# Towards Ground Truthing Observations in Gray-Box Anomaly Detection

Jiang Ming
College of Information Sciences & Technology
Pennsylvania State University, USA
mingjiangpku@gmail.com

Haibin Zhang and Debin Gao
School of Information Systems
Singapore Management University, Singpaore
{hbzhang,dbgao}@smu.edu.sg

***Abstract*—Anomaly detection has been attracting interests from researchers due to its advantage of being able to detect zero-day exploits. A gray-box anomaly detector first observes benign executions of a computer program and then extracts reliable rules that govern the normal execution of the program. However, such observations from benign executions are not necessarily *true* evidences supporting the rules learned. For example, the observation that a file descriptor being equal to a socket descriptor should not be considered supporting a rule governing the two values to be the same.**

**Ground truthing such observations is a difficult problem since it is not practical to analyze the semantics of every instruction in every program to be protected. In this paper, we propose using taint analysis to automatically help the ground truthing. Intuitively, the same taint source of two values provides ground truth of the data dependence. We implement a host-based anomaly detector with our proposed taint tracking and evaluate the accuracy of rules learned. Results show that we not only manage to filter out incorrect rules that would otherwise be learned (with high support and confidence), but manage recover good rules that are previously believed to be unreliable. We also present overheads of our system and time needed for training.**

**Keywords: anomaly detection, taint analysis, system call monitor, ground truthing**

## I. Introduction

Anomaly-based detectors construct a model of normal behaviors of a program and detect deviations from such a model. Unlike signature-based detectors, anomaly detectors could detect zero-day exploits since the vulnerable program will behave differently even if the exploit is new and unknown. Many anomaly detectors have been proposed to monitor system calls of a program. Some of them focus on control flow information [1], [2], [3], [4], [5], [6], [7], [8] while others investigate how data flows among arguments and returns of system calls [9], [10], [11], [12], or a combination of them [13].

Unlike white-box anomaly detectors which construct the normal behavior from static analysis of the code or binary [6], gray-box anomaly detectors first subject the program to a set of benign inputs and observe its execution. Such observations serve as training data from which the normal execution of the program is learned. For example, if the return value of a system call is the same as the argument of another system call in many observations, a rule can be formed to govern the execution of these two system calls. This forms the basis of data-flow analysis in gray-box anomaly detection.

However, observations from benign executions are not necessarily *true* evidences supporting the rules learned. Fig 1 shows the source code of a sequence of system calls with some of the arguments and returns replaced by their observed values. In this example, a number of arguments and returns have the value of 5. An anomaly detector seeing a large number of observations like this would learn a rule that all these arguments and returns have to have the same value. However, a simple analysis into the semantics of the program reveals that the value 5 in the first three system calls (the value of a file descriptor) and the value 5 in the other system calls (the value of a socket descriptor) are completed unrelated. Therefore, this observation is not a true evidence supporting the rule learned.

Note that setting a higher threshold to the support and confidence in learning the observations cannot solve this problem. The problem is not about the lack of supportive observations or the existence of conflicting observations. The observations might have a high support and even 100% confidence, but the observations might not be *true* evidence to begin with.

Ground truthing the observations is hard, because, in general, it requires knowledge of the semantics of each argument/return of each instruction in each program to be protected. It is not practical to manually analyze such program semantics. Most data dependency analysis techniques would not help unless we turn to a white-box anomaly detector to analyze the source or binary of the program, which might not be an option at all in many scenarios.

In this paper, we propose using taint analysis to automatically help the ground truthing. Dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources such as user input [14], [15], [16], [17], [18]. Intuitively, the same taint source of two system call arguments or returns provides ground truth of their data dependency. In the observation shown in Fig 1, taint analysis could tell us that the value 5 in the first three system calls are of the same taint source while the value 5 in the other system calls are tainted differently. Armed with the taint information, we are a big step further in ground truthing the observations support on the rules learned.

We implement our proposed system based on a state-of-the-art anomaly detector that learns three sets of rules from data-flow and control-flow analysis [13]. Trace-driven evaluations on two different programs show that our proposed anomaly

```
1   open(dir, O_RDONLY|O_LARGEFILE) = 5
2   read(5, buf, 32768) = 32768
3   close(5) = 0
4   socket(PF_FILE, SOCK_STREAM, 0) = 5
5   fcntl64(5, F_SETFL, O_RDWR) = 0
6   connect(5, sockaddr *serv_addr, socklen_t addrlen) = -1
7   close(5) = 0
```

Fig. 1: Source code and its observed arguments and returns

detector not only manages to filter incorrect rules that would otherwise be learned (with high support and confidence) by a previously proposed state-of-the-art anomaly detector, but to recover good rules that are previously believed to be unreliable. We also present the overhead of our system in both the learning phase and the online monitoring phase. Results show that although there is a significant overhead of 5 times of the (offline) training overhead, online monitoring with our system is even faster than existing systems.

## II. Related Work

There has been a huge amount of work on anomaly detection using system calls. Gao et al. systematically categorized them into white-box, gray-box, and black-box approaches [4] depending on the information the detector uses. In this paper, we focus on gray-box (and black-box) detectors that do not analyze the source or binary of the program but observe benign executions of it to build the anomaly detection model.

Among the gray-box anomaly detectors, many capture the control-flow information of system calls in order to detect code-injection attacks [1], [2], [3], [4], [5], [7], [8]. Observations in these detectors contain system call sequence information but not values of arguments or returns, and are very much reliable in the sense that the observed order of system calls is always true (in benign executions). Therefore, the ground truthing in these observations concerns only whether a system call immediately follows another or there could be other system calls made between them. This paper does not discuss the ground truthing of this type of observations due to its relative simplicity.

Other detectors focus more on the data flow [9], [10], [11], [12], [13] by monitoring the arguments and returns of system calls. In one of the latest and most sophisticated anomaly detectors, Peng et al. [13] leverage the results from control flow analysis to learn more accurate and useful rules governing data flows among system call arguments and returns. However, even in this latest and most sophisticated work, the anomaly model does not investigate whether the observations (training data) *truly* support the rules learned. For example, when an observation shows that two system call arguments equal to one another, is it possible that they are, in fact, unrelated and simply equal to one another by chance? Could many of such observations lead to bad rules learned? It is clear that ground truthing these observations is difficult as it cannot be achieved by comparing values in the observation or calculating support or confidence. More work needs to be done.

Taint tracking has been proposed to deal with a broad range of security problems [19], [20], [21]. For example, James et al. [22] propose using taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. TEMU [23], which is built upon a whole-system emulator, QEMU, is proposed to do dynamic taint analysis. Our proposed ground truthing system obtains taint information of system call arguments and returns with a modified version of TEMU.

Perhaps the closest work to our proposed ground truthing system is the anomalous taint detection proposed by Lorenzo et al. [24], since both our ground truthing system and the anomalous taint detection make use of taint information for anomaly detection. However, the purposes of monitoring the taint propagation in the two systems are entirely different. Anomalous taint detection aims at lowering the false positives of the detector by focusing on only system calls that are tainted with external inputs. The idea was that untainted system calls could not be exploited by attackers, and therefore do not have to be part of the detection model. Our work on ground truthing goes one step further to investigate whether observations (on tainted system calls) truly support rules learned by the anomaly detector, regardless of its taint source being external inputs or not.

## III. Ground truthing observations and anomaly detection model

In this section, we first present the details of our proposal on ground truthing observations with taint information from dynamic monitoring of the program execution. We then show how the proposed ground truthing technique is applied on one of the latest anomaly detectors that learns data flow among system call arguments and returns. Implementation and evaluation on the proposed system are shown in the next section.

### A. Ground truthing observations with taint analysis

Revisiting the source code shown in Fig 1, we find that the cause of the misleading observation is that some arguments and returns have the same value although they are independent of one another. This shows that equality (or other relations) does not necessarily reveal data dependency. On the other hand, reliable data dependency can be obtained by dynamic taint analysis. The same taint source of two values provides ground truth of the data dependency.

Our anomaly detector therefore needs to assign taint tags to system call arguments and returns and monitor their propagation. Note that this is different from most previous work involving taint analysis (e.g., in [17], [18]). In most previous work on taint analysis, taint sources are data sources that are not trusted and include keyboard, network, and hard disk. In our work, we don't care if the data source is trusted or not, but how data flows among system call arguments and returns. Therefore, we label out-arguments and returns as taint sources (with different taint tags) even if they are trusted.

Assuming that the system calls shown in Fig 1 are the only system calls observed. When we see the first system call, none of the arguments or returns has a taint tag associated yet. We therefore assign a unique taint tag to each of them, as shown in the first row in the table in Fig 2. The program continues executing and the taint tags travel together with the corresponding values in memory. When the second system call request is observed, we realized some of its arguments carry taint tags, e.g., the first argument carries a taint tag of 1000, which is the same taint tag assigned to the return of the first system call. With this, our ground truthing system finds that the return of `open` and the first argument of `read` not only have the same value but depend on each other. Therefore, this observation can be used reliably for anomaly model construction.

| System call | Return value | Argument 1 | Argument 2 | Argument 3 |
|---|---|---|---|---|
| open | 5 (1000) | configure (1001) | 0x100000 (1002) | |
| read | 32768 (1001) | 5 (1000) | 0x8091DC0 (1003) | 32768 (1004) |
| close | 0 (1005) | 5 (1000) | | |
| socket | 5 (1006) | 1 (1007) | 1 (1008) | 0 (1009) |
| fcntl64 | 0 (1010) | 5 (1006) | 4 (1011) | 2 (1012) |
| connect | -1 (1013) | 5 (1006) | 0x8079520 (1014) | 16 (1015) |
| close | 0 (1016) | 5 (1006) | | |

Fig. 2: Taint tags propagating with system call arguments and returns

When system call `read` is about to return, our system finds that the second and the third arguments are not tainted yet, and therefore assigns two new taint tags to them. Again, the program continues to execute and we monitor the propagation of the taint tags together with the corresponding values in memory. When the fourth system call `socket` returns, our system realizes that although its return has the same value as some arguments and returns of the previous system calls (value equals 5), it does not come with a taint tag (1006 is the taint tag assigned to it later). Therefore, our ground truthing system concludes that this observation cannot be used for anomaly model construction.

### B. Constructing an anomaly detection model

After explaining how our ground truthing system analyzes an observation to determine if it can be used for anomaly model construction, we now explain how it can be used in conjunction with an existing anomaly detector to construct a more accurate anomaly detection model. Here we pick one of the most sophisticated and latest anomaly detector that leverages control-flow information to learn data-flow rules among system call arguments and returns [13], although our ground truthing system could be applied to other anomaly detectors, too.

*a) System call patterns:* System call patterns are learned from control-flow information in observations without ground truthing. A system call pattern is a sub-sequence of system calls (without considering arguments or returns) that are observed frequently. Intuitively, each pattern extracted corresponds to an atomic task performed by the program. For example, the following system call pattern $P_1$ consists of four system calls, `open`, `fstat64`, `mmap`, and `close`.

$$P_1 = \langle 005, 197, 192, 006 \rangle$$

We represent each system call in this pattern with the notation $P_1 S_i$ where $i \in [1, 4]$ in this example.

*b) Relations mining for rule sets A, B, and C:* Each relation to be mined governs two pieces of information to follow certain rules. For example, the two pieces of information could be two system call arguments or one system call argument and one system call return. The rule to follow could be that the two values equal to one another, or one is always greater than the other. For example, a particular rule for $P_1$ governing the equality of the return of the first system call and the first argument of the second system call could be represented as

$$P_1 S_1 A_0 = P_1 S_2 A_1$$

where $A_0$ denotes the return and $A_i, i \geq 1$ denotes an argument of the corresponding system call. Peng et al. has more detailed descriptions on the various relations that can be learned in their proposed anomaly detector [13].

After system call patterns are found, rules are learned among system call arguments and returns within each pattern (rule set A), on repeating patterns (rule set B), and across different patterns (rule set C).

Not every equality of arguments or returns is a rule. We perform the mining by calculating *support* and *confidence* of each rule, where support and confidence are calculated on observations that have passed the ground truthing mechanism. Unlike the previous work by Peng et al. where support is increased by one for any observations in which, e.g., two system call arguments have the same value, we increase support by one only if the two arguments have the same taint tag. This additional requirement is the ground truth that the observation truly supports the rule to be learned. Rules with support and confidence lower than thresholds minsupp or minconf, respectively, are filtered out due to their inconsistency.

## IV. Implementation and evaluation

In this section, we first present the implementation of our ground truthing system (Section IV-A). To evaluate the effectiveness of the ground truthing, we apply the ground truthing algorithm on the training of two popular programs (see Section IV-B) and compare the anomaly detection models built from our system with those built by one of the latest and most sophisticated anomaly detectors (see Section IV-C and Section IV-D). In the end, we show the overhead of our system by looking at the speed of convergence, time needed for training, and performance overhead in online monitoring in Section IV-E.

### A. Implementation details

As shown in Fig 3, there are many steps involved in applying our ground truthing system to anomaly model construction. Here we concentrate on the implementation of the ground truthing sub-system with taint analysis as other components are similar to existing anomaly detectors, e.g., that proposed by Peng et al. [13].

We extend TEMU [23], a whole-system dynamic binary analysis platform, to perform taint analysis on system calls. As discussed in Section III-A, our system differs from other systems for taint analysis in that we have to assign taint tags to all system call arguments and returns even if they are trusted. To do that, we intercept system calls in TEMU's instrumentation to obtain real-time system call information including the system call numbers, arguments, and returns. We then label the return values and out-arguments of all system calls as taint sources if they are not tainted yet. Taint sources are differentiated by labeling them with different taint tags. To track taint propagating among system calls, we also enable TEMU to trace all kernel instructions in addition to user instructions. The implementation of all these translate to $3,500$ lines of code added/modified in TEMU.

The output of our taint analysis for ground truthing observations is a system call execution trace augmented with taint information. The trace lists all system calls and values of the corresponding arguments and returns. Each system call argument and return also comes with a taint tag indicating its data dependency with other arguments and returns.

Other parts of the system are very similar to the anomaly detector proposed by Peng et al. [13]. We manage to obtain the source code of the anomaly detector proposed by Peng et al., and integrate our ground truthing system with it with small modifications. The original source code of Peng's anomaly detector also enables us to do a fair comparison between the two systems in their effectiveness, which is shown in the next subsections.

### B. Training

We perform our evaluation (ground truthing observations, training, and online monitoring) with the same desktop computer (running Linux kernel 2.6.27 on a 2.66-GHz dual core CPU and 4 GB of memory). Two popular programs from GNU utilities, `cmp` and `gzip` are used in our evaluation.

*1) cmp:* `cmp` is used to compare two files byte by byte. When difference between two files are found, `cmp` returns the position at which the difference occurs. Training of `cmp` consists of running it 100 times with files to be compared of different types (e.g., txt, doc, jpg, mp3 and cpp) and different sizes (e.g., from 1 KB to 7 MB).

*2) gzip:* For the purpose of demonstrating the effectiveness of our ground truthing system, we focus the training of `gzip` on its decompressing functionality. To do that, we first have to prepare for some compressed files. We pick 25 files of different types (e.g., txt, doc, jpg, mp3, tar, and ppt) and different sizes (e.g., from 6 KB to 7 MB) and compress them in two different ways, one in which the compressed file and the original file share the same prefix in their names (e.g., `abc.mp3` is compressed into `abc.mp3.gz`), and one where files have unrelated names (e.g., `abc.mp3` is compressed into `file1.gz`). This gives us 50 compressed files.

Training is then performed by decompressing these 50 files with two different command line options `-d` and `-dN`. The latter restores the original filename (e.g., `file1.gz` decompresses into `abc.mp3`), while the former does not (e.g., `file1.gz` decompresses into `file1`). This training is designed specifically to show the effectiveness of our ground truthing technique in which the taint tags tell us how the output filename is derived. Table I gives an example of the resulting decompressed file under four different scenarios.

| Source file | | `abc.mp3` | `abc.mp3` |
|---|---|---|---|
| Compressed file | | `abc.mp3.gz` | `file1.gz` |
| Decompressed file | `-d` | `abc.mp3` | `file1` |
| | `-dN` | `abc.mp3` | `abc.mp3` |

TABLE I: Decompressed file of gzip under different settings

We perform the training on two anomaly detectors with various settings of minsupp and minconf. DF denotes one of the latest and most sophisticated anomaly detectors that leverage control-flow information to learn data-flow relations among system call arguments and returns [13], and GT denotes our ground truthing technique. Table II summaries the number of rules learned by the two detectors.

| minsupp | | | 0 | 10 | 15 | 20 |
|---|---|---|---|---|---|---|
| minconf | | | 0.0 | 0.75 | 0.8 | 0.99 |
| cmp | rule set A | DF | 987 | 885 | 885 | 885 |
| | | GT | 56 | 54 | 54 | 54 |
| | rule set B | DF | 110 | 107 | 107 | 107 |
| | | GT | 98 | 98 | 98 | 98 |
| | rule set C | DF | 1611 | 1611 | 1585 | 1585 |
| | | GT | 118 | 118 | 93 | 93 |
| gzip | rule set A | DF | 1710 | 592 | 566 | 551 |
| | | GT | 145 | 28 | 27 | 26 |
| | rule set B | DF | 75 | 63 | 60 | 47 |
| | | GT | 42 | 33 | 30 | 24 |
| | rule set C | DF | 5692 | 1065 | 931 | 893 |
| | | GT | 1754 | 17 | 7 | 0 |

TABLE II: Number of rules learned

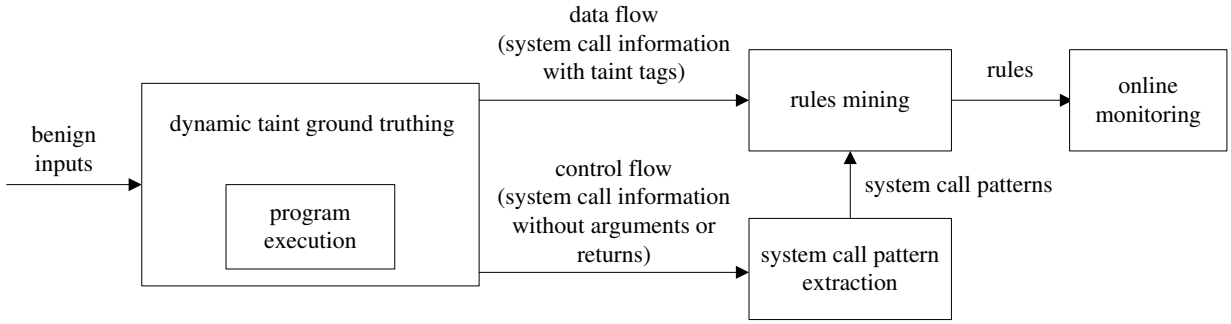



Fig. 3: System overview

### C. Filtering out bad rules

Table II shows that there are big differences between the number of rules learned by the two detectors. Upon further investigation, we realize that almost all rules learned by GT exist in rules learned by DF, with only a very small number of exceptions (see Section IV-D for discussion on good rules recovered by GT). In other words, the effectiveness of ground truthing is huge. A very large amount of rules learned by DF are not truly supported by observations.

Further more, we see that these large differences exist in all four settings of minsupp and minconf. In other words, having higher settings of minsupp and minconf in DF does not help filtering out the rules that violate the ground truthing check. Our ground truthing mechanism is a necessary step whose effectiveness cannot be achieved by manipulating the settings of minsupp and minconf.

We now turn our attention to these large number of rules filtered out by our ground truthing mechanism and analyze them in detail. After investigating into them, we find that most of these rules fall into one of the following two categories.

- The values of two system call attributes are the same, but they carry different meanings.
- Two system call attributes have the same meaning and value, but the two system calls are not related at all.

*1) Same value but different meanings:* This usually happens in arguments and returns where constants are used. These constants for different system calls might have the same value when they carry completely different meanings. For example, 0 in a return value may indicate success of the call, while 0 in an argument might be used to specify a particular action to be performed. Observations might show the same value of these constants, which in turn confuse DF to learn a rule; on the other hand, GT checks the taint tags of them to find out that they are unrelated, and therefore filters out the observations.

Here is one system call pattern learned in the training of `gzip`

$$P_2 = \langle \texttt{rt\_sigaction}, \texttt{open}, \texttt{fstat64}, \texttt{read}, \texttt{rt\_sigprocmask}, \texttt{open}, \texttt{rt\_sigprocmask}, \texttt{brk}, \texttt{brk} \rangle$$

and its rule from DF with a support 64 and confidence 1.0

$$P_2 S_3 A_0 = P_2 S_8 A_1$$

It says that the return of `fstat64` equals to the first argument of `brk`. `fstat64` returns 0 when it succeeds in checking information of a file. `brk` is usually used to change the size of the data segment. With an argument of 0, it finds the current location of the program `break`. It's apparent that these two values have different meanings and no data dependency. GT correctly identifies the ground truth of the observations and filters them out.

By the same token, another rule learned by DF says that the third argument of `rt_sigaction` equals to the third argument of `rt_sigprocmask`, which is NULL in the observations. GT, again, correctly performs ground truthing on the observations and filters them out since they don't share the same taint tag.

*2) Same value, same meaning, but unrelated system calls:* Here is a system call pattern from the training of `cmp`

$$P_5 = \langle \texttt{read}, \texttt{read}, \texttt{read} \rangle$$

and its rule learned in DF with a support of $9,420$ and confidence of 1.0

$$P_5 S_1 A_0 = P_5 S_2 A_0$$

where the return value of `read` indicates the number of bytes read.

By analyzing the source code that leads to these system calls, we find that they correspond to reading different files (files for comparison in executing `cmp`). In most cases, the returns of `read` are $4,096$ even when reading different files. Without ground truthing, DF finds such a rule with very large support and excellent confidence. However, the two system calls involved are actually unrelated.

We also see similar examples where the rule governs arguments of `read`, where the observations also fail the ground truthing test.

### D. Recovering good rules

Ground truthing helps filter out observations that do not truly support a rule from the rule mining process. However, ground truthing does not generate new observations from which rules are learned. That said, the filtering out of noisy observations could potentially make a rule that is previously believed to be unreliable (with low confidence) stand out. An

example appears in $P_2$ in training `gzip` (see Section IV-C), where both DF and GT learns a rule that the first arguments of the two `open` system calls share the same prefix.

The two `open` system calls correspond to the opening of the compressed file and the decompressed (output) file, respectively. As shown in Table I, these two files may or may not share the same prefix, depending on the option used (`-d` or `-dN`) and the filename of the compressed file. In our particular way of training, 25% of the time (when the source and compressed file do not share the same prefix and `-dN` is used) they do not share the same prefix. Therefore, DF finds a confidence of 0.75 for this rule.

On the other hand, GT checks for the taint tags on the two arguments as well, and realizes that the two arguments only share the same taint tag when option `-d` is used. When `-dN` is used, the name of the decompressed file is derived from the content of the compressed file instead of the name of the compressed file. Interestingly, when option `-d` is used, the two arguments of `open` always share the same prefix, and therefore the corresponding rule has a confidence of 1.0. GT manages to recover such a good rule, unlike DF in which the rule could have been filtered out due to its low confidence.

Note that these rules recovered might not be enforceable by the online monitoring, which is further discussed in Section V.

### E. Overhead

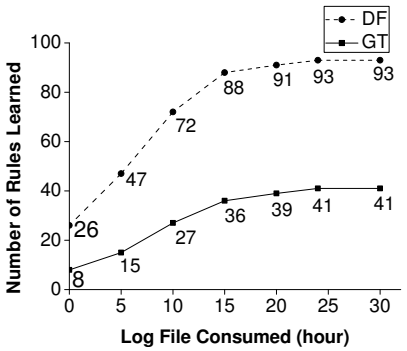
We perform our experiment to evaluate the overhead of our ground truthing system from three aspects — speed of convergence, time required for training, and overhead in online monitoring. To make a fair comparison between our ground truthing system and DF [13], we replicate the evaluation system setting in DF as much as possible.

Evaluation is performed on a system that tries to protect an `apache2` web server. Training of the anomaly detection model is performed by subjecting the server to 30 hours of logs of a university website in which there are a total of 148,370 (benign) `http` requests. We perform this experiment on both DF and GT in order to make a fair comparison.

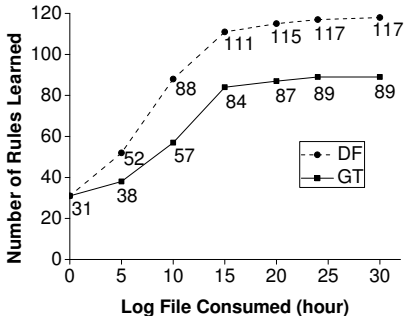
*1) Speed of convergence:* We have all `http` requests in a log file and replay them to the web server to perform training. Fig 4 shows the number of rules learned in the process of replaying the requests in the log file when maxdist $= 10$, minsupp $= 100$, and minconf $= 1.0$ for both DF and GT. It shows that the speed of convergence on the two techniques is similar to one another. It needs to consume roughly 25 hours of the `http` requests before learning converges.

*2) Training time:* Training is performed on a desktop computer with a 2.66-GHz dual core processor and 4 GB of memory. Training takes roughly 10 hours for DF while 50 hours for GT. This is mainly due to the taint propagation required in the training of GT. However, since training can be performed offline and potentially paralleled, we do not believe that the longer training time makes our ground truthing system impractical.

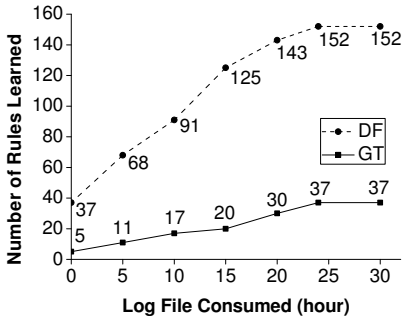
*3) Overhead in online monitoring:* The same machine used in training the anomaly detector is now used to host the web



(a) Rule set A



(b) Rule set B



(c) Rule set C

Fig. 4: Speed of convergence

server for online monitoring. The computer is running Linux with kernel 2.6.27, which is instrumented to have all system calls intercepted. Arguments and returns of each system call are checked for consistency with the rules learned in training. Note that the online monitoring system does not perform taint analysis. Therefore, the difference between DF and GT in this step is merely on the different rules used for consistency check. We discuss the limitation of this approach in Section V.

Just like what Peng et al. did in their evaluation of DF, we use a program to simulate (multiple) clients accessing the web server. Each client is configured to send a (valid) request to the web server every 10 milliseconds. When we simulate more and more concurrent clients, the web server becomes more and more heavily loaded.

Each experiment lasts for 60 seconds. We measure the

latency experienced by each client as an indication of the overhead of the system. This latency is measured as the difference between the time an `http` request is sent and the time the corresponding response is received by the client. Fig 5 shows the result when the number of concurrent clients (who are located in the same LAN as the web server) increases from 1 to 64. It also shows the latency when the server is not instrumented for online monitoring at all.

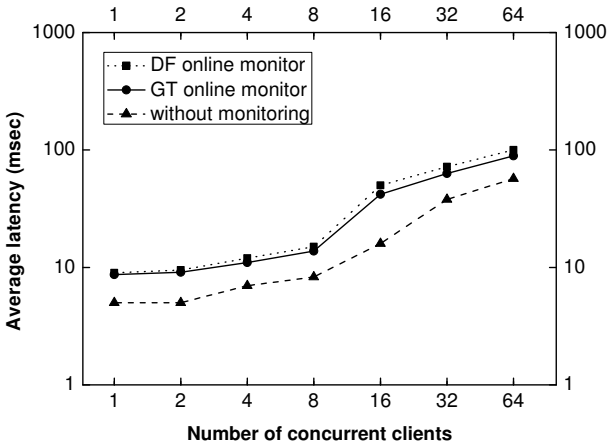



Fig. 5: Average latency experienced

Results show that GT constantly leads DF by a few milliseconds. The difference ranges from 2 to 3 milliseconds when there are fewer than 8 concurrent clients to about 20 milliseconds when there are 64 concurrent clients. As discussed earlier, GT and DF capture the same amount of information during online monitoring. The only difference between them is the number of rules to be checked. Since GT learns a significant smaller number of rules (due to the ground truthing filtering out many noisy observations), it has a better performance than DF. Comparing GT and a web server that is not configured for anomaly detection, GT adds roughly 4 to 40 milliseconds of latency.

## V. Limitations and conclusion

There are some limitations to the ground truthing system we propose in this paper. First, the ground truthing of observations is done by monitoring the propagation of taint tags. Therefore, some limitations of existing taint analysis, e.g., undertainting and overtainting [16] are inherited to our ground truthing system. Such imprecision introduced in tainting system call arguments and returns could potentially introduce errors into the rule mining process.

Second, although our ground truthing system makes a large step towards ground truthing observations by monitoring the taint propagation of system call arguments and returns which indicates data dependency, there is still subtle difference between that and the ultimate ground truth. For example, the fact that a system call argument is derived from the value of the return of another system call, and the fact that they equal to one another in many observations do not necessarily guarantee that they equal to one another in all future executions.

Third, our online monitoring does not perform taint analysis, which makes it impossible to enforce some rules learned. For example, the rule that we manage to recover from the training of `gzip` discussed in Section IV-D could only be enforced if taint analysis is performed in online monitoring. We choose not to do so because the large overhead taint analysis has on the system. We leave it our future work to make taint analysis in online monitoring more practical.

Last, though not intuitively true, some observations filtered out by our ground truthing system could actually contribute positively to the training of rules. One example is when the intrusion detector learns the relation of two data sources that appear to be unrelated (in the sense of data propagation). Such relations might not be supported by taint analysis and the corresponding observations will be filtered out, but they may nevertheless good rules capturing normal execution of a program.

In conclusion, we propose using taint analysis to automatically perform ground truthing of observations in an anomaly detector that tries to learn rules governing system call arguments and returns. Our experiments show that our ground truthing system manages to detect and filter out many observations that do not truly support rules learned. It is a significant improvement to the latest and most sophisticated anomaly detectors in that many bad rules are filtered out and good rules are restored. A trace-driven evaluation shows that although (offline) training takes significant longer time, convergence of the training exhibits about the same characteristics as in existing anomaly detectors, and online monitoring is even a little more efficient than existing ones.

## References


[1] H. Feng, O. Kolesnikov, P. Fogla, and W. Lee, "Anomaly detection using call stack information," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.

[2] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[3] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *Proceedings of the 11th ACM Conference on Computer and Communication Security(CCS 2003)*, 2003.

[4] ——, "On gray-box program tracking for anomaly detection," in *Proceedings of the 13th USENIX Security Symposium*, 2004.

[5] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[6] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proceedings: IEEE Symposium on Security and Privacy. Berkeley, California*, 2001.

[7] A. Wespi, M. Dacier, and H. Debar, "An intrusion-detection system based on the teiresias pattern-discovery algorithm," in *Proceedings of the 1999 European Institute for Computer Anti-Virus Research Conference*, 1999.

[8] ——, "Intrusion detection using variable-length audit trail patterns," in *Proceedings of the 3rd International Symposium on Recent Advances in Intrusion Detection*, 2000.

[9] S. Bhatkar, A. Chaturvedi, and R.Sekar, "Dataflow anomaly detection," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[10] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *Proceedings of ESORICS 2003*, 2003.

[11] G. Tandon and P. Chan, "Learning rules from system calls arguments and sequences from anomaly detection," in *ICDM Workshop on Data Mining for Computer Security(DMSEC), Melbourne, FL*, 2003.

[12] ——, "Learning useful system call attributes for anomaly detection," in *Proceedings of the 18th International FLAIRS Conference*, 2005.

[13] P. Li, H. Park, D. Gao, and J. Fu, "Bridging the gap between data-flow and control-flow analysis for anomaly detection," in *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC 2008), Anaheim, California, USA*, 2008.

[14] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior based malware clustering," in *Proceedings of the Network and Distributed System Security Symposium*, 2009.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. Mc-Daniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[16] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[17] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic spyware analysis," in *Proceedings of the 2007 Usenix Annual Conference*, 2007.

[18] H. Yin and D. Song, "Panorama: capturing system-wide information flow for malware detection and analysis," in *ACM Conference on Computer and Communications Security (CCS 2007)*, 2007.

[19] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks(DSN), pages 378-387, Washington, DC, USA*, 2005.

[20] G. W. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural support for programming languages and operating systems, pages 85-96, New York, NY, USA*, 2004.

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

[22] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of NDSS 2005, San Diego, California, USA*, 2005.

[23] H. Yin and D. Song, "Temu: Binary code analysis via whole-system layered annotative execution," EECS Department, University of California, Berkeley, Tech. Rep., Jan 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-3.html

[24] L. Cavallaro and R. Sekar, "Anomalous taint detection," University of Stony Brook, Technical Report SECLAB08-06, 2008.