proposed in [6] 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");
   execv(loginprg, argv);
}
```

As a result of a successful attack, the application invokes the program interpreter `/bin/`$\underline{\texttt{sh}}$ `-h` $\underline{\texttt{-p}}$ `-p` (underlined characters are tainted). Our approach detects this attack in a similar way as it detects the attack launched on `Null HTTPD` described above.

Stack Buffer Overflow Attack against User Input Data The exploitation of this stack-based buffer overflow vulnerability was somewhat tricky but the authors of [6] 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 TOCTOU). As in the attack previously reported, also this memory error exploit is detected in a similar way. In fact, if the tainted argument does not contain attack patterns during the training phase (e.g., directory traversal `../` pattern), an attack attempt during detection phase will present a different structure from the one previously observed.

Straight Overflow on Tainted Data The following example has been recently proposed by Mutz *et al.* in [21]. 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 affirmative case, the program prints an error message and quits. Otherwise (i.e., non privileged file), more data is read into the array `user_data`, through 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) {
17        fprintf(fp, "%s", user_data);
18        fclose(fp);
19      }
20    }
21 }
```

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. It is possible to infer the structures of their arguments as well, but due to the nature of the program, this might raise too many FPs. Of course, using l_{max} by itself could raise FPs as well, as it highly depends on the accuracy of the learning step. Nonetheless, this does not depend on the value of the observed data, and therefore on the precision of the underlying statistical models.

Format Bug to Bypass Authentication The following example has been proposed by Kong *et al.* in [16]. Even in this example, the memory error attack is simple. 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, 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)
13       access_granted();
14 }
```

The proposed 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 learnt during the learning phase, than no tainted formatting directives can be subsequently encountered during detection phase without raising an alarm. On the contrary, the approach checks whether tainted formatting directives encountered during detection phase are consistent with the ones learnt during the training phase. Since the training phase should be attack-free, no dangerous and tainted patterns should have been learnt. Thus, it should be hard to mimicry those patterns in order to successfully exploit the format string vulnerability, as this require the use of tainted `%` format string directives (required both to corrupt memory and to leak information).

Untainted Format String Attacks As pointed out in [7] it is possible to exploit format string vulnerabilities in such a way to being able of writing untainted data to untainted memory locations. Even if this attack might be hard to perform in a real setup, our anomalous taint detection provides protection against it, by using the learning rules defined for tainted sinks' arguments (in a format string vulnerability, the format string is still, however, tainted even when the address to write to is not).

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. It is possible to note that the overall FPs rate for these applications are in the order of 10^{-4} and 10^{-3}, respectively. Table 2, instead, highlights what models were responsible for the overall false positives rate obtained.

As shown by Table 2, the majority of false positives were caused by violation of the LCP and structural inference models (number of false positives and their rate in parenthesis). After a quick examine, LCP violations can be ignored as it turned out they were caused by a small bug in our prototype implementation (a comparison which compared one less character during detection than the one learnt during the learning phase). As for structural inference, we expected such a high number false positives as the model proposed in § 2 is a simple non-probabilistic model. In fact, the main goal

#	App	# Traces (Learning)	# Traces (Detection)	FP	Overall FP rate
1	`proftpd`	$68,851$	$983,740$	200	2.0×10^{-4}
2	`apache`	$58,868$	$688,100$	2000	2.9×10^{-3}

Table 1. Overall False Positives.

of our anomalous taint detection approach is to couple taint analysis with anomaly detection so that each technique can improve its drawbacks by exploiting the advantages provided by the other. The approach, in fact, is general enough so that more powerful learning-rules, such as the one proposed in literature [20,2], can be plugged into it. Therefore, if we consider only FPs caused by violation to the taintedness of sinks' arguments, it is easy to see that this policy performed fairly well for `ProFTPD` (3×10^{-5} FPs rate), while it performed well for `Apache`, by not reporting FPs at all.

#	App	Taint	LCP	Min	Struct Inf.	T. Max	Overall FPs
1	`proftpd`	30 (3.0×10^{-5})	30 (3.0×10^{-5})	0 (0)	140 (1.4×10^{-4})	0 (0)	200 (2.0×10^{-4})
2	`apache`	0 (0)	300 (4.3×10^{-4})	0 (0)	1700 (2.4×10^{-3})	0 (0)	2000 (2.9×10^{-3})

Table 2. False Positives Breakdown.

Table 3 shows how untainted unknown traces, along with FPs caused by violation of the taintedness of sinks' arguments, would have raised the FPs rate if they would have been considered (i.e., if taint information would not have been considered). FPs for `ProFTPD` would have been increased of one order of magnitude, while the one for `Apache` would have been increased from 0 to the order of 10^{-4}.

#	App	Unknown untainted traces	Taintedness of sinks args	FPs (taint information)
1	`proftpd`	210 (2.1×10^{-4})	30 (3.0×10^{-5})	240 (2.4×10^{-4})
2	`apache`	300 (4.3×10^{-4})	0 (0)	300 (4.3×10^{-4})

Table 3. Unknown/Untainted Traces.

Finally, the following table shows the percentage of tainted events the have been encountered during the learning and detection phase. As the table depicts, half of the traces of `Apache` have been considered during detection, while only a small fraction of them have been characterized for `ProFTPD`. No characterization has to be done for the remaining traces thus avoiding any alarm to be raised due to imprecision of the models generated by the underlying learning-rules.

#	App	# Traces (Learning)	# Tainted (%)	# Traces (Detection)	# Tainted (%)
1	`proftpd`	$68,851$	$2,986$ (4.3%)	$983,740$	$7,120$ (0.72%)
2	`apache`	$58,868$	$46,059$ (82.1%)	$688,100$	$35,470$ (51.5%)

5 Discussion

The taint-tracking mechanism of any taint-based analysis provides *deterministic* protection when a memory error exploit corrupts a *code* pointer (absolute or partial overwrite). In fact, code pointers are usually initialized and manipulated by application code (e.g., function return addresses), which is considered to be *trusted*[6], therefore untainted. As a direct consequence, mimicry attacks [36,35,17,33,32] which rely on hijacking the execution flow of the vulnerable process to either invoke in-trace system calls (or, more generally, sinks), or to corrupt security sensitive data by executing foreign code are no longer possible.

Unfortunately, while this class of arbitrary code execution and mimicry attacks are defeated, others could still be possible. More precisely, mimicry attacks can target sinks' arguments, or generically tamper the process address space with the intent to corrupt security sensitive data. In fact, learning-based rules are used to automatically infer a security policy to be enforced on system calls or functions of interest (i.e., sinks). Unfortunately, sometimes these rules could be either over permissive or over restrictive. False negatives (FNs) and false positives (FPs) are, respectively, the consequence of this characterization. Following this reasoning, it is possible to distinguish these cases:

Untainted sinks arguments. Taint information is used by the whole approach to infer security policies to be enforced on sinks during detection phase. This already gives a better process behavior characterization than compared to previous models (see [37,35,9,15,29,21,20,2]), especially when the rules learn that particular sinks' arguments are untainted (see § 4). Generally, previous models had focused their attention on *every* event of the monitored process to better characterize its behavior. Of course, over simplified models carry minimal information and are more likely to be defeated by mimicry-like attacks. Likewise, over specialized events characterizations as well as unknown unseen events encountered during detection phase, would lead to high false positives rates, as already described in § 1. In our approach, instead, a sink argument a found to be *untainted* during training phase, *must* be untainted during detection as well. Therefore, a large number of mimicry attacks that aim to tamper with untainted data are no longer possible, as attack-provided data or, more generally, input data are always considered untrusted and thus marked as tainted.
Moreover, untainted events, that is sinks whose arguments are untainted, encountered during detection phase are not considered as attacks' manifestations, therefore lowering the overall false positives (FPs) rate.
As described in § 2, things change a little when a combination of tainted and untainted arguments are characterized. In fact, the untainted part is characterized by using the minimum length, and longest common prefix models. These models are highly dependant on the value of the data seen during training. However, these data are untainted, therefore (i) they cannot be modified by an attacker without raising an alarm (in our threat model, network inputs – and thus attacker-provided inputs as well – are always marked as tainted), and (ii) depending on the considered sink, they should be more "predictable", and thus easier to characterize which in turn contributes to further constraining FPs rate.

Tainted sinks arguments. Unfortunately, tainted sinks arguments can be controlled by an attacker. Therefore, mimicry attacks are still possible on the models – maximum length, and structural inference – used to characterize these tainted inputs. However, as basic taint analysis low-level policies limit the execution of foreign or already present code which does not logically follow the

[6] As noted elsewhere, it is possible to relax this requirement as function pointers might be initialized based on tainted control dependency conditions. A conservative approach is to permit the code pointer to be either untainted or tainted due to control dependency taint propagation. In the latter case, an enumerated set of admissible and legal addresses learnt during training is maintained and checked against for consistency during detection. The main drawback is that a "selected" mimicry attack could be executed (instead of an arbitrary one). However, the ability to cause meaningful damage is constrained.

normal execution flow, it is harder for an attacker to stick to the models inferred during training phase. In fact, this extremely relies on the type of memory error vulnerability involved [39] and the position where the vulnerability is located. Nonetheless, a critique of the adopted models in this context follows.

(a) *Maximum length.* As described in § 2, the main purpose of this simple model is to provide an upper bound to the number of bytes considered for a given sink argument. An over permissive model (i.e., upper bound too high) would permit overflows to occur during detection phase. As a direct consequence, variables adjacent to the overflown buffer could be controlled by an attacker potentially missing attacks (e.g., mimicry-like). On the other end, an over restrictive model (i.e., upper bound too low) would wrongly characterize a given sink argument. As a direct consequence, FPs would be more likely to occur.

(b) *Structural inference.* As already noted in related literature [20], there are cases where an attacker is able to craft its input in order to stick to the considered model yet being able to cause harm. As described in § 2, the purpose of the structural inference model is to learn the structure of a given sink argument. While the model herein considered could be more vulnerable to mimicry-like attacks, others (e.g., [20]) are not.

We remark on the fact that some of the considered model are not new (see for instance [20,21], which also propose a better structural inference model). Moreover, it is important to note that the learning rules and models considered herein can be definitely replaced by more accurate models (e.g., temporal relationship among system calls arguments [2], other statistical models [21]). The strategy of coupling taint analysis with anomaly detection offers independent benefits from the underlying learning-rules and models adopted, as already pointed out in § 2 and § 4.

Security sensitive data. The learning-rules adopted by our approach not only consider whether a sink argument is tainted or not, but also keep track of *how* security sensitive data or, more generally, memory locations, have become tainted. In fact, as pointed out in § 2 and showed in § 4.1, taint marks carry different values depending on whether they originate from data or control dependency taint propagation, or a combination of both. As shown in § 4.1, this permits to thwart memory error attacks which target arbitrary non pointers data as well.

Finally, it is worth reminding that the approach proposed in this paper can, sometimes, detect memory error exploits attempts even before reaching a sink argument (e.g., format string and tainted formatting directive). While the approach described in the previous sections is generally vulnerability-independent, it can also be more successful depending on the underlying vulnerability considered (e.g., format string versus buffer overflow). For instance, one of the conditions that must hold to successfully exploit a format string vulnerability [28,12] is that the formatting string has to be controlled by the attacker. This means, that the format string has to be *tainted*. Therefore, our approach provides protection in two ways. First, during learning it signals that a particular formatting string is *tainted*. As there is usually no reason to have a tainted format string, the application could be right fixed (e.g., by substituting `printf(buf)` with `printf("%s", buf)`). Second, if this is not possible, as the learning step has to be performed in an attack-free environment, no dangerous formatting directives should be learnt. Therefore, the inferred structure will not contain any *dangerous tainted* formatting directive (e.g.,`%x` or `%n`).

6 Related Literature

Information flow analysis has been researched for a long time [1,10,8,19,34,22,27]. Early research was focused on multi-level security, where fine-grained analysis was not deemed necessary [1]. More recent work has been focused on language-based approaches, capable of tracking information flow at variable level [25]. Most of these techniques have been based on static analysis, and assume considerable cooperation from developers to provide various annotations, e.g., sensitivity labels for function parameters, endorsement and declassification annotations to eliminate false positives. Moreover, they typically work with simple, high-level languages. In contrast, much of security-critical contemporary software is written in low-level languages like C that use pointers, pointer arithmetic, and so on. As a result, information flow tracking for such software has been primarily based on runtime tracking of explicit flows that take place via assignments.

Recently, several different information flow-based approaches, often known as *taint analysis* as they are concerned with data integrity, have been proposed [23,38,5,16,31]. They give good and promising results when employed to protect benign software from memory errors [23,38], and a broader class of attacks [38] by usually relying, for instance, on some implicit assumptions which are common grounds on benign software (e.g., no tainted code pointers should be de-referenced, no tainted SQL directive should be used). Other researchers (e.g., [16,31]) extended basic taint-tracking techniques in order to generically address attacks which corrupt data and data pointers [6] as well. Preliminary results seem to be promising, even some of them require architectural change [16] and it is still unclear whether they can thwart a broad range of memory error attacks while exhibiting only a limited rate of false positives.

The idea of using syscall obfuscation for preventing computer intrusions has been introduced by [18], where an obfuscation scheme based on the randomization of the system call mappings has been used for dealing with some type of buffer overflows. Following this idea, Forrest *et al.* [11,15] introduced a learning-based anomaly detection strategy in order to characterize the behavior of an application P. 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 a sterile 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 given length are collected during a process runtime, and compared against the contents of the database.

The N-gram model is very simple and very efficient but it is characterized by a relatively high degree of false alarms [13], mainly because *correlations* among syscalls are lost, since there is no provision for storing information about the *position* where the syscalls are invoked. Furthermore, in [35] it has been shown that such a host intrusion detection system (HIDS) is unable to detect two particular forms of computer attacks, namely *mimicry* [36,35,17,33,32] and *impossible path execution* (IPE) [9,35] attacks. Quite recently various authors started to propose variations to the N-gram model in order to improve its "precision", i.e. its ability to correctly detect a computer instrusion, with a particular attention to both the IPE and mimicry attacks. All these models try to overcome the limitations of the original model adopting a better characterization of a program behavior. Such a characterization is obtained by saving for any considered syscall, additional information such as the value of the program counter, the stack configuration, and information regarding the control flow graph (see for instance [29,35,9,14]). However, even these models suffer of some limitations. For example, in [35,9] it has been shown that the callgraph model proposed in [35] as well as the model proposed in [29] are not able to deal with some forms of IPE, while in [36,17] it has been shown that all the models above mentioned are susceptible, with various degrees of resistance, to some forms of mimicry attacks.

In a recent paper, Kruegel *et al.* [17] observed that even if the introduction of such techniques in anomaly-based HIDS [3,9,29] has significantly reduced the possibility to perform successful traditional mimicry attacks [33,32,36], they do not impose any kind of restriction on the execution of arbitrary code which does not directly invoke system calls (i.e., system call-free code). For instance, the *execution* of a piece of code that is able to modify writable memory segments represents a threat by itself. This observation, brought Kruegel *et al.* to devise a variation of the traditional mimicry attack which is able to hijack a program execution flow, execute malicious system call-free code, relinquish the execution flow to the diverted program to regain it later on. This malicious code is usually executed as a preamble of in-trace syscalls. Its main objective is either to change the value of the system call parameters in order to eventually execute arbitrary code, or to modify the value of some control-dependent data variable in order eventually influence the process execution flow. In [17] a proof of concept tool is provided which is able to automatically identify, inside a program, the instructions which can be used for such a scope. More precisely, the main goal of the automatic mimicry is to elude HIDS checks by continuously diverting the process execution flow in order to execute arbitrary code with the purpose of changing system calls parameters without directly invoking any system call. However, most of the time these steps cannot be completed at once. Thus, any piece of malicious code has to take care of continuously regaining the control of the execution flow. Such a task is usually performed by modifying appropriate code pointers. It is worth noting that the taint analysis component of anomalous taint detection approach already limit the execution of foreign code by disallowing to de-reference tainted code pointers.

On the basis of the previous observation (i.e., execution of foreign code), other techniques have been recently proposed for containing automatic mimicry [17]. For instance, [4] proposes a strategy based on the use of static analysis techniques which is able to localize critical regions inside a program, which are segments of code that could be used for exploiting an automatic mimicry attack. Once the critical regions have been recognized, their code is instrumented in such a way that, during the executions of such regions, the integrity of the dangerous code pointers is monitored, and any unauthorized modification will be restored at once with the legal values.

As pointed out in [17] and further reiterated by [6], for instance, it is clear that system call monitoring by itself is no longer sufficient to correctly characterize an application behavior. To this end, researchers proposed statistical models [21,20] which try to characterize system calls *arguments* to improve the precision of the application's behavioral profile. Likewise, data flow relationship between system calls arguments [2] have been recently proposed to address broader classes of attacks (e.g., memory errors, race conditions).

7 Conclusion

In this paper, we presented an approach which combines taint analysis and learning-based anomaly detection techniques. By exploiting the information provided by the taint-tracking component, our approach was able to detect several memory error attacks, including those which corrupt arbitrary data and data pointers. False positives, one of the main drawbacks of learning-based approaches, are caused due to the fact that training can never be exhaustive, and thus some unseen or not characterized (but legitimate) behaviors will be classified as attacks. 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. As taint analysis also provides information about the ability of the attacker to exercise a vulnerability, true positives are improved as well. Moreover, our approach is independent from the underlying learning-rules adopted. In fact, more powerful learning-based approaches can be transparently plugged into our approach without affecting the benefit provided by the taint-tracking component. Moreover, we believe that taint information can also be exploited in order to detect other data corruption attacks, in particular those where control-dependencies are involved.

References

1. D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.
2. Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 48–62, Washington, DC, USA, 2006. IEEE Computer Society.
3. Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. An Efficient Technique for Preventing Mimicry and Impossible Paths Execution Attacks. In *3rd International Workshop on Information Assurance (WIA 2007)*, April 2007.
4. Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Static Analysis on x86 Executable for Preventing Automatic Mimicry Attacks. In *GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2007.
5. Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.
6. Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.
7. M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing Hardware Architectures for Security. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking (held in conjunction with the 33rd International Symposium on Computer Architecture)*, 2006.
8. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
9. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection using Call Stack Information. *IEEE Symposium on Security and Privacy, Oakland, California*, 2003.
10. J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, May 1974.
11. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 120, Washington, DC, USA, 1996. IEEE Computer Society.
12. gera and riq. Advances in format string exploitation. Phrack Magazine, Volume $0xb$, Issue $0x3b$, Phile $\#0x07$ of $0x12$.
13. A. K. Ghosh and A. Schwartzbard. A Study in Using Neural Networks for Anomaly and Misuse Detection. In USENIX *Security Symposium*, 1999.
14. J. T. Giffin, S. Jha, and B. P. Miller. Detecting Manipulated Remote Call Streams. *11th USENIX Security Symposium*, 2002.
15. S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.
16. Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. Improving Software Security via Runtime Instruction-level Taint Checking. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24, New York, NY, USA, 2006. ACM Press.
17. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating Mimicry Attacks Using Static Binary Analysis. In *Proceedings of the USENIX Security Symposium*, Baltimore, MD, August 2005.
18. M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.
19. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.
20. D. Mutz, F. Valeur, C. Kruegel, and G. Vigna. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1):61–93, February 2006.

21. Darren Mutz, William Robertson, Giovanni Vigna, and Richard Kemmerer. Exploiting Execution Context for the Detection of Anomalous System Calls. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, 2007.
22. A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.
23. James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *NDSS*, 2005.
24. A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting, 2005.
25. Perl. Perl taint mode. `http://www.perl.org`.
26. Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection 2005 (RAID)*, 2005.
27. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), January 2003.
28. scut / team teso. Exploiting Format String Vulnerabilities, September 2001. version 1.2.
29. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 144, Washington, DC, USA, 2001. IEEE Computer Society.
30. G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.
31. G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM Press.
32. Kymie M. C. Tan, Kevin S. Killourhy, and Roy A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection*, 2002.
33. Kymie M. C. Tan, John McHugh, and Kevin S. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Information Hiding*, pages 1–17, 2002.
34. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security (JCS)*, 4(3):167–187, 1996.
35. D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy, Oakland, California*, 2001.
36. D. Wagner and P. Soto. Mimicry Attacks on Host Based Intrusion Detection Systems. *In Proc. Ninth ACM Conference on Computer and Communications Security.*, 2002.
37. H. Xu, W. Du, and S. J. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Path s. *RAID LNCS 3224 Springer-Verlag*, pages 21–38, 2004.
38. Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced Policy Enforcement: a Practical Approach to Defeat a Wide Range of Attacks. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.
39. Y Younan, W Joosen, and F Piessens. Code injection in C and C++: A Survey of Vulnerabilities and Countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, Belgium, July 2004.