




BLACKBOX ANDROID MALWARE DETECTION USING MACHINE LEARNING AND EVASION ATTACKS TECHNIQUES

Professor Dr. Răzvan Bocu
Transilvania University of Brasov//Siemens Industry Software, Romania
razvan@bocu.ro





Many thanks to our sponsors and partners!

Powered by



PLATINUM SPONSORS



HACKING VILLAGE PARTNERS



SILVER SPONSORS



MOBILITY PARTNER



TOYOTA Cluj-Napoca prin Profi Auto

COMMUNITY & MEDIA PARTNERS



DIRECTORATUL NAȚIONAL DE SECURITATE CIBERNETICĂ



UNIVERSITATEA BABES-BOLYAI
BABES-BOLYAI TUDOMÁNYEGYETEM
BABES-BOLYAI UNIVERSITÄT
BABES-BOLYAI UNIVERSITY
TRADITIO ET EXCELLENTIA



BRCC | British Romanian Chamber of Commerce



ȘCOALA INFORMALĂ DE IT®



About...

- Professor in the Department of Mathematics and Computer Science, Transilvania University of Brasov, Romania
- Didactic (both BSc and MSc levels) and research duties, which pertain to machine learning and artificial intelligence, complex networks, cloud and distributed infrastructures, software systems engineering, bioinformatics.

About... (cont'd)

- Scientific Researcher in the Department of Research and Technology, Siemens Industry Software, Brasov, Romania.
- In this capacity, I oversee and I am involved in various research-related activities with strategic and business value.

Context

- Over the past ten years, researchers have extensively explored the vulnerability of Android malware detectors to adversarial examples through the development of evasion attacks. Nevertheless, the feasibility of these attacks in real-world use case scenarios is debatable.
- Most of the existing published papers are based on the assumptions that the attackers know the details of the target classifiers used for malware detection. Nevertheless, in reality, malicious actors have limited access to the target classifiers.
- This proposed talk presents a problem-space adversarial attack designed to effectively evade blackbox Android malware detectors in real-world use case scenarios. The proposed approach constructs a collection of problem-space transformations derived from benign donors that share opcode-level similarity with malware applications through the consideration of an n -gram-based approach.

Context

- These transformations are then used to present malware instances as legitimate entities through an iterative and incremental manipulation strategy.
- The proposed presentation will describe a manipulation model that is based on a query-efficient optimization algorithm, which can identify and implement the required sequences of transformations into the malware applications. The model has already been evaluated relative to more than 1,000 malware applications. → Proves effectiveness of the reported approach relative to the generation of real-world adversarial examples in both software and hardware-related scenarios.
- The experiments that we conducted demonstrate that the proposed model may effectively trick various malware detectors into believing that malware entities are legitimate. More precisely, the proposed model generates evasion rates of 90%–95% relative to data sets like DREBIN, Sec-SVM, ADE-MA, MaMaDroid, and Opcode-SVM.

Context

- The average number of required computational operations belongs to the range [1..7]. Additionally, it is relevant to note that the proposed adversarial attack preserves its stealthiness against the virus detection core of three popular commercial antivirus softwares.
- The obtained evasion rate is 87%, which further proves the proposed model's relevance for real-world use case scenarios.

Contribution

- We propose a comprehensive and generalized evasion attack, which can bypass blackbox Android malware classifiers through a two-step process: (i) *preparation* and (ii) *manipulation*. The first step involves implementing a donor selection technique to create an action set comprising a collection of problem-space transformations → this relates to code snippets known as *gadgets*.
- These gadgets are derived by conducting program slicing on benign apps, known as donors, which are publicly available. By injecting each gadget into a malware app, specific payloads from a benign donor can be incorporated into the malware app.
- The proposed technique utilizes an *n-gram-based similarity(sequence of n adjacent symbols in a particular order)* method to identify suitable donors, particularly benign apps that exhibit similarities to malware apps at the opcode level(**specifies operation to be executed**). Applying transformations derived from these donors to malware apps can enable them to appear legitimate(benign), or move them towards blind spots of ML classifiers.

Contribution

- This approach aims to achieve the desired outcome of introducing transformations that not only ensure adherence to problem-space constraints(**preserved semantics, robustness to preprocessing, and plausibility**), but also possibly lead to malware classification errors.
- We propose a *blackbox evasion* attack that generates real-world Android Adversarial Attacks(AE-Adversarial Examples) that adhere to problem-space constraints. To the best of our knowledge, this is one of the few studies in the Android scope that successfully evades ML-based malware detectors by effectively manipulating malware samples without performing feature-space perturbations.
- We demonstrate this is a *query-efficient* attack capable of deceiving various blackbox ML-based malware detectors through minimal querying. Thus, our proposed problem-space adversarial attack achieves evasion rates of 92%, 88%, 89%, 98%, and 84% against DREBIN (Arp et al., 2014), Sec-SVM (De-montis et al., 2017), ADE-MA (Li and Li, 2020), MaMaDroid (On-wuzurike et al., 2019), and Opcode-SVM (Jerome et al., 2014), respectively.

Contribution

- Our proposed attack can operate with either *soft labels* (confidence scores), or *hard labels* (classification labels) of malware apps, as specified by the target malware classifiers, to generate AEs.
- We assess the practicality of the proposed evasion attack under real-world constraints by evaluating its performance in deceiving popular commercial antivirus products. Specifically, our findings indicate that proposed approach may significantly diminish the effectiveness of three popular commercial antivirus products, achieving an average evasion rate of approximately 86%.

Android application package (APK)

- APK is a compressed file format with a *.apk* extension. APKs contain various contents such as Resources and Assets. However, the most crucial contents, particularly for malware detectors, are the Manifest (AndroidManifest.xml), and Dalvik bytecode (classes.dex). The Manifest is an XML file that provides essential information about Android apps, including the package name, permissions, and definitions of Android components.
- It contains all the metadata required by the Android OS to install and run Android apps. On the other hand, Dalvik bytecode, also known as Dalvik Executable or DEX file, is an executable file that represents the behavior of Android apps.
- *Apktool* is a popular reverse engineering tool for the static analysis of Android applications. This reverse engineering instrument can decompile and recompile Android apps.
- During the decompilation process, the DEX files of Android apps are compiled into a human-readable code called *smali*(<https://github.com/JesusFreke/smali>). Also, *Soot*(<https://soot-oss.github.io/soot/>) and *FlowDroid*(<https://github.com/secure-software-engineering/FlowDroid>) are two Java-based frameworks that are used for analyzing Android apps. Soot extracts different information from APKs, which are then used during static analysis. One of the advantages of Soot for malware detection is its ability to generate call graphs. Soot cannot generate accurate call graphs for all apps because of the complexity of the control flow of some APKs.

Android application package (APK)

- Proposed approach can create precise call graphs based on the application's life cycle. It is worth noting that it uses Apktool, FlowDroid, and Soot in different components of its pipeline to generate adversarial examples.

Machine Learning Android malware detection

- ML has demonstrated its potential as an effective solution in static malware analysis, enabling the identification of sophisticated and previously unknown malware through the generalization capabilities of ML algorithms.
- It is important to note that static analysis is a prominent approach for detecting malicious programs, where apps are classified based on their source code (aka, static features) without execution. This approach offers fast analysis, allowing for the examination of an app's code comprehensively, with minimal resource usage in terms of memory and CPU.
- Various types of features are considered in the static analysis, including syntax features (e.g., requested permissions and API calls), opcode features (e.g., n-gram opcodes), image features (e.g., grayscale representations of bytecodes), and semantic features (e.g., function call graphs).

Adversarial transformations

- Considering the programming domain, a safe transformation refers to a problem-space transformation that maintains the semantic equivalence of the original program while ensuring its executability. In the adversarial malware domain, safe transformations, which guarantee preserved-semantics constraint, can become adversarial transformations if they are also plausible and robust to processing (refer to Appendix A for additional details regarding these constraints).
- Generally, in the context of Android malware detection, attackers have three types of adversarial transformations at their disposal to manipulate malicious apps: (i) *feature addition*, (ii) *feature removal*, and (iii) *feature modification*. Feature addition involves adding new elements, such as API calls, to the programs, while feature removal entails removing contents like user permissions.

Adversarial transformations

- Feature modification combines both addition and removal transformations in malware programs. Most studies have primarily focused on feature addition, as removing features from the source code is a complex operation that may cause malware apps to crash.
- Code transplantation, system-predefined transformation, and dummy transformation are three potential methods for adding features to manipulate Android apps. However, two main issues arise when considering feature additions:
- **(i) What specific content should be included.** By deriving problem-space transformations from feature-space perturbations, the attacker aims to ensure that the additional contents (e.g., API calls, Activities, etc.) are guaranteed to appear in the feature vector of the manipulated malware app. Therefore, attackers may either use dummy contents (e.g., functions, classes, etc.), or system-predefined contents for this purpose. As the plausibility of these transformations is debatable, malicious actors may also make use of content present in already existing Android apps. The ***automated software transplantation*** technique can then be used to allow attackers to successfully carry out safe transformations. They extract some slices of existing bytecodes from benign donor apps during the *organ harvesting* phase, and the collected payloads are injected into malware apps in the *organ transplantation* phase.

Adversarial transformations

- **(ii) Where contents should be injected.** New contents must preserve the semantics of malware samples. Therefore, they should be injected into areas that cannot be executed during runtime. As an example, new contents can be added after RETURN instructions, or inside an IF statement that is always false. However, these injected contents are not robust to preprocessing actions, **if** static analysis can discard unreachable code. One creative idea to add unreachable code that is undetectable is the use of *opaque predicates* (Moser et al., 2007). Thus, new contents are injected inside an IF statement where its outcome can only be determined at runtime.

Proposed attack model

- **Adversarial Goal.** The purpose is to manipulate Android malware samples in order to deceive static ML-based Android malware detectors. The proposed attack is an untargeted attack (Carlini et al., 2019) designed to mislead binary classifiers considered in Android malware detection, causing Android malware apps to be misclassified. More precisely, the objective is to trick malware classifiers into classifying malware samples as benign.
- **Adversarial Knowledge.** The proposed evasion attack has blackbox access to the target malware classifier. Therefore, it does not have knowledge of the training data \mathcal{D} , the feature set \mathcal{X} , or the classification model f , more precisely to the classification algorithm and its hyperparameters. The attacker can only obtain the classification results (hard labels or soft labels) by querying the target malware classifier.

Proposed attack model

- **Adversarial Capabilities.** This is designed to deceive blackbox Android malware classifiers during their prediction phase. Our attack manipulates an Android malware app by applying a set of safe transformations, known as Android gadgets (slices of the benign apps' bytecode), which are optimized through interactions with the blackbox target classifier. To ensure adherence to problem space constraints, this leverages a tool developed by the authors of paper (Pierazzi et al., 2020), for extracting and injecting gadgets. Furthermore, in order to avoid major disruptions to apps, the manipulation process of a malware app is conducted gradually, making it resemble benign apps. This is achieved by injecting a minimal number of gadgets extracted from benign apps into the malware app, and the process continues until the malware app is misclassified or reaches the predefined evasion cost. In addition to the problem-space constraints discussed in previous research (Pierazzi et al., 2020), our model must also adhere to two additional constraints highlighting the significance of minimizing evasion costs:

Proposed attack model

- **Number of queries.** Proposed approach is a decision-based adversarial attack that aims to generate AEs while minimizing the number of queries, thus reducing the associated costs.
- **Size of adversarial payloads.** In order to generate executable and visually inconspicuous AEs, such as those with minimal file size (Demetrio et al., 2021), proposed approach aims to minimize the size of injected adversarial payloads.
- It is relevant to note that each gadget consists of an organ, which represents a slice of program functionality, an entry point to the organ, and a vein, which represents an execution path that leads to the entry point (Pierazzi et al., 2020). Proposed model extracts gadgets from benign apps by identifying entry points, which are typically API calls, through string analysis. The proposed attack assumes that the benign apps used for gadget extraction are not obfuscated, particularly in terms of their API calls.

Proposed attack model

- This is because the proposed model relies on string analysis to identify entry points, which limits its ability to extract gadgets from obfuscated apps. The gadget injection is **considered successful** when both the classification loss value of the manipulated app increases, and the injected adversarial payload conforms to the predefined size of the adversarial payload. Additionally, the injected gadgets are placed within the block of an obfuscated condition statement that is always evaluated as False during runtime, and cannot be analyzed during early preprocessing stages.
- **Defender's Capabilities.** We assume that the target ML models do not employ adaptive defenses that are aware of the operations performed by proposed model due to disclosing detectors' vulnerability to the detection core of our proposed model. Specifically, these target models are unable to enhance their resilience by incorporating AEs generated by proposed model during adversarial training. Furthermore, they lack the capability to detect and block queries from proposed model, **if** they become suspicious of its origin. Additionally, our analysis suggests that proposed model can still be effective, even if we relax the second assumption regarding the defender's capabilities. This is supported by empirical evidence demonstrating that our attack often requires only a minimal number of queries to generate AEs.

Methodology

- The primary goal of proposed model is to transform a malware app into an adversarial app in such a way that it retains its malicious behavior, but is no longer classified as malware by ML-based malware detectors. This is achieved through an iterative and incremental algorithm used in the proposed attack, which aims to disguise malware APKs as benign ones.
- The attack algorithm generates real-world AEs from malware apps using *problem-space transformations* that satisfy problem space constraints.
- These transformations are extracted from benign apps in the wild, which are similar to malware apps using an *n-gram*-based similarity model.

Methodology

- In this approach, a *random search* algorithm is used to optimize the manipulations of apps.
- Each malware app undergoes incremental tweaking during the optimization process, where a sequence of transformations is applied over different iterations.
- These transformations are extracted from benign apps in the wild, which are similar to malware apps using an *n-gram*-based similarity model.

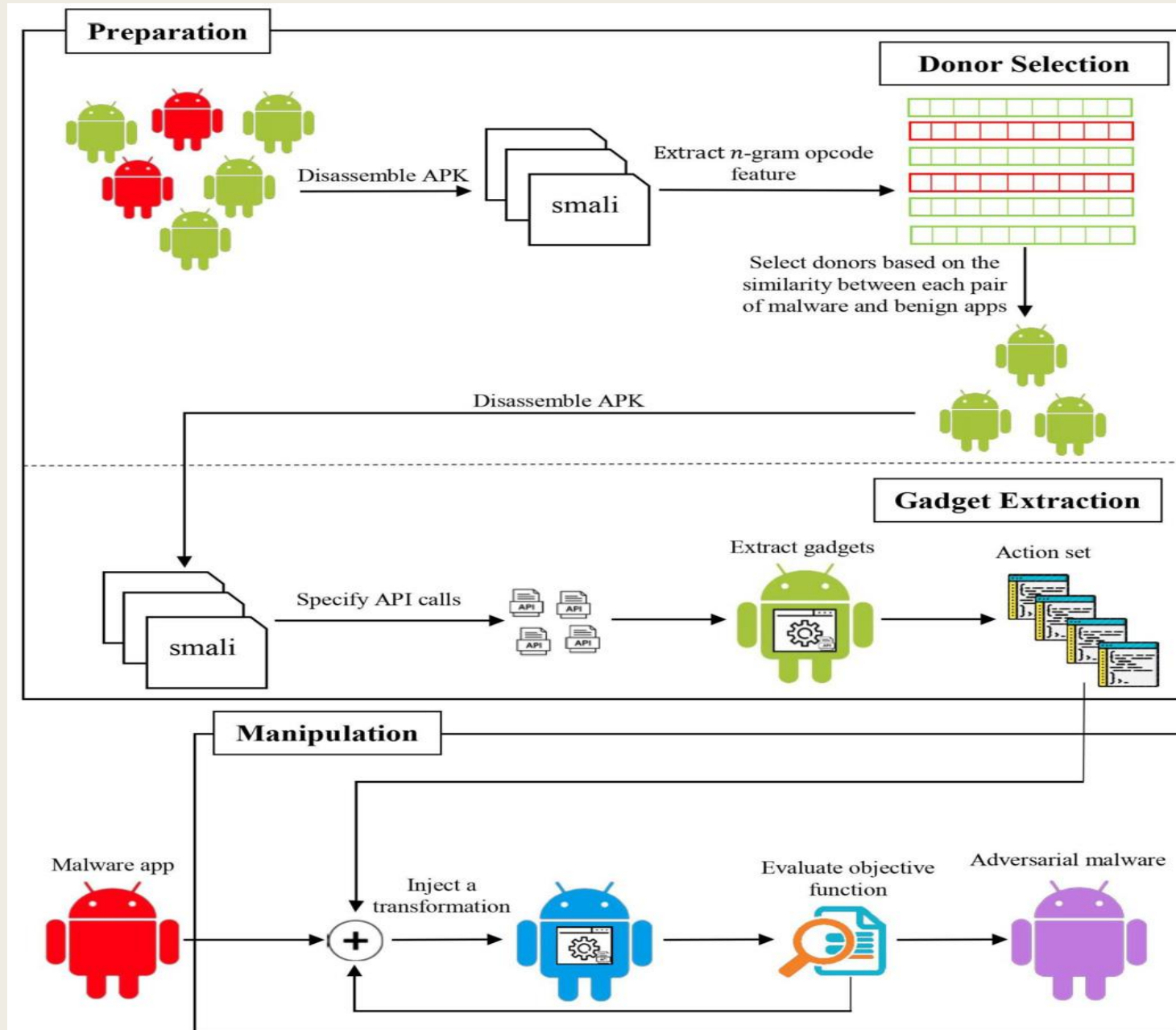
N-Grams

- ***n*-Grams** are contiguous overlapping sub-strings of items (e.g., letters or opcodes) with a length of n from the given samples (texts or programs). This technique captures the frequencies or existence of a unique sequence of items with a length of n in a given sample. In the area of malware detection, several studies considered *n*-grams to extract features from malware samples, such as (Fuyong and Tiezhu, 2017; Jain and Meena, 2011; Moskovitch et al., 2008; Santos et al., 2013; Shabtai et al., 2012).
- These features can be either byte sequences extracted from binary content or opcodes extracted from source codes. *n*-Grams opcode analysis is one of the static analysis approaches for detecting Android malware that has been investigated in various related works, such as (Canfora et al., 2015; Islam et al., 2020; Jerome et al., 2014; Mas'ud et al., 2016; Varsha et al., 2017). To conduct such an analysis, the DEX file of an APK is disassembled into smali files. Each smali file corresponds to a specific class in the source code of the APK that contains variables, functions, etc. *n*-Grams are extracted from the opcode sequences that appear in different functions of the smali files.

Random Search

- **Random Search (RS)** (Rastrigin, 1963) is a simple yet highly exploratory search strategy that is used in some optimization problems to find an optimal solution. It relies entirely on randomness, which means RS does not require any assumptions about the details of the objective function or transfer knowledge (e.g., the last obtained solution), going from one iteration to another

Workflow of the attack pipeline



Preparation

- The primary objective of this step is to construct an action set comprising a collection of safe transformations that can directly manipulate Android applications. Each transformation in the action set should be capable of altering APKs without causing crashes while preserving their functionality.
- Proposed model uses program slicing (Weiser, 1984), also implemented in Pierazzi et al. (2020), to extract the gadgets that make up the transformations collected in the action set. During the preparation step, two important considerations are determining appropriate donors and identifying suitable gadgets.
- Employing effective gadgets enables the modification of a set of features that can alter the classifier's decision. Steps: *donor selection*, and *gadget extraction* (previous slide 24).

Manipulation

- Proposed approach uses Random Search (RS) as a blackbox optimization method. Specifically, for each malware sample z , proposed model utilizes RS to find an optimal subset of transformations \mathcal{D} in order to generate an adversarial example z^* .
- RS offers a significant advantage in terms of query reduction compared to other heuristic optimization algorithms, such as Genetic Algorithms (GA). This is true because RS only requires one query in each iteration to evaluate the current solution.

Generating an adversarial example

Input: z , the original malware sample; Δ , the action set; L , the objective function; ϕ , the feature mapping function; c , the payload-size cost function; Q , the query budget; α , the allowed adversarial payload size.

Output: z^* , an adversarial example; δ , an optimal transformations.

```
1  $q \leftarrow 1$  ;
2  $z^* \leftarrow z$ ;
3  $L_{best} \leftarrow -\infty$ ;
4  $\delta \leftarrow \emptyset$ ;
5 while  $q \leq Q$  and  $z^*$  is classified as a malware do
6      $\lambda \leftarrow$  Select a transformation randomly from  $\Delta \setminus \delta$ ;
7      $z' \leftarrow T_\lambda(z^*)$ ;
8      $l = L(\phi(z'))$ ;
9     if  $c(z, z') \leq \alpha$  then
10         if  $L_{best} \leq l$  then
11              $L_{best} \leftarrow l$ ;
12              $z^* \leftarrow z'$ ;
13              $\delta \leftarrow \delta \cup \lambda$ 
14         end
15     end
16 end
17 return  $z^*, \delta$ 
```

Research questions for the experimental process

- **RQ1.** How does the evasion cost affect the performance of proposed model?
- **RQ2.** Does proposed approach constitute a versatile attack that can evade different Android malware detectors without relying on any specific assumptions?
- **RQ3.** How does the performance of proposed model compare to other similar attacks?
- **RQ4.** Is proposed model applicable in real-world use case scenarios?
- **RQ5.** How does proposed model demonstrate its performance considering the restriction of not being able to query the target detectors?
- **RQ6.** How does the proposed RS-based manipulation strategy affect the performance of proposed model?
- **Experimental setup:** AlmaLinux 9.3 workstation, AMD Ryzen 9 5950X CPU, 128 GB RAM.

Target detectors

- To ensure that our conclusions are not limited to a specific type of malware detection, we evaluate proposed model against various malware detectors to demonstrate the effectiveness of the proposed attack patterns. In particular, our evaluation focuses on assessing the performance against well-known Android malware detection models, such as DREBIN (Arp et al., 2014), Sec-SVM (Demontis et al., 2017), ADE-MA (Li and Li, 2020), MaMaDroid (Onwuzurike et al., 2019), and Opcode-SVM (Jerome et al., 2014).
- These models have been extensively studied in the context of detecting problem-space adversarial attacks in the Android domain, see papers (Chen et al., 2019; Grosse et al., 2017; Li et al., 2021b; Pierazzi et al., 2020; Zhang et al., 2021).

Considered data

- We evaluate the performance of proposed model using the dataset mentioned by Pierazzi et al. (2020). This dataset consists of $\approx 170K$ samples, each represented using the DREBIN (Arp et al., 2014) feature set.
- The samples are feature representations of Android apps collected from AndroZoo (<https://androzoo.uni.lu/>) and labeled by Pierazzi et al. (2020) using a threshold-based labeling approach. These collected apps were published between January 2017 and December 2018. According to the labeling criteria in Pierazzi et al. (2020), an APK is considered malicious or clean if it has been detected by any 4+ or 0 VirusTotal (VT) (<https://www.virustotal.com/gui/home/upload>) engines, respectively.
- It is important to note that the threshold-based labeling approach does not rely on specific engines, but instead considers the number of engines involved. Therefore, the engines used for labeling may vary from sample to sample.

Considered data

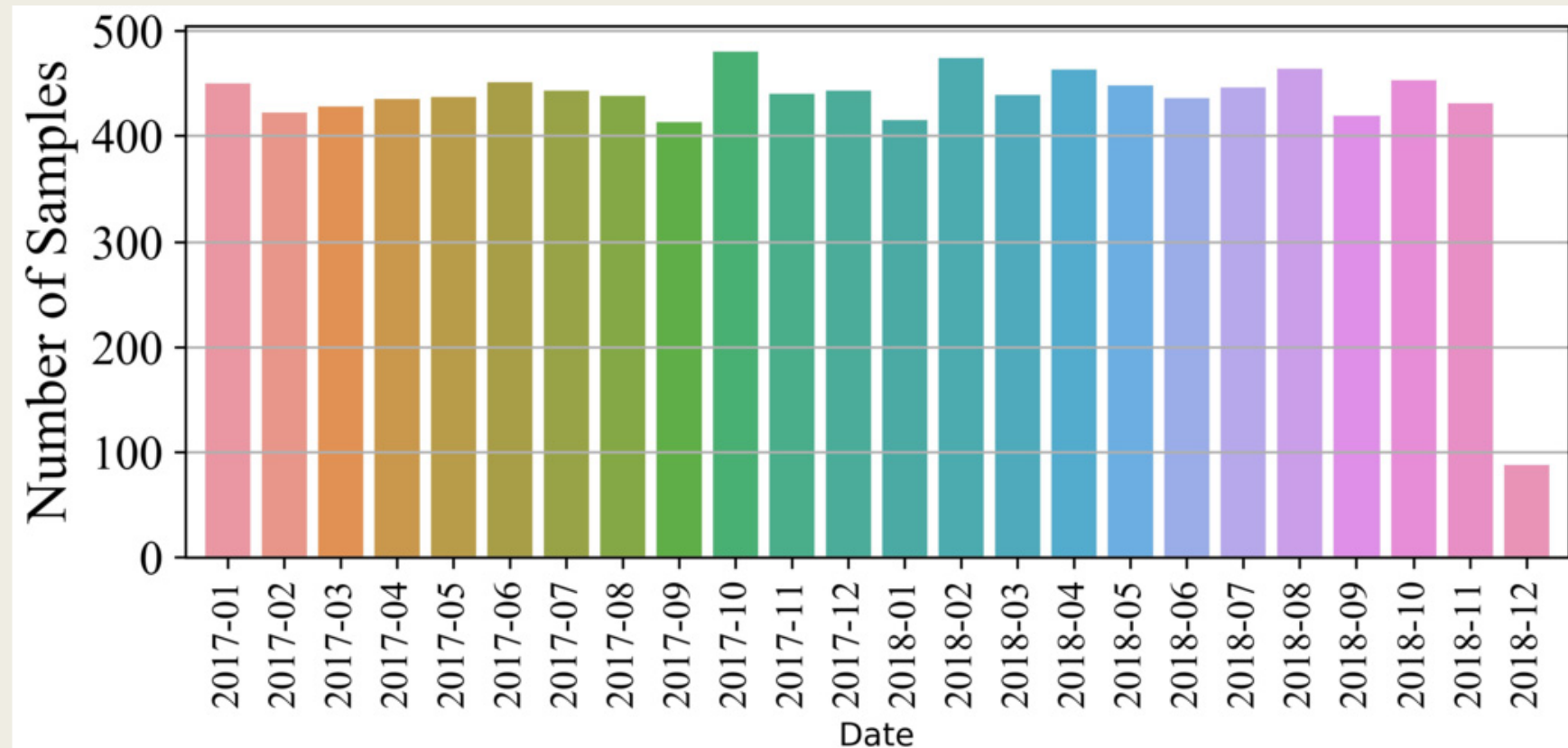
Dataset	# of benign samples	# of malware samples
Inaccessible Dataset	10,000	2,000
Training Samples	90,000	10,000
Dataset used to tweak proposed model	20,000	10,000

Considered data

- **There is no overlap between the inaccessible and accessible datasets.** Proposed model exclusively makes use of the available dataset, which comprises 20,000 benign samples for donor selection, and 10,000 malware samples for the creation of AEs. To fulfill the requirement of direct utilization of apps in our problem-space attack, we collect 30,000 apps corresponding to proposed model accessible samples from AndroZoo, based on the apps' specifications provided with the dataset (Pierazzi et al., 2020).
- In this study, we employ two training sets with different scales (12,000 and 100,000) for training classifiers. The proportion between benign and malware samples in the training sets is chosen to avoid spatial dataset bias.
- A training set with a reasonable size (12,000) is used due to the time-consuming preprocessing required by the apps in the MaMaDroid and Opcode-SVM, particularly the former.

Considered data

- Temporal distribution of the smaller training set, demonstrating the absence of temporal bias as these apps were published across various months. The larger training set follows a similar distribution.



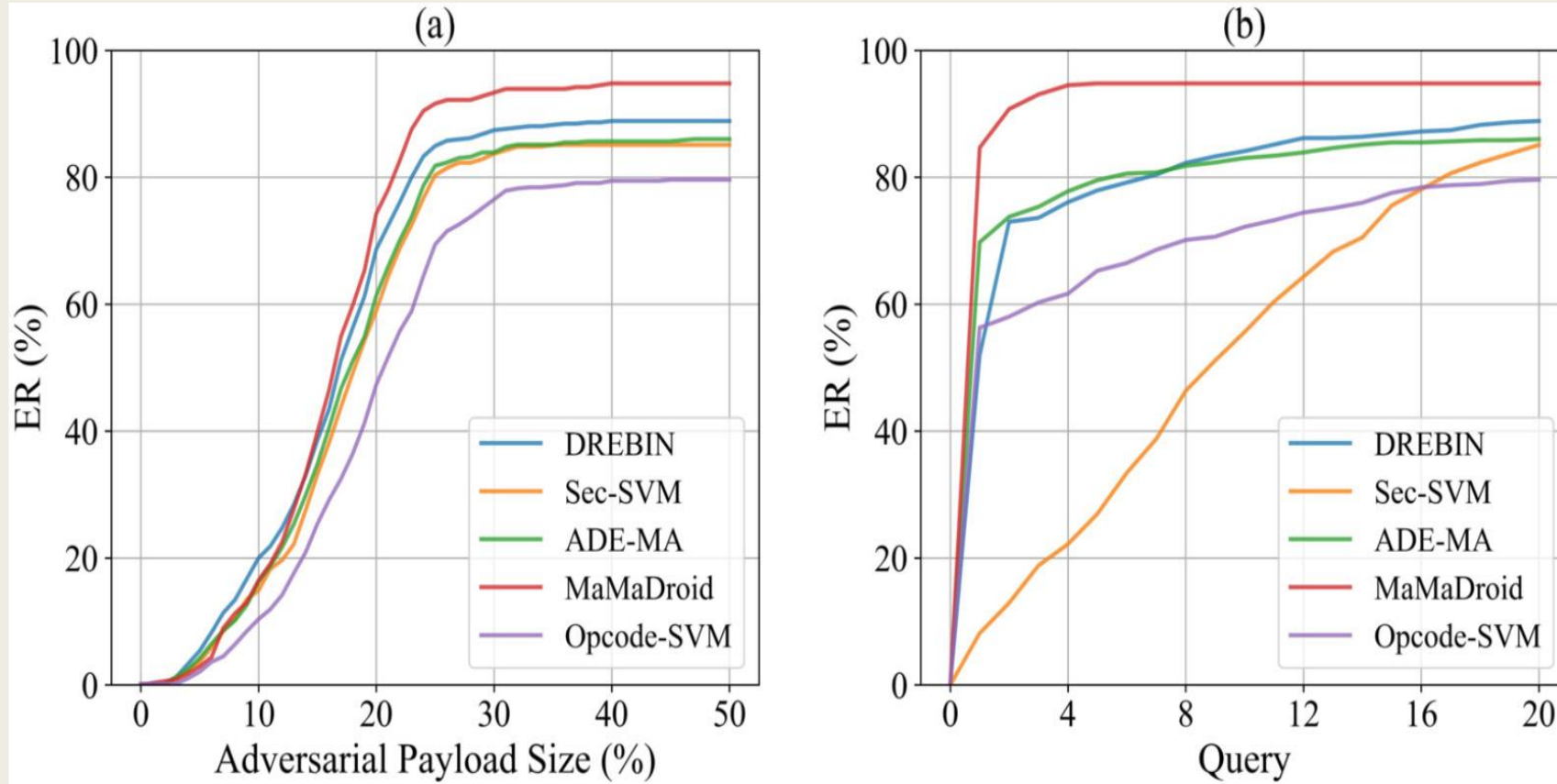
Considered data

- Note that MaMaDroid and Opcode-SVM employ their own distinct feature representations, which differ from the DREBIN feature representation used in Pierazzi et al. (2020). Therefore, to provide the training set for these detectors, we have to directly collect all considered apps in the training set from AndroZoo based on the specifications provided by Pierazzi et al. (2020).

Experimental metrics

- We consider the True Positive Rate (TPR) and False Positive Rate (FPR) as performance metrics for evaluating the effectiveness of malware classifiers in detecting Android malware.
- Additionally, we consider the Evasion Rate (ER) and Evasion Time (ET) as proposed model's performance assessment metrics in deceiving malware classifiers.
- ER is calculated as the ratio of correctly detected malware samples that are able to evade the target classifiers after manipulation, relative to the total number of correctly classified malware samples.
- ET represents the average time, expressed in seconds, required by proposed model to generate an AE, encompassing both the optimization and query times.
- The optimization time primarily consists of the execution times of random search, injecting problem-space transformations, and performing feature extraction to represent manipulated apps within the feature space.

RQ1. The evasion rates of proposed approach in fooling various malware detectors under different adversarial payload sizes and query numbers



RQ2. Effectiveness of proposed model in misleading different malware detectors. NoQ, NoT, and AS denote Avg. No. of Queries, Avg. No. of Transformations, and Avg. Adversarial Payload Size, respectively.

Type of Threat	Target Model	ER (%)	ET (s)	NoQ	NoT	AS (%)
Soft Label	DREBIN	88.9	210.3	3	2	15.5
	Sec-SVM	85.1	495.4	9	4	16.4
	ADE-MA	86.0	126.2	2	1	16.3
	MaMaDroid	94.8	131.4	1	1	15.9
	Opcode-SVM	79.6	114.1	3	2	18.3
Optimal Hard Label	DREBIN	84.5	240.6	4	2	16.2
	Sec-SVM	82.6	613.1	9	6	16.5
	ADE-MA	84.4	121.2	2	1	16.3
	MaMaDroid	94.8	133.7	1	1	15.9
	Opcode-SVM	74.1	101.2	2	1	18.2
Non-optimal Hard Label	DREBIN	79.7	357.2	4	4	16.9
	Sec-SVM	78.2	782.8	9	9	17.3
	ADE-MA	82.7	157.3	2	2	16.4
	MaMaDroid	94.8	132.6	1	1	15.9
	Opcode-SVM	66.6	76.2	1	1	18.3

Comments

- Data in previous table demonstrate a 15% improvement in the evasion rate(ER) of proposed model when considering Opcode-SVM in the soft-label setting, compared to the non-optimal hard-label setting. Furthermore, when operating in the soft-label setting, proposed model requires notably fewer transformations to bypass DREBIN and Sec-SVM, as compared to the non-optimal hard-label setting , which confirms the effectiveness of described approach in solving the optimization problem.
- Table further illustrates that our optimization leads to a substantial reduction in evasion time(ET) compared to the non-optimal hard-label setting. Specifically, for DREBIN and Sec-SVM, this leads to a time reduction of $\approx 41\%$ and $\approx 37\%$, respectively. This significant enhancement can be attributed to the reduction in the number of transformations, achieved through the utilization of our proposed optimization model.
- In short: the results demonstrate that the proposed adversarial attack is a versatile blackbox attack that does not make assumptions about target detectors, including the ML algorithms or the features used for malware detection. As a consequence, it can operate effectively in various attack settings.

RQ3. Proposed model vs. other attacks

- We conduct an empirical analysis to assess how the proposed attack pattern performs in comparison to other similar attacks.
- We consider four baseline attacks: PiAttack (Pierazzi et al., 2020), Sparse-RS (Croce et al., 2022), ShadowDroid (Zhang et al., 2021), and GenDroid (Xu et al., 2023) operating in whitebox, graybox, semi-blackbox, and blackbox settings, respectively.
- These attacks serve as suitable benchmarks, allowing us to assess the performance of described approach from different perspectives, such as evasion rate and the number of queries.
- Similar to our model, Sparse-RS, ShadowDroid, and GenDroid generate AEs by querying the target detectors.
- Additionally, PiAttack is a problem-space adversarial attack that employs a similar type of transformation to generate AEs. Although PiAttack is a whitebox evasion attack, it establishes a benchmark for optimal evasion performance, facilitating the evaluation of the comparative effectiveness of other attacks with limited or zero knowledge about the targeted detectors.

RQ3. Proposed model vs. other attacks

- We selected DREBIN, Sec-SVM, and ADE-MA as the target detectors because they align with the threat models of PiAttack, Sparse-RS, and ShadowDroid.
- Although our model has zero knowledge about DREBIN, Sec-SVM, and ADE-MA, its evasion rates for bypassing these detectors are comparable to PiAttack, where the adversary has full knowledge of the target detectors.
- Our empirical analysis shows that the reported model requires adding more features to evade DREBIN, Sec-SVM, and ADE-MA. Concretely, on average, proposed model makes 54–90 new features appear in the feature representations of the malware apps when it applies transformations to the apps for evading DREBIN, Sec-SVM, and ADE-MA, while the transformations used by PiAttack, on average, trigger 11–68 features.
- Reference attack's ability to add a smaller number of features is attributed to its complete knowledge of the details of DREBIN, Sec-SVM, and ADE-MA, while our model lacks this specific information.

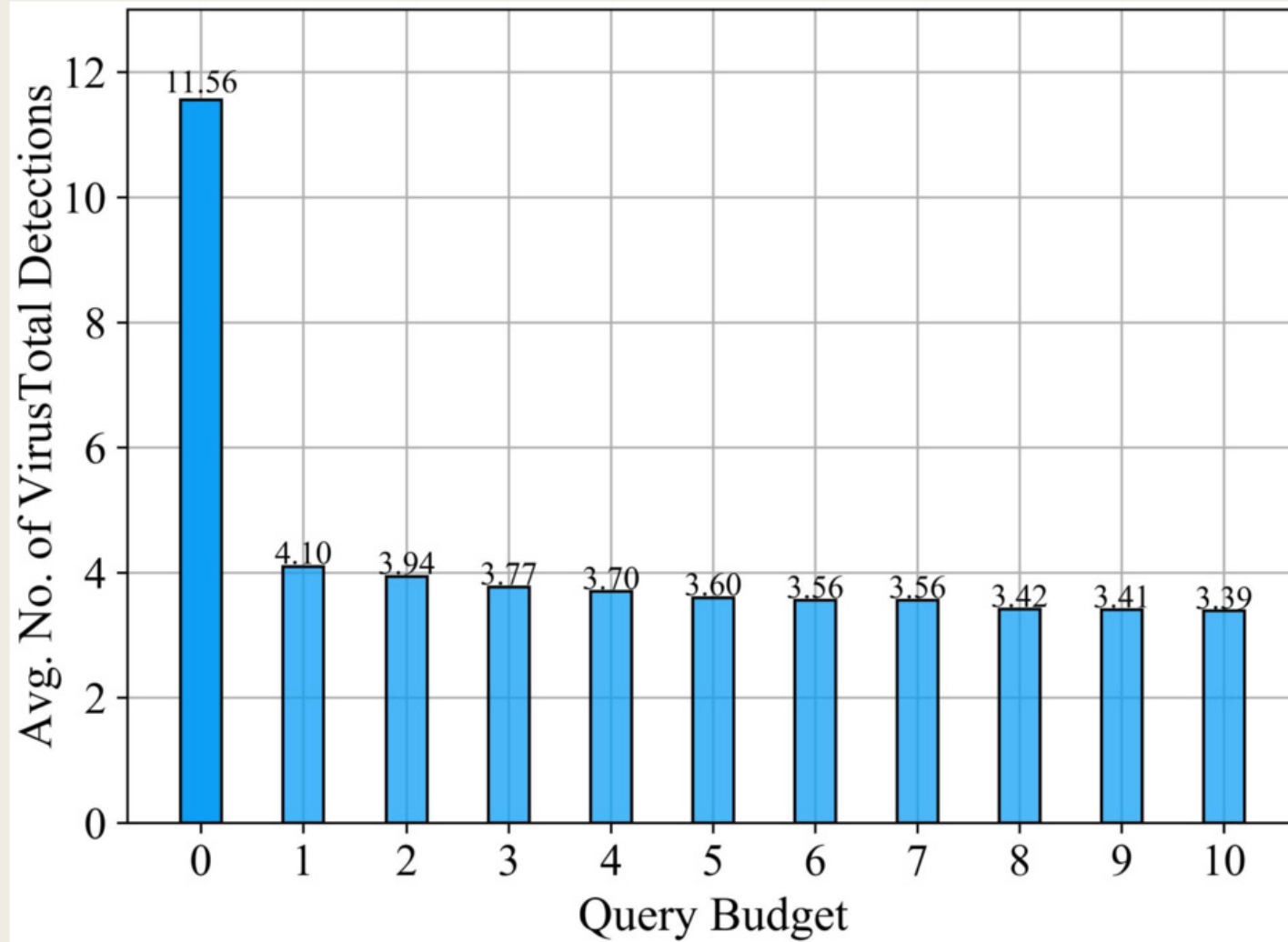
RQ3. Proposed model vs. other attacks

- The evasion rate of Sparse-RS for DREBIN and Sec-SVM demonstrates that random alterations in malware features do not necessarily result in the successful generation of AEs, even when adversaries have access to the target models' training set. Although proposed model operates solely in a blackbox setting, this attack outperforms Sparse-RS by a considerable margin for both DREBIN and Sec-SVM, i.e., 74.8% and 89.8% improvement, respectively. Moreover, it considerably surpasses ShadowDroid (<https://diaowenrui.github.io/paper/icpads21-zhang.pdf>) in attacking Sec-SVM and ADE-ME. In contrast to our model, ShadowDroid is unsuccessful in effectively evading Sec-SVM, which is a robust detector against AEs.
- GenDroid exhibits superior evasion rates compared to our model when targeting DREBIN and ADE-MA; nevertheless, its efficacy is substantially nullified when facing Sec-SVM, a resilient malware detector. Our empirical analysis also highlights the remarkable efficiency of our model in terms of the number of queries compared to other query-based attacks.
- Specifically, on average, our model requires only 1–7 queries to bypass DREBIN, Sec-SVM, and ADE-MA, while Sparse-RS, ShadowDroid, and GenDroid demand 2–195, 29–64, and 81–336 queries, respectively.

RQ4. Real-world effectiveness

- We selected three popular antivirus engines in the Android ecosystem based on the recent ratings of the endpoint protection platforms reported by AV-Test[Total AV, AVG, Avast] (<https://www.av-test.org/en/>).
- Our proposed attack pattern can effectively evade all antivirus products with a few queries. Thus, the effectiveness of our model can be primarily attributed to the transformations rather than the optimization technique. This is evident from the fact that in most cases, only one query is required to generate AEs.

RQ4. Our model can effectively deceive VirusTotal(VT) engines with an average of 73.97%. It is worth noting that the findings in this experiment validate the results observed in previous studies, such as (Ceschin et al., 2019).



RQ5

- Considered transferable adversarial examples (AE).
- We evaluated the evasion rates of AEs generated on a model (e.g., Sec-SVM), which works as a surrogate model, in misleading other target models (e.g., DREBIN). This is a stricter threat model that indicates the performance of our model in cases where adversaries are not capable of querying the target detectors.
- Thus, as soon as our model considers a stronger surrogate model (e.g., Sec-SVM), the AEs exhibit higher transferability.

RQ6

- We performed an empirical analysis to evaluate the performance of our model when utilizing an alternative search strategy for manipulation.
- We introduced a baseline manipulation method based on genetic algorithms(GA) for use relative to our model, where the fitness function of the baseline is the same as the (Random Search)RS-based method.
- Using RS outperforms GA. Specifically, RS not only leads to a 36.5% enhancement in (Evasion Rate)ER but also accelerates our model by $\approx 3\times$. These improvements are achieved with only 3 queries compared to the GA's 24 queries.

Limitations

- The adversarial payload size (the relative increase in the size of adversarial examples) that might be relatively high, especially for the small Android malware apps. This shortcoming may cause malware detectors to be suspicious of the AEs, particularly for popular Android applications. Improving the organ harvesting used in the program slicing technique, in particular, finding the smallest vein for a specific organ, can address this limitation, as each organ usually features multiple veins of different sizes.
- Our model crafts malware apps to mislead the malware detectors that use *static* features for classification. We do not anticipate our proposed evasion attack to successfully deceive ML-based malware detectors that work with behavioral features specified by dynamic analysis, as the perturbations are injected into malicious apps within an IF statement that is always False. Therefore, it remains an interesting avenue for future work to evaluate how our proposed attack can bypass behavior-based malware detectors.

Limitations

- Considering that our model uses a specific optimization problem, it can be extended to other platforms, such as Windows, if attackers offer problem-space transformations that are tailored to manipulate real-world objects like Windows Portable Executable files. Note that the transformations used in our model can only be applied to manipulate Android applications. We leave further exploration as future work since it is beyond the scope of this study.
- Our study comprehensively covers various malware detection systems, considering diverse classifiers on different features with various types. However, there is an opportunity to improve the validity of our findings since the evaluation is conducted in controlled laboratory settings. Future research should delve deeper into the applicability of our adversarial attacks framework to real-world environments, where dynamic factors like evolving malware landscapes and deployment scenarios may impact the attack's performance.

Conclusion

- This presentation reports a novel Android evasion attack in the problem space, designed to generate real-world adversarial Android malware capable of evading ML-based Android malware detectors in a blackbox setting.
- Unlike previous approaches, this directly operates in the problem space without initially focusing on finding feature-space perturbations. Experimental results demonstrate the effectiveness of this approach in deceiving various academic and commercial malware detectors.

Contact

- Professor Dr. Răzvan Bocu
- Email address: razvan@bocu.ro

Thank you!

Questions and discussion....