# *MadeCR*: Correlation-based Malware Detection for Cognitive Radio

Yanzhi Dou, Kexiong (Curtis) Zeng, Yaling Yang, Danfeng (Daphne) Yao
Virginia Polytechnic Institute and State University
{yzdou, kexiong6, yyang8, danfeng}@vt.edu

***Abstract*—Cognitive Radio (CR) is an intelligent radio technology to boost spectrum utilization and is likely to be widely spread in the near future. However, its flexible software-oriented design may be exploited by an adversary to control CR devices to launch large scale attacks on a wide range of critical wireless infrastructures. To proactively mitigate the potentially serious threat, this paper presents *MadeCR*, a Correlation-based *Ma*lware *de*tection system for *CR*. *MadeCR* exploits correlations among CR applications' component actions to detect malicious behaviors. In addition, a significant contribution of the paper is a general experimentation method referred to as *mutation testing* to comprehensively evaluate the effectiveness of the anomaly detection method against a large number of artificial malware cases. Evaluation shows that *MadeCR* detects malicious behaviors within 1.10s at an accuracy of 94.9%.**

## I. Introduction

Cognitive Radio (CR) refers to an intelligent radio that can be programmed and configured dynamically. It is proven to provide drastically improved spectrum utilization. While the benefits of CR are obvious and its future appears promising, it also brings along uniquely challenging security and information assurance issues [1]. A traditional radio can only affect its own network because its operation spectrum is constrained by its fixed hardware design. Thus, while malware may affect these traditional radios, the impact is limited to a single wireless system. CR, however, often has flexible software-oriented design that can be configured to opportunistically access available spectrum in an expansive spectrum range. Thus, by remotely exploiting the inherent vulnerability of software in CR, an adversary can control CR devices to launch large scale attacks on wireless systems at many different spectrum bands. This broad potential threat makes compromised CR much more destructive than traditional systems. Therefore, unlike a traditional wireless system, where it may be acceptable to consider security as an afterthought, security issues of CR must be addressed proactively for this technology to pass regulation requirement and become widely deployed.

Existing proposals for CR security fall into the following three categories. The first category (e.g., [2], [3]) seeks to prevent downloading of malicious software into CR systems through the use of digital signature and encryption. The second category (e.g., [1], [4]) leverages ordinary personal computer protection measures to reduce the chance that CR systems be infected with malicious software. The third category of methods (e.g., [5], [6]) focus on the networking aspect of CR security, where mechanisms are designed to ensure that CR networks can resist attacks from compromised CR nodes. Despite these recent advances in CR security, run-time software behavioral monitoring has not been systematically investigated for CR. By integrating proper domain knowledge, the run-time behavioral monitoring method can prevent CR nodes from being compromised in the first place.

The aim of this paper is to enhance the security of CR device itself by designing an effective malware detection system named *MadeCR* (a Correlation-based *Ma*lware *de*tection system for *CR*). Due to the proactive defense requirement on CR security, *MadeCR* must be built and evaluated while there is a lack of existing CR malware. This means that *MadeCR* must effectively detect zero-day attacks. *MadeCR* achieves accurate detection by exploiting the correlations among CR component actions. A normal-operating CR exhibits strong correlations among its operational events at software level. A CR that is infected by malware, however, is unlikely to exhibit similar correlations. Leveraging anomaly detection techniques [7], *MadeCR* builds models of normal audit data (or data containing no intrusions) to extract correlation properties among CR actions and detects malware based on measuring the deviations of their behaviors from the normal model. To the best of our knowledge, this is the first work that enhances the security of CR networks at device level through monitoring CR applications' behaviors.

Although anomaly detection is widely used in intrusion detection systems (IDS) in many areas, it encounters the following unique challenges when first applied to CR systems. *(1)* CR anomaly analysis needs to include data flow analysis, which is generally considered too difficult and too expensive due to the occurrence of continuous numeric values in data calculation and data passing. Traditional data flow analysis attempting to enumerate all the appearing samples in continuous numeric space would lead to state explosion, and hence data flow analysis is usually avoided in previous IDS works [8]. Monitoring data flow is important for the correct execution of a CR application and cannot be ignored. In CR, data from both external environments (e.g., sensed licensed user signals) and internal resources (e.g., waveform capabilities) is collected and then analyzed by cognitive engine, which is responsible for configuring radio and system parameters accordingly. These configuration parameters are then delivered to software radio platform to alter the radios' behaviors. In the process, one small data error can drastically alter the behaviors of a CR.

978-1-4799-8381-0/15/$31.00 ©2015 IEEE

*(2)* CR anomaly analysis needs to detect suppression attacks, which are not considered in existing IDSs. We define a suppression attack in CR as an attack suppresses CR nodes' desirable behaviors, making some functionalities of the nodes become inactive. For example, in normal situations, a CR user needs to periodically check if licensed users appear in its current channel. A malware may suppress such checking action so that the CR user will never switch channel, resulting in jamming to licensed users. Traditional IDSs cannot detect suppression attacks because the particular category of attacks causes harm not by introducing new undesirable behaviors, but rather by suppressing a CR application's desirable behaviors. Suppression attacks represent a large category of anomalies where probability distribution of function call invocations is skewed, differing from the distribution in normal executions. This kind of attacks is possible when a vulnerable CR node is hijacked and its control flow is altered.

*(3)* Traditional IDSs evaluate their detection performance on the malware captured in the real environment. Since CR has not reached a widely-deployed state, there is a lack of real malware cases in the field that can be used to evaluate the detection accuracy of our proactive defending scheme. How to conduct effective evaluation becomes the third challenge.

In this paper, we make the following contributions:

- We involve data flow analysis in the design of *MadeCR* and solve the complexity issue by innovatively combining clustering techniques with CR operation features.
- We solve the unique challenge of suppression attacks by designing new computational methods for security verification against them.
- We propose a new approach to automatically generate artificial CR malware cases through mutation testing techniques.
- We employ a new method to automatically identify the critical vulnerable components in CR system and refine the detection system accordingly.

The rest of the paper is organized as follows. We introduce the attack model and security goals in Section II. In Section III, we present the design of *MadeCR* in detail. After that, we describe the approach to generate artificial malware cases for evaluation in Section IV. Evaluation results are showed in Section V. We provide the related work in Section VI. Finally, we conclude this paper in Section VII.

## II. Attack model and security goals

We consider a setting where an attacker can maliciously modify the CR execution during software download, initialization and runtime [9]. For example, the attacker may:

- Reconfigure a CR node with improper RF parameters (e.g., center frequency, transmit power, modulation scheme).
- Suppress some functionalities of a CR node by inserting infinite loop before executing the corresponding code segments.

Generally speaking, an attacker's goal is to make a CR function out of the regulations, which causes interference to other users' normal communication or even denial-of-service (DoS) attacks on critical public wireless infrastructures. Another possible goal is to gain disproportionate access to network resources for some CR system owners.

*The goal of MadeCR is to detect malicious modifications to CR execution*. In particular, *MadeCR* needs to guarantee that if the malicious modifications to CR execution are targeted at security-critical parts (e.g., RF parameters), it can accurately detect them. GNURadio and USRP – the most popular software and hardware platform for CR development – are used as the foundation for developing and evaluating the prototype of *MadeCR*.

The design of *MadeCR* has the following assumptions:

**Assumption 1**: To cause harm, a compromised CR application must conduct different behaviors from that normally seen, and these behaviors may be readily monitored.

**Assumption 2**: The training data is nearly complete with regard to all possible normal behaviors of a CR application.

This is a key assumption for using a learning algorithm for anomaly detection [10]. We satisfy this by the design of the *Application Driver* Component of *MadeCR* (see Section III)

**Assumption 3**: The detection system itself will not be compromised.

This is an implicit assumption for nearly all IDSs. It can be satisfied by putting the detection system in a more secure environment through the use of a virtual machine monitor (VMM) proposed in [11].

## III. System Design

The overview of *MadeCR* is shown in Figure 1. It is composed of several components, including *Application Driver*, *Operation Tracer*, *Argument Processor*, *Normal Profile Database*, *Anomaly Detector*, and *Anomaly Feature Database*. *Argument Processor* and *Anomaly Detector* are the most important security components in our design. *Argument Processor* is able to expose the essence in data flow correlations and reduce the complexity of data flow analysis by innovatively adopting clustering techniques. *Anomaly Detector* aids in detecting suppression attacks by introducing new computational methods for security verification to $N-$gram model [12] and Hidden Markov Model (HMM) [13].

### A. Application Driver

In the training phase, *Application Driver* needs to traverse as many as possible branches while executing a CR application, so that we can obtain a more comprehensive description of the application's behaviors. Therefore, the *Application Driver* executes the CR application for a large number of times. At each time, the application is fed with varied user inputs, application demands and operation environment.

### B. Operation Tracer

*Operation Tracer* focuses on monitoring radio-related function calls because they are most relevant to CR operations. A CR application needs to invocate these function calls to access system resources and reconfigure the radio. In *MadeCR*, we build a prototype of *Operation Tracer* to record CR

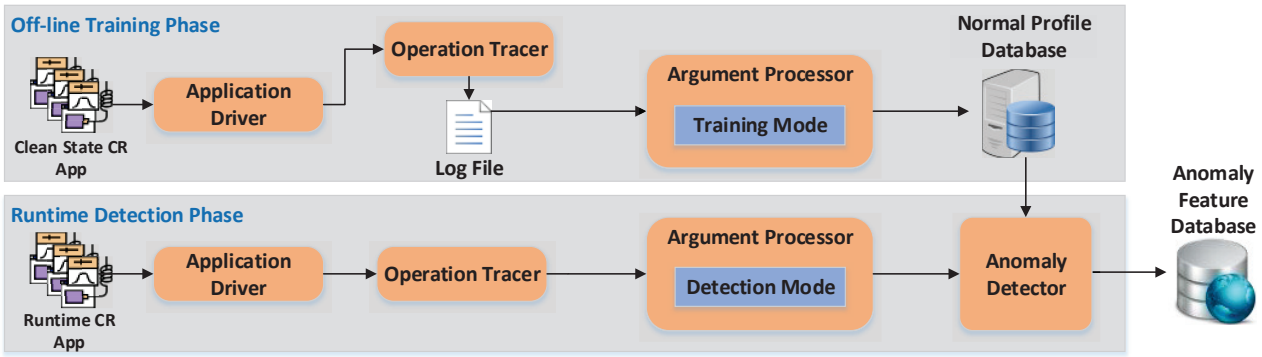


Fig. 1. Overview of *MadeCR*

applications' function calls to GNU Radio libraries. Examples of the recorded function calls are $set\_center\_freq(freq)$, $sample\_rate = get\_samp\_rate()$, and $send\_pkt(msg)$. As CR applications using GNU Radio are primarily written in Python programming language, *Operation Tracer* leverages Python's buildin tracing modules to record function call relationships. For each function being called, its name, file path of its definition, beginning line number of its function body, arguments passed to it and its return value are recorded. The first three items can uniquely identify a function, which can provide information of vulnerable points where the attacker aims. The arguments and return values of function calls are important for monitoring CR applications' data flow. The function name, file path, line number, and return values are collected using Python's "trace" module. The arguments are obtained through the "inspect" module of Python. Both numbers and strings in arguments and return values are recorded. The trace data is chronologically arranged and stored into a log file, which is encrypted to avoid being tampered with.

### C. Argument Processor

The complexity of data flow analysis is known to be very high and hence is avoided in existing intrusion detection works. However, analyzing data flow is absolutely necessary in CR since it is critical for CR operation. *MadeCR* innovatively uses *Argument Processor* to solve this challenge. In a high level, *Argument Processor* reduces the potentially very large data into sequences of a small number of states. Each of the states is uniquely identified by a label. This transformation converts data in data flow from continuous space into discrete space.

*1) Overall Design:* The inputs to *Argument Processor* are the trace data in the form of sequences of function calls. To expose the correlations between each function's input arguments and return value, the first processing step of *Argument Processor* disintegrates every function call instance in the trace data to two elements, one with only the input arguments and one with only the return value. In order to be brief yet not cause ambiguity, we use argument as the general term for both input argument and return value in the rest of this paper.

In the second step, to reduce data analysis complexity, *Argument Processor* compresses the infinite choices of argument values into a finite number of clusters. The purpose of the clustering operation is to reduce the state space in data flow analysis. Finding the right state space is a non-trivial problem. In CR, some functions do not have one single normal behavior but a variety of normal behaviors in different yet naturally occurring operation contexts of a CR application. Under different operation contexts, these functions may take different argument values as input, then act differently in response to these inputs, and finally output different return values. *Argument Processor* must use a small number of states to describe all these possible set of normal data and control flow behaviors under the entire range of the potentially infinite natural operation contexts. *Argument Processor* solves this challenge by leveraging the fact that while the number of unique operation contexts and input/return data values may be infinite, similar operation contexts usually lead to similar arguments, which then often create the same function behavior.

Following this observation, in the training mode, *Argument Processor* divides elements of the same function into clusters based on these elements' argument values. The element instances in each cluster have similar argument values and hence are likely to be caused by similar contexts and exhibit similar function behaviors. Each cluster is given a unique name and hence the sequences of function calls in the training trace data become the sequences of cluster labels. Figure 2 shows an example of the clustering process. Similarly, in the runtime detection mode, Argument Processor classifies each element in the runtime trace into the closest cluster according to its argument values and converts the trace also into a sequence of cluster labels.

*2) Training Mode:* Specifically, the clustering algorithm in the training mode works as follows. Recall that *Operation Tracer* records argument values, including numbers and strings, in the training mode. Some of these argument values are drawn from a small set of possible alternatives (i.e. tokens, state flags of CR operation, elements of an enumeration such like modulation schemes) and are called countable arguments. Arguments that cannot be enumerated are defined to be uncountable. For example, the channel signal strength acquired by sensors, choices of sampling rate, and level of noise floor are all uncountable arguments. Countable arguments are naturally separated and each of its unique value creates its own element cluster. Uncountable arguments can be divided

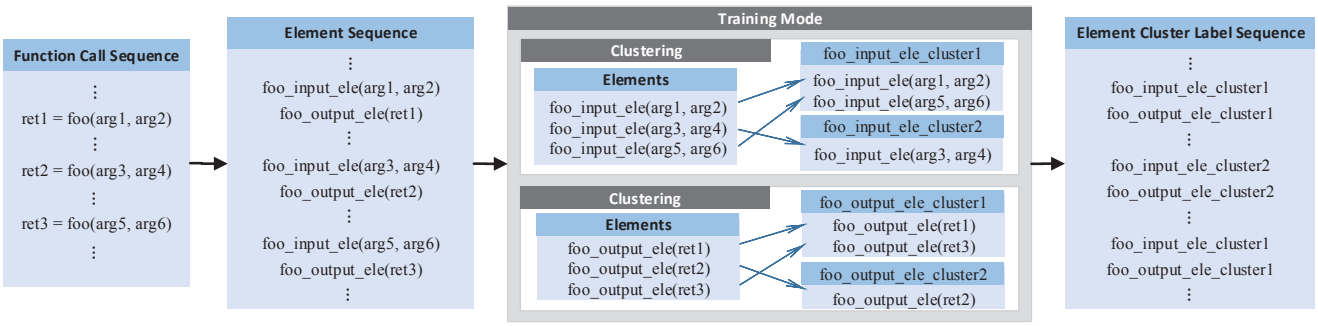


Fig. 2. An example of clustering process

further into two groups: uncountable strings and uncountable numbers. Uncountable strings are ignored because most of them are used to represent text messages, such as operating messages printed to the screen, and do not affect execution flow.

For uncountable numbers, a single-linkage and bottom-up hierarchical agglomerative clustering algorithm [14] is adopted to acquire clusters for elements with uncountable numbers as their arguments. Defining the values of an element's uncountable arguments as an argument vector, the clustering algorithm initially treats each element with its argument vector as a singleton cluster. It then successively agglomerates pairs of clusters until stop criterion is satisfied, as shown in the following algorithm:

a) Each of the $M_d$ elements with different argument vectors is assigned to an individual cluster at the outset. A $M_d \times M_d$ distance matrix $D$, whose entry at $(i, j)$ is the distance between argument vector $i$ and $j$, is computed.
b) Among all the current clusters, two clusters with the smallest distance $min(D)$ are picked.
c) The two picked clusters are merged and the distance matrix $D$ is updated by recomputing the distance between the newly aggregated cluster and the remaining ones.
d) Repeat step b and c until the total number of final clusters $N_s$ is reached.

Single-linkage agglomerative clustering algorithm defines the distance between two clusters as the minimum distance between two elements, one in each cluster. The distance $d$ between two elements is defined by standardized euclidean distance of their argument vectors:

$$d_{i,j} = \sqrt{\sum_{m=1}^{M} \left( \frac{a_i^{(m)} - a_j^{(m)}}{\sigma_a^{(m)}} \right)^2}, \qquad (1)$$

where $d_{i,j}$ is the distance between argument vector $i$ and argument vector $j$. Argument vector has entries of $a^{(1)}, ...a^{(M)}$. $\sigma_a^{(m)}$ is the standard deviation of argument $a^{(m)}$ over all invocation cases of the element in the training data.

"L method" proposed in [15] is adopted to determine the optimal number of clusters (i.e. the optimal $N_s$) for the stop criterion. In each step b) of the agglomerative clustering approach mentioned above, we record the distance between the two clusters which are the most similar, and denote it as merge distance. An evaluation graph is plotted by taking the number of clusters as x axis and the merge distance as y axis. The L method finds the knee point in the evaluation graph, which is the point of maximum curvature. This knee point is the optimal $N_s$ because it represents a balance of clusters that are both highly homogeneous, and also dissimilar to each others.

*3) Detection Mode:* In detection mode, *Argument Processor* applies $k$-nearest neighbours ($k$-NN) algorithm [16] to classify every runtime element into an element cluster, which is built in the training mode. Then, the *Argument Processor* assigns the runtime element a label that indicates the cluster it belongs to. $k$-NN algorithm classifies a runtime element as follows. Using (1), it computes the distances between the runtime element and all the training-time elements that map to the same function call . Then, the first $k$ training-time elements that are closest to the runtime element are picked out and form a set. The runtime element is classified into the cluster that has the largest proportion of elements in the set. The underlying assumption is that elements belonging to the same cluster will gather together in the space of argument vector.

*4) Computational Complexity:* In the training mode, the computational complexity of agglomerative clustering approach is $O({M_d}^2)$ [14]. "L method" needs $O(M_d)$ computations to find the knee point of a evaluation graph based on the results of agglomerative clustering approach. So the total computational complexity of the training mode is $O({M_d}^2)$. In the detection mode, the computational complexity of $k$-NN algorithm is $O(M_d)$.

### D. Anomaly Detector

Through *Argument Processor*, the trace of CR applications are converted into sequences of element cluster labels. *Anomaly Detector* aims at recognizing abnormal sequences whose composition and variety substantially deviate from normal executions. In *Anomaly Detector*, we design new computational methods for security verification against the unique suppression attacks in CR. We demonstrate how new security rules can be integrated into existing $N-$gram and HMM models. Specifically, we detect abnormal distributions of cluster labels through extracting statistical data from training data sets.

*1) $N-$gram model:* Given a cluster label sequence, all the unique substrings of fixed length $N$ in the label sequence are called $N-$grams. They can be generated by sliding an $N-$size

window along the label sequence and recording all the unique substrings in the process. The set of these substrings is named an $N-$gram set. In the training phase, $N-$gram sets are put into *Normal Profile Database*, while in the detection phase, *Anomaly Detector* compares runtime $N-$gram sets with the $N-$gram sets in *Normal Profile Database* to calculate their deviations using the following four metrics:

• Metric $N_{oc}$ is called the number of missing $N-$grams and is defined to be the number of $N-$grams that are in *Normal Profile Database* but not in the runtime set. Ideally, we expect $N_{oc}$ to be very small for new instances of normal trace and jump significantly when suppression attacks occurs.

• Metric $S_{Ac}$ is a local measure. Recall that we run a clean state CR application for many times in *Application Driver*. Thus, we get many training $N-$gram sets, $J_{s1}, J_{s2}, ..., J_{sn}$, in *Normal Profile Database*, where each training $N-$gram set corresponds to one execution of the CR application. For each training $N-$gram set, we compute the number of $N-$grams that are in the training set but not in the runtime $N-$grams set $I$. The training $N-$gram set that has the smallest such number, denoted as $J_{sk}$, is termed as the most similar training set to the runtime $N-$gram set $I$. $S_{Ac}$ is defined to be the maximum Hamming distance between an $N-$gram $j$ in $J_{sk}$ and $I$, namely $S_{Ac} = \max_{j \in J_{sk}} d(j, I)$. Here, the Hamming distance between an $N-$gram $j$ and an $N-$gram set $I$ is defined as the minimum hamming distance between $j$ and any $N-$gram in $I$. The hamming distance between two $N-$grams $i$ and $j$ is simply the number of positions where the two $N-$grams differ.

• Metric $N_o$ is defined as the number of outliers, which are the $N-$grams appear in the runtime $N-$gram set but not in the normal $N-$gram database. Ideally, we expect $N_o$ to be zero for new instances of normal trace and jump significantly when anomalous behaviors occurs.

• Metric $S_A$ is defined as $S_A = \max_{i \in I} \min_{J_{sj} \in D} d(i, J_{sj})$, where $D$ is *Normal Profile Database* and $I$ is the runtime $N-$gram set.

$N_o$ and $S_A$ are proposed in the existing literature [12], which focus on detecting the existence of undesirable behaviors. However, the two metrics are not able to capture the suppression attacks, which cause harm not by introducing new undesirable behaviors, but rather by suppressing a CR application's desirable behaviors. To mitigate suppression attacks, we design the other two metrics, $N_{oc}$ and $S_{Ac}$, to capture the lack of desirable behaviors from a runtime trace.

We propose the following criterion to aggregate the four metrics to determine whether a runtime case is malicious:

$$R = (N_{oc} \geq t_{oc} \wedge S_{Ac} \geq t_{Ac}) \vee (N_o \geq t_o \wedge S_A \geq t_A) \quad (2)$$

The runtime case is marked malicious if $R = True$. $t_{oc}$, $t_{Ac}$, $t_o$, $t_A$ are the corresponding thresholds, which must be set appropriately to reduce false positive rate while attempting to detect more malware. These thresholds are set empirically. To figure out a proper $t_{oc}$, we count the number of "outlier" $N-$grams of each training set by comparing the normal database excluding a training set with the particular training set. $t_{oc}$ is configured as the largest "outlier" number among all the training sets. Essentially, $t_{oc}$ tries to capture the largest possible suppressed behaviors that can be exhibited by a normal trace. Hence, if a program trace's $N_{oc}$ is larger than $t_{oc}$, it is very likely that some functionality of the program is maliciously suppressed. We set $t_{Ac}$ as $N$, the number of elements in an $N-$gram. We set $t_o$ as 1 and $t_A$ as $\lceil \frac{N}{3} \rceil$, which means that we regard a case as malicious if it contains at least one $N-$gram whose hamming distance is greater than $\lceil \frac{N}{3} \rceil$ to the normal database.

*2) Hidden Markov Model:* Assuming that the execution of a CR application follows some state transitions, the normal CR behavior can be modeled by constructing a HMM in *Normal Profile Database*. The states in the HMM are not directly observable, but each state can randomly produce element cluster labels following certain emission probability distribution. A state can transit to other states with some transition probability. We can construct this HMM in the training phase by using the sequences of element cluster labels produced by the *Argument Processor* as follows.

First, we set number of states equal to the number of unique element cluster labels. Transitions are allowed to be made between any two states. We adopt the well-known Baum-Welch algorithm [17] to train the HMM. The algorithm uses the element cluster label sequences as training data and has four steps, i.e. initialization, E-step, M-step, and iteration. Transition and emission probabilities are initialized randomly. During training, the state probabilities and emission probabilities are iteratively adjusted in the E-step and M-step to increase the likelihood that the model will generate the traces in the training data. Training is terminated when the likelihood of the model producing a second set of normal sequences (not used in training) stops improving. In this way, over-fitting is avoided.

Using the HMM in *Normal Profile Database*, *Anomaly Detector* can detect suppression attacks as follows. Traces of suppression attacks tend to produce less unique state transitions, because some desirable transitions representing the corresponding desirable function invocations are suppressed. Thus, *Anomaly Detector* uses Viterbi algorithm [17] to calculate the most likely state transition sequence of a runtime trace. The number of unique state transitions in the sequence is counted and denoted as $Q$. If $Q$ is smaller than a threshold $Q_T$, the runtime trace is marked to be generated under suppression attacks. The value of $Q_T$ is set as the smallest number of unique transitions among all the normal traces.

To detect other attacks, *Anomaly Detector* computes the likelihood $P$ of a runtime trace to be produced by the HMM using the forward algorithm [17], and then compare this probability with a threshold $P_T$. If $P < P_T$, the trace is marked malicious. As the likelihood of producing a trace is sensitive to its length in HMM, we divide it by the trace length for normalization so that the comparison with other traces of different lengths can be fair. The proper value of $P_T$ can be found through experiments as described in Section V.

Combining the two detection methods, the aggregated

HMM detection criterion in *Anomaly Detector* is:

$$R = (Q < Q_T) \vee (P < P_T) \quad (3)$$

If $R = True$, the trace is malicious.

*3) Comparison of Space and Time Complexity:*

• Storing HMM in *Normal Profile Database* takes smaller space than $N-$gram. Assume the training data has $S$ unique element cluster labels. HMM's transition matrix and emission matrix have $S^2$ values to store, respectively. For $N-$gram, in the worst case, the number of unique N-grams to store can be $\frac{S!}{(S-N)!}$ ($N$ is the window size).

• Generally, in the training phase, the time complexity of $N-$gram is smaller than HMM; In the detection phase, the time complexity of $N-$gram is larger than HMM. In the training phase, for HMM, the Baum-Welch algorithm has a complexity of $O(TS^2)$ [17], where $T$ is the length of the trace of cluster labels. For $N-$gram, incorporating a new training trace into *Normal Profile Database* only takes $O(T)$. In the detection phase, the complexity of the Viterbi algorithm and the forward algorithm used in the HMM approach are both $O(TS^2)$ [17]. For $N-$gram, it takes $O(N)$ comparisons to determine whether a runtime $N-$gram is a mismatch if the normal $N-$grams are stored in a forest of trees. To compute the distance between a runtime $N-$gram and the normal $N-$gram database, it requires $O(DN)$ comparisons, where $D$ is the number of $N-$grams in the the normal database. If $D \approx S^N$, the complexity will be $NS^N$.

### E. Anomaly Feature Database

All the anomaly behaviors captured during the execution of CR applications are recorded. The logging data obtained in *Operation Tracer* provides a comprehensive record of the anomalies, including which parameters, which function calls, and which files may behave anomalously. The data is sent to a remote *Anomaly Feature Database*, which collects the anomaly reports from every CR node and mines the malware patterns from the vast data for further signature-based detection. Moreover, the anomaly data exposes the vulnerable parts of the software, which will aid the maintainers in fixing them and updating the software to defend further attacks.

### F. System Refinement

We further improve the detection accuracy and performance of our solutions by refining the system. Specifically, we automatically select a set of security-critical function calls, and only monitor their running state while neglecting other irrelevant calls. The refinement method is able to lower *MadeCR*'s false alarm rate by reducing the ambiguity caused by irrelevant logging data. Additionally, by concentrating on the most crucial function calls, it can increase both detection accuracy and detection speed.

The refinement method is based on the observation that many intercepted function calls provide little contextual information of the running states of CR applications. For example, some functions are only involved in printing messages on screen, and some functions are only tokens of thread scheduling in the Python programming language. What's worse, the latter group of the function calls are often lead to false alarms because these functions tend to occur at unexpected locations in execution traces. On the other hand, since the operation fidelity of CR is determined by the integrity of its RF parameters, function calls that can alter the values of the RF parameters directly or indirectly are critical for CR security. Based on the above observation, in the refined version of *MadeCR*, only these security-critical function calls are logged by *Operation Tracer* and processed by the later components of *MadeCR*. How to automatically identify these security-critical function calls will be addressed in Section IV.

## IV. Artificial Malware Generation

Traditional IDSs follow a reactive defense strategy, which means that they are only built after the protected systems have been attacked. Thus, the effectiveness of IDSs can be evaluated using real malware. However, such reactive defense strategy is not appropriate for CR due to CR's potentially serious threat to not only CR systems but also other critical wireless infrastructures. *MadeCR*, thus, is developed before the appearance of real malware. This poses a challenge on how to assess the effectiveness of *MadeCR*. Traditional evaluation methods using the real-world malware are no longer feasible due to a lack of malware in the new field. To fill the gap between proactive defense methods and their evaluations, we propose a method to automatically generate artificial malware.

*1) Observation:* The artificial malware cases are created by adopting mutation testing techniques, which are currently used in software testing field. It involves injecting various mutations into a program's source code or byte code to create mutants. A software test's quality is evaluated by the percentage of mutants that are caught by the test.

We observe that malware, in a certain sense, can be viewed as a special type of mutants. A malware case usually needs to inject malicious code into an application and/or modify the application's variable values and instruction execution sequences. Hence, if a malware detection scheme is effective on detecting artificially generated mutants at runtime, it should be also effective on detecting malware in real world. Based on this rationale, we automatically generate artificial CR malware by applying mutation operators to GNU Radio software and applications.

*2) Method:* As shown in Table I, we apply six mutation operators on the application at byte code level to automatically generate mutants. The mutation operators change the original program in different ways and can create a wide range of malware-alike behaviors. For example,"VAR" modifies a variable's value and may cause changes to CR's center frequency and transmit power. "JMP" modifies the branch condition, which can cause a CR application to skip the channel switching action upon discovery of licensed users. "ARI" and "CMP" can change the arithmetical operations and comparisons in the application respectively, which incurs data error (e.g., wrong computed signal sampling rate). "JPT" makes a program jump to a random position, which may cause unexpected behaviors. "IIL" can insert a infinite loop at an arbitrary position, which

can impede the application's execution and thus suppress its normal functionalities.

Since mutation testing is not originally designed for malware generation, not every mutant causes malicious behaviors. Some mutants are benign and do not change the original program's semantics. For example, a mutant that changes "$for(i = 0; i \leq 5; i++)$" to "$for(i = 0; i \neq 5; i++)$" will not alter the program's behavior. In addition, some other mutants only change the unimportant branches and variables of the testing CR application. For example, some branches are just involved in writing operation logs to a file, so mutating them will not affect the radio's behaviors. The mutants that do not change the radio's behaviors are viewed as benign mutants. To accurately reflect *MadeCR*'s performance, these benign mutants are excluded from the testing set and only mutants that exhibit malicious behaviors are used for evaluation.

TABLE I
DESCRIPTION AND STATISTICS OF MUTATION OPERATORS

| Mutation operator | Description | # of all mutants | # of malicious mutants |
|---|---|---|---|
| JMP | Alter Branch Condition | 92 | 55 |
| JPT | Randomize Jump Target | 491 | 393 |
| VAR | Change Variable Value | 597 | 417 |
| ARI | Replace Arithmetic Operator | 799 | 598 |
| CMP | Replace Comparison Operator | 308 | 144 |
| IIL | Insert Infinite Loop | 468 | 339 |
| Total | | 2755 | 1946 |

*3) Identifying security-critical functions calls:* Besides serving as testing cases, the mutants can be used to automatically identify security-critical function calls for system refinement as follows.

Firstly, we construct a security-critical database, which is initially composed of all the USRP Hardware Driver (UHD) APIs that take the RF parameters as input, and output the corresponding waveforms. The UHD is the final boundary from software side to hardware side, and all the GNU Radio applications need to invocate these APIs to reconfigure a radio. If these APIs behave anomalously, the radio will be reconfigured with anomalous waveforms. Secondly, we send all the mutants (including the benign ones) into *MadeCR* and obtain their anomaly features (e.g., mismatched $N-$grams) in *Anomaly Feature Database*. If an anomaly feature of a mutant contains an API that is included in the security-critical database, all the other function calls in the anomaly feature are added to the security-critical database. This step is to ensure that functions that may indirectly cause anomalies in RF parameters are included in the security-critical database. After the security-critical database is completed, only the function calls in the database need to be tracked to detect the malware targeting at RF parameters.

Since all the function calls that have the potential to alter the RF parameters are monitored, our refinement method ensures the validity of RF operation. Similar approaches can also be used to protect other critical operation parameters of CR.

## V. PROTOTYPE & EVALUATION

A prototype of *MadeCR* is implemented on a host computer with Intel Core i7-2600 CPU @ 3.40GHz * 8 and 8GB memory. We set up a CR testbed with GNU Radio 3.7.1 and three N210 USRPs. Each USRP is individually connected with a host computer. Two of the USRPs are secondary transceivers and the rest one is a primary transmitter. The secondary transceivers execute a CR application derived from CORNET [18], which is an open-access cognitive radio testbed. In this application, each secondary transceiver senses both the external environment and internal applications' requirements, and makes adjustment to its radio parameters accordingly. Specifically, for external environment, the secondary transceivers will switch its current transmission channel to a white channel if it perceives the existence of a primary user on the current channel; For internal applications' requirement, the secondary transceivers will change its transmission bandwidth according to the applications' required data rate. The CR application has altogether 703 lines of source code and 2114 instructions of byte code. As shown in Table I, after applying mutation testing techniques, we generate 2755 mutants and 1946 are selected as malicious ones.

We evaluate two versions of *MadeCR*: one version uses $N-$gram model and the other uses HMM in its *Anomaly Detector*. For each version, we consider two implementations: a basic implementation that analyzes all the intercepted function calls, and a refined implementation that only analyzes security-critical function calls. In each case, we focus on two key metrics: the accuracy in malware detection and the computation overhead of the detection. We measure the detection accuracy in terms of true positive (TP) and false positive (FP). True positive happens when the malicious traces are detected, while false positive happens when the benign traces are mistakenly classified to be malicious. In general, the higher the TP and the lower the FP are, the better the detection accuracy is. As will be shown later, our results indicate that the refinement approach proposed in this paper can increase TP rate and reduce FP rate at the same time, which are often considered unachievable simultaneously.

To generate normal traces, the original CR application is run for 2000 times in *Application Driver* and each time lasts for 22 seconds. 60% of the generated traces are randomly selected for training, and the rest 40% are used as testing traces to assess the FP rate of *MadeCR*. To obtain the malicious testing traces to evaluate the TP rate, every malicious mutant is executed for 22 seconds in *Application Driver*.

### A. Accuracy

We test both the basic and the refined implementations of the two versions of *MadeCR* with different data modeling methods on the testing traces. The consequent four *MadeCR* implementations are denoted as $N-$gram, $N-$gram refined, HMM, and HMM refined respectively. For each *MadeCR* implementation, different values of the important configuration parameters are set and the impact on the detection accuracy of *MadeCR* is studied. For $N-$gram and $N-$gram refined, the sliding window size $N$ is a particularly important configuration parameter. For HMM and HMM refined, the threshold $P_T$, which is set to differentiate the likelihood of a malicious trace

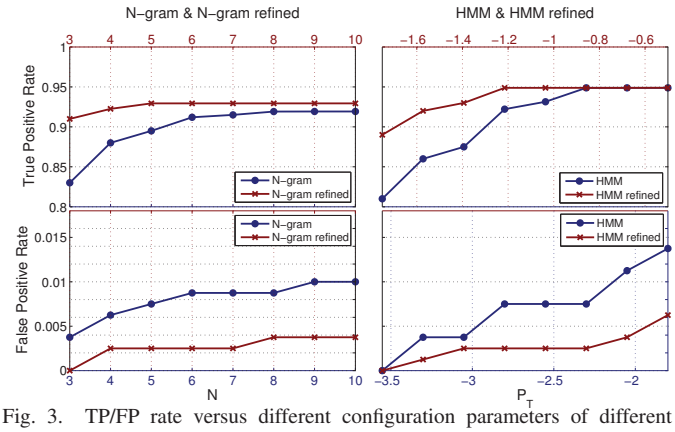


Fig. 3. TP/FP rate versus different configuration parameters of different *MadeCR* implementations ($N$ is the sliding window size in $N-$gram, $P_T$ is the likelihood threshold in HMM)

to be produced by the learnt HMM from that of normal traces, is studied. Because likelihoods and its threshold $P_T$ tend to be very small when traces are long, we present the likelihoods and its threshold $P_T$ in logarithmic form.

Our first experiment examines the relationship between the configuration parameters and the TP/FP rate of each *MadeCR* implementation and the results are shown in Figure 3. Both the TP and FP rates of the two $N-$gram-based implementations increase with $N$. However, the $N-$gram refined is more stable against $N$ compared with $N-$gram. When $N = 5$, the $N-$gram refined reaches the highest TP rate (93.94%) while only incurring 0.25% FP rate. For HMM and HMM refined, as $P_T$ increases, more traces are classified to be malicious, which increases both TP and FP. The highest TF rates of the two HMM-based implementations are both 94.86%, which is better than the two $N-$gram implementations. In comparison, however, HMM refined raises less FP than HMM when they produce the same TP rate. Also, it can be observed that the refinement approach can contribute larger detection accuracy improvement to $N-$gram than to HMM. This is because HMM is a more complex model. Even without refinement, it still has a certain ability level to extract useful information by itself for malware detection. On the other hand, $N-$gram is a simpler approach and does not have the same capability. Hence, $N-$gram has to completely rely on the refinement scheme to prune away useless information. The results shown in Figure 3 can be used to choose proper configuration parameters for each *MadeCR* implementation. In the next two experiments, we use the $N-$gram refined and HMM refined *MadeCR* implementations and the configurations of these two implementations are set at the points that achieve the highest TP rate (i.e. 93.94% and 94.86% respectively).

The second experiment focuses on the mutants generated by the IIL mutation operator, essentially the mutants that mimic suppression attacks. For these mutants, *MadeCR* can produce 96.76% TP rate using $N-$gram refined and 96.82% using HMM refined.

In order to further illustrate the detection capability of *MadeCR* when it is applied to real-world malware, the third experiment generates high order mutants as testing cases by the insertion of two or more mutations to the original program. The real-world malware usually changes the source code at more than one positions to make more significant changes to the execution flow. Similarly, more mutations applied to the program can also introduce more changes to the program's behaviors. In Table II, the TP rate of $N-$gram refined and HMM refined in detecting mutants of different orders are shown. The order of a mutant is defined to be the number of mutations applied to the original program to generate the mutant. 542 2-order mutants and 542 3-order mutants are used as testing cases. As we can see, *MadeCR* can achieve a better detection accuracy for the mutants with more than 1 mutations, thus we believe *MadeCR*'s detection accuracy for real-world malware should also be very high.

TABLE II
TRUE POSITIVE RATE VERSUS MUTATION ORDERS

| Mutation order | 1 | 2 | 3 |
|---|---|---|---|
| $N-$gram refined | 93.94% | 98.52% | 99.63% |
| HMM refined | 94.86% | 99.08% | 99.81% |

### B. Computation Overhead

We evaluate the computation overhead of different *MadeCR* implementations, and especially focus on the overhead reduction introduced by refinement. As shown in Table III, of all the 97 unique functions captured in the training phase, 57 functions are extracted as security-critical functions. The average length of traces intercepted in 22 seconds is 1683, which is reduced to 983 after filtering out the irrelevant functions through the refinement process. For HMM, the number of states is 246 in basic HMM implementation, and it reduces to 109 after refinement.

TABLE III
THE REDUCTION EFFECT OF REFINEMENT

| | Basic | Refined | Percentage reduction |
|---|---|---|---|
| # of unique functions | 97 | 57 | 41.42% |
| Average trace length | 1683 | 983 | 41.59% |
| # of states in HMM | 246 | 109 | 55.70% |

Figure 4 shows the average time to process one trace in the training phase and the detection phase respectively. As we can see, HMM takes much longer time for training while shorter time for detection compared with $N-$gram. The refinement approach can increase the efficiency of both $N-$gram model and HMM. The improvement effect is larger for HMM: The training time is reduced from 182.28 seconds to 18.38 seconds, while detection time is reduced from 4.81 seconds to 1.10 seconds.

## VI. RELATED WORK

The sequence of system calls produced by applications has been widely used in anomaly detection analysis [7]. In [12], Hofmeyr et al. modeled sequences of system calls using $N-$gram model. Mismatches of $N-$grams in testing traces compared to normal traces were used as anomaly score. In [13], Warrender et al. proposed a HMM-based anomaly detection approach, which used the likelihood of testing sequences generated by the learnt HMM and average state transition probability of testing sequences as anomaly scores. In addition, they compared the performance of the HMM-based approach and the $N-$gram-based approach, and found out that the

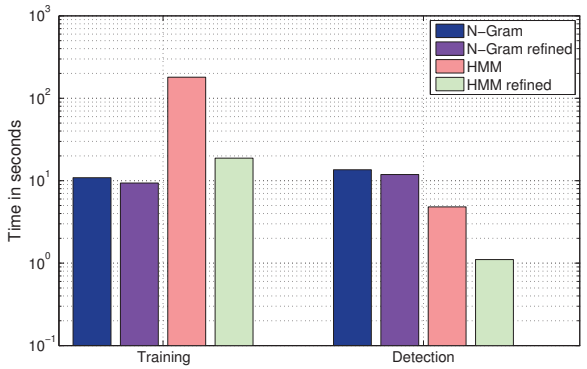


Fig. 4. Performance Evaluation

HMM-based approach gave the best accuracy on average at high computational cost. In [21], Hao et al. analyzed traffic causality and scalable triggering relation to detect stealthy malware activities. In [22], Elish et al. profiled user-trigger dependence to detect malware in Android platform. However, none of these approaches can detect suppression attacks. We extend their works by designing new computational methods for security verification to detect suppression attacks.

In [8], Mutz et al. applied four models to characterize system call arguments and to identify anomalous occurrences. The models focused on analyzing string length, character distribution, string structure and tokens of system call arguments respectively. For each system call, an anomaly score was calculated for the argument in question. However, the important information of correlation relationships between system calls were not included in their work. In [19], Maggi et al. extended the work in [8] by building a behavioral Markov model incorporating both arguments and sequences of system calls. The models proposed in the two papers are not readily applicable to analyze arguments in a CR application because none of them can address continuous numeric values.

In [20], David Wagner and Drew Dean discovered that ignoring security-irrelevant system calls in execution traces can reduce the computation overhead and even increase accuracy of the detection model. However, how to efficiently filter out these certain system calls were not addressed in their work. In our paper, we propose a practical approach to automatically prune away the irrelevant function calls and extract the security-critical ones.

## VII. Conclusion & Future Work

This paper presents *MadeCR*, the first approach to detect malware for CR networks by learning correlations of CR applications' behaviors. *MadeCR* monitors both control flow and data flow during the execution of CR applications to extract useful correlations. *MadeCR* is able to detect suppression attacks, which are significant threats to CR networks. To reduce false positives and computation overhead, we propose a new refinement approach to extract security-critical function calls from all the intercepted function calls. Artificial mutants are generated to test the effectiveness of *MadeCR*.

For future work, we will extend *MadeCR* from only monitoring CR applications' operation events at CR software to multiple-layer, such as radio hardware, operating system, user application, and network. A multi-layer detection will be more effective to capture malware.

## References


[1] R. Falk, J. Esfahani, and M. Dillinger, "Reconfigurable radio terminals—Threats and security objectives," in *SDR Forum Input Document, SDRF-02-I-0056*, 2002.

[2] M. Kurdziel, J. Beane, and J. Fitton, "An SCA security supplement compliant radio architecture," in *Military Communications Conference, MILCOM 2005. IEEE*, pp. 2244–2250.

[3] W. Scott, A. Houle, and A. Martin, "Information assurance issues for an SDR operating in a manet network," in *SDR Forum, November*, 2006.

[4] T. Ulversoy, "Software defined radio: Challenges and opportunities," *Communications Surveys & Tutorials, IEEE*, vol. 12, no. 4, pp. 531–550, 2010.

[5] J. L. Burbank, "Security in cognitive radio networks: The required evolution in approaches to wireless network security," in *3rd International Conference on Cognitive Radio Oriented Wireless Networks and Communications, CrownCom 2008.* IEEE, pp. 1–7.

[6] R. Chen, J.-M. Park, Y. T. Hou, and J. H. Reed, "Toward secure distributed spectrum sensing in cognitive radio networks," *Communications Magazine, IEEE*, vol. 46, no. 4, pp. 50–55, 2008.

[7] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computing Surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[8] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 9, no. 1, pp. 61–93, 2006.

[9] C. Li, N. K. Jha, and A. Raghunathan, "Secure reconfiguration of software-defined radio," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. 1, p. 10, 2012.

[10] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *Usenix Security*, 1998.

[11] T. Garfinkel, M. Rosenblum *et al.*, "A Virtual Machine Introspection Based Architecture for Intrusion Detection." in *NDSS*, vol. 3, 2003, pp. 191–206.

[12] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[13] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy, S&P 1999.* IEEE, pp. 133–145.

[14] R. Sibson, "SLINK: an optimally efficient algorithm for the single-link cluster method," *The Computer Journal*, vol. 16, no. 1, pp. 30–34, 1973.

[15] S. Salvador and P. Chan, "Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms," in *16th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2004.* IEEE, pp. 576–584.

[16] E. Alpaydin, *Introduction to machine learning*. MIT press, 2004.

[17] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[18] T. R. Newman, A. He, J. Gaeddert, B. Hilburn, T. Bose, and J. H. Reed, "Virginia tech cognitive radio network testbed and open source cognitive radio framework," in *5th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, TridentCom 2009.* IEEE, pp. 1–3.

[19] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 381–395.

[20] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy, S&P 2001*. IEEE, pp. 156–168.

[21] H. Zhang, D. D. Yao, and N. Ramakrishnan, "Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014, pp. 39–50.

[22] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang, "Profiling User-Trigger Dependence for Android Malware Detection," *Computers & Security*, 2014.