Università degli Studi di Salerno

Dipartimento di Informatica

Tesi di Laurea di I livello in

Informatica

Adversarial Attacks on Vision-based Deep Neural Networks in Autonomous Driving Vehicles

Relatore
Giuseppe Scanniello
Correlatore
Dott. Nome Cognome

Candidato
Michelangelo Esposito

Academic Year 2021-2022

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

List of Figures

List of Acronyms and Abbreviations

Chapter 1

Introduction

Things to add:

- where different types of coverage are useful
- Add formulas for coverage criteria

Chapter 2

Problem formulation

2.1 Autonomous Driving Vehicles

The **Automatic Land Vehicle in Neural Networl**, ALVINN, was the first self-driving car ever proposed, in 1989; it was based on neural networks responsible to detect lines, segment the environment, and drive the car. While the general principles on which it was based worked well, the limited computed capabilities at the time, didn't make the solution take off. Furthermore, the absence of data meant that it was extremely difficult to gather the necessary samples and datasets on which to base the models that would manage vision and driving.

LiDAR stands for Light Detection And Ranging. It's a method to measure distance of object by firing a focused laser beam and measuring the time it takes for it to bounce back at the source after being reflected by something. Compared to traditional camera images, LiDAR data can provide additional information about the surrounding scene...

A self-driving vehicle cannot rely on LiDAR alone, however. While this technology works great in most environments, including dark areas, if the scene surrounding the car becomes more noisy, due to rain or fog for example, the LiDAR sensor can become imprecise or fail.

For this reason, a third sensor is usually employed, the RADAR, which scans the surrounding area based on radio waves. . .

2.2 3D Object detection

The most suited category of neural networks for image processing are Convolutional Neural Networks, CNNs. . . CNNs can capture different patterns

Region Based Convolutional Neural Networks, R-CNN, is a more scalable alternative to traditional CNNS. This approach, originally proposed by R. Girshick et al. [1] in 2014. . .

CNNs have been successfully employed on point cloud data as well; PointRCNN

2.3 Adversarial attacks on neural networks

Most vision-based recognition software on ADS is based on CNNS; often, CNN-based deep learning models are vulnerable to the so called adversarial,

There can be small, pixel-level changes to an image that will cause the AI model to incorrectly interpret it or, on the other hand, a completely new image can be used to trick to model into thinking it is something else. The first kind is particularly dangerous, since such changes can be invisible to the human eye, and thus harder to detect.

There are mainly two categories of methods to achieve adversarial attacks, namely, optimization-based methods and fast gradient step method (FGSM)-based approach.

study on street signs: K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust Physical-World Attacks on Deep Learning Visual Classification," in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. IEEE, 2018, pp. 1625–1634.

In general, adversarial attacks are organized in three categories: evasion, poisoning, and extraction attacks:

- Evasion attacks modify the input to a classifier such that it is misclassified, while keeping the modification as small as possible. Evasion attacks can be black-box or white-box: in the white-box case, the attacker has full access to the architecture and parameters of the classifier. For a black-box attack, clearly this is not the case.

- In poisoning attacks, attackers have the opportunity of manipulating the training data to significantly decrease the overall performance, cause targeted misclassification or bad behavior, and insert backdoors and neural trojans

- Extraction attacks aim to develop a new model, starting from a proprietary black-box model, that emulate the behavior of the original model.

Chapter 3

Literature

In this chapter we provide a general overview of the literature concerning evolutionary test case generation, in both the Object-Oriented and procedural contexts. The main techniques and algorithms employed for the problem will be examined and compared, to act as the base for our study.

3.1 EvoSuite

EvoSuite is an example of an evolutionary algorithm that optimizes the whole test suite towards just one coverage criterion, rather than generating test cases directed towards multiple coverage criteria. With EvoSuite, any collateral coverage isn't a concern since all coverage is intentional, given that the ultimate goal is to generate the whole test suite. The algorithm starts with a randomly generated population of test suites. The fitness function rewards better coverage of the source code; if two suites have the same coverage, the one with fewer statements is chosen. For each test suite, its fitness is measured by executing all of its test cases and keeping track of the executed methods and of the minimal branch distance for each branch.

- Expand on bloat in EvoSuite
- algorithm pseudocode

3.2 NSGA-II

A popular algorithm for many-objective search problems is the Non-dominated Sorting Genetic Algorithm II (NSGA-II). This algorithm is based on three principles:

- It uses elitism when evolving the population: the most fit individuals are carried over along the offsprings.
- It uses an explicit diversity-preserving mechanism, the Crowding distance.
- It emphasizes the non-dominated solutions, as its name suggests.

In the context of test cases, domination can be expressed by the following relation:

Definition 1. A test case x dominates another test case y, also written $x \prec y$ if and only if ...

Algorithm 1: NSGA-II

```
input  : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
         Population size $M$
output : A test suite $T$

1  begin
2      $t \leftarrow 0$
3      $P_t \leftarrow$ RANDOM-POPULATION($M$)
4      while $not(search\_budget\_consumed)$ do
5          $Q_t \leftarrow$ GENERATE-OFFSPRING($P_t$)
6          $R_t \leftarrow P_t \cup Q_t$
7          $F \leftarrow$ FAST-NONDOMINATED-SORT($R_t$)
8          $P_{t+1} \leftarrow 0$
9          $d \leftarrow 1$
10         while $(|P_{t+1}| + |F_d| \leqslant M)$ do
11             CROWDING-DISTANCE-ASSIGNMENT($F_d$)
                $P_{t+1} \leftarrow P_{t+1} \cup F_d$
12             $d \leftarrow d + 1$
13         Sort($F_d$) // according to the crowding distance
14         $P_{t+1} \leftarrow P_{t+1} \cup F_d[1 : (M - |P_{t+1}|)]$
15         $t \leftarrow t + 1$
16     $S \leftarrow t + 1$
```

The NSGA-II algorithm works as follows:

- Starting from an initial population of individuals Pt, generate an offspring population Qt of equal size and merge the two together, obtaining the population Rt.
- Perform non-dominated sorting of the individuals in Rt based on target indicators and classify them by fronts, i.e. they are sorted according to an ascending level of non-domination. This ensures that the top Pareto-optimal individuals will survive to the next generation.
- If one of the fronts in the sorted sequence doesn't fit in terms of population size, crowding distance sorting is performed.
- Create the new population based on crowded tournament selection, then perform crossover and mutation.

Figure 3.1 summarizes the main loop of the algorithm:

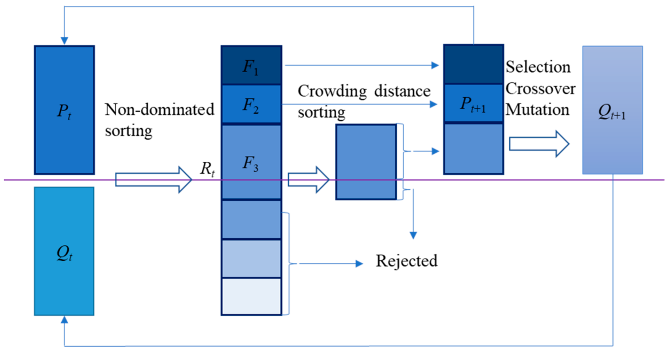

Figure 3.1: NSGA-II algorithm main loop.

In the context of software engineering, NSGA-II has been applied to problems such as software refactoring and test case prioritization, with two or three objectives. If the number of objectives begins to grow, however, the performance of the algorithm doesn't scale up well [**DBLP:journals/csur/LiLTY15**]. To overcome these limitations, there have been various adjustments and optimization of this algorithm...

3.3 SPEA2

Strength Pareto Evolutionary Algorithm (SPEA) is another multi-objective evolutionary search algorithm based on the concept of Pareto domination for fitness and selection, originally proposed by Zitzler and Thiele in 1999 [**DBLP:journals/tec/ZitzlerT99**]. SPEA2 was presented as an evolution of the algorithm by the same authors [**DBLP:journals/tec/ZitzlerT01**]. A peculiarity of SPEA2 is that it uses an archive to store non-dominated solutions generated in each iteration; as we will see later, the archive approach is also employed by more modern solutions.

During the evolution, a strength value is calculated for each individual in each generation and is used for the selection of the fittest individuals. For any solution i, this strength value is measured based on the numbers of j individuals, belonging to the archive and the population, dominated by i.

Algorithm 2: SPEA2

```
input  : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
         An archive A
         Population size $M$
1 begin
2     $P_t \leftarrow$ RANDOM-POPULATION($M$)
3     while $not(search\_budget\_consumed)$ do
4         CALCULATE-STRENGTH($P_t$, $A$)
5         UPDATE-ARCHIVE($P_t$, $A$)
6         BINARY-TOURNAMENT-SELECTION($P_t$)
7         CROSOVER($P_t$)
8         MUTATION ($P_t$)
```

If the size of the archive doesn't reach the population size, dominated individuals are used to fill up the remaining spaces. Similarly to NSGA-II isn't suitable for problems with more than three objectives[**DBLP:conf/icccsec/LiuZ19**].

3.4 MOSA

The Multi-Objective Sorting Algorithm, MOSA [**DBLP:conf/icst/PanichellaKT15**] is a GA proposed as an improvement over NSGA-II and SPEA2 for many-objective test case generation, with the main goal of allowing for a higher number of objectives to optimize. With this algorithm, the coverage of the different branches makes up the set of objectives to be optimized. The main characteristic about MOSA is that instead of ranking candidates for selection based on their Pareto optimality, it uses a preference criterion; this criterion selects the test case with the lowest objective score for each uncovered target; these selected individuals are given a higher chance of survival, while other test cases are ranked with the traditional NSGA-II approach.

As the first step, MOSA starts from a randomly generated initial population of test cases. To create the next generation, MOSA generates offsprings by implementing the classic operations of selection, crossover and mutation. Before selection occurs however, for each uncovered branch, the test case with the lowest objective score(branch distance + approach level) is determined; these test cases will have assigned the rank 0 and will make up the first front. The remainder of the test cases will be sorted according to the traditional NSGA-II approach.

After the rank assignment step is complete, the crowding distance is calculated to determine which individuals to select: the individuals with higher distance from the rest are given a higher chance of being selected. The algorithm attempts to select as many test cases as possible, starting from front 0, until the population size is reached.

Algorithm 3: MOSA

```
input  : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
         Population size $M$
output : A test suite $T$

1  begin
2      $t \leftarrow 0$
3      $P_t \leftarrow$ RANDOM-POPULATION($M$)
4      $archive \leftarrow$ UPDATE-ARCHIVE($P_t$, Ø)
5      while $not(search\_budget\_consumed)$ do
6          $Q_t \leftarrow$ GENERATE-OFFSPRING($P_t$)
7          $archive \leftarrow$ UPDATE-ARCHIVE($Q_t$, $archive$)
8          $R_t \leftarrow P_t \cup Q_t$
9          $F \leftarrow$ PREFERENCE-SORTING($R_t$)
10         $P_{t+1} \leftarrow 0$
11         $d \leftarrow 0$
12         while $(|P_{t+1}| + |F_d| \leqslant M)$ do
13             CROWDING-DISTANCE-ASSIGNMENT($F_d$)
                $P_{t+1} \leftarrow P_{t+1} \cup F_d$
14             $d \leftarrow d + 1$
15         Sort($F_d$) // according to the crowding distance
16         $P_{t+1} \leftarrow P_{t+1} \cup F_d[1 : (M - |P_{t+1}|)]$
17         $t \leftarrow t + 1$
18     $T \leftarrow archive$
```

Algorithm 4: PREFERENCE-SORTING

```
input  : $T = \{t_1, ..., t_n\}$, a set of candidate test cases
         Population size $M$
output : A non-dominated ranking assignment $F$

1  begin
2      $F_0$ // first front
3      for $u_i \in U$ and $u_i$ is uncovered do
4          // for each uncovered test target, the best test case according
             to the preference criterion is selected
5          $t_{best} \leftarrow$ test case in T with minimum objective score for $u_i$
6          $F_0 \leftarrow F_0 \cup \{t_{best}\}$
7      $T \leftarrow T - F_0$
8      if $|F_0| > M$ then
9          $F_1 \leftarrow T$
10     else
11         $U^* \leftarrow \{u \in U : u$ is uncovered$\}$
12         $E \leftarrow$ FAST-NONDOMINATED-SORT($T$ , $\{u \in U^*\}$)
13         $d \leftarrow 0$
14         for All non-dominated fronts in $E$ do
15             $F_{d+1} \leftarrow E_d$
```

Algorithm 5: DOMINANCE-COMPARATOR

```
input  : Two test cases to compare, $t1$ and $t2$
         $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
begin
    $dominates1 \leftarrow false$
    $dominates2 \leftarrow false$
    for $u_i \in U$ and $u_i$ is uncovered do
        $f_i^1 \leftarrow$ values of $u_i$ for $t_1$
        $f_i^2 \leftarrow$ values of $u_i$ for $t_2$
        if $f_i^1 < f_i^2$ then
            $dominates1 \leftarrow true$
        if $f_i^2 < f_i^1$ then
            $dominates2 \leftarrow true$
        if $dominates1 == true$ and $dominates2 == true$ then
            break
    if $dominates1 == dominates2$ then
        // $t1$ and $t2$ don't dominate each other
    else
        if $dominates1 == true$ then
            // $t1$ dominates $t2$
        else
            // $t2$ dominates $t1$
```

Finally, MOSA uses an archived population that keeps track of the best performing test cases, in order to form the final test suite. The archive is continuously updated as follows:

Algorithm 6: UPDATE-ARCHIVE

```
input  : A set of candidate test cases $T$
         An archive $A$
output : An updated archive $A$

begin
    for $u_i \in U$ do
        $t_{best} \leftarrow \emptyset$
        $best_length \leftarrow \infty$
        if $u_i$ already covered then
            $t_{best} \leftarrow$ test case in $A$ covering $u_i$
            $best_length \leftarrow$ number of statements in $t_best$
        for $t_j \in T$ do
            $score \leftarrow$ objective score of $t_j$ for target $u_i$
            $length \leftarrow$ number of statements in $t_j$
            if $score == 0$ and $length \leq best_length$ then
                replace $t_best$ with $t_j$ in $A$
                $t_best \leftarrow t_j$
                $best_length \leftarrow length$
    return $A$
```

3.5 DynaMOSA

One of the limitations of MOSA is its inability to consider the implicit dependencies between targets, which simply appear as independent objectives to optimize. Such dependencies can be quite common in practice: most commonly, there exist branches which may only be satisfied if and only if other branches in the outer scope have been already covered.

DynaMOSA, Dynamic Many-Objective Sorting Algorithm [**DBLP:journals/tse/PanichellaKT18**] is an algorithm that focuses on these dynamic dependencies, and has been proposed as an evolution of MOSA. Before introducing the algorithm, a few definitions are needed [**DBLP:journals/tse/PanichellaKT18**]:

Definition 2. (Dominator): A statement $s1$ dominates another statement $s2$ if every execution path to $s2$ passes through $s1$.

Definition 3. (Post-dominator): A statement $s1$ post-dominates another statement $s2$ if every execution path from $s2$ to the exit point passes through $s1$.

Definition 4. (Control dependency): There is a control dependency between program statement $s1$ and program statement $s2$ iff: (1)$s2$ is not a postdominator of $s1$, and (2) there exist a path in the control flow graph between $s1$ and $s2$ whose nodes are postdominated by $s2$.

Definition 5. (Control dependency graph): The graph $G = \langle N, E, s\rangle$, consisting of nodes $n \in N$ that represent program statements, and edges $e \in E \subseteq NXN$ that represent control dependencies between program statements, is called control dependency graph. Node $s \in N$ represents the entry node, which is connected to all nodes that are not under the control dependency of another node.

This definition can be extended to other coverage criteria. For example, given two branches $b1$ and $b2$, we say that $b1$ holds a control dependency on $b2$ if $b1$ is postdominated by a statement $s1$ which holds a control dependency on a node $s2$ that postdominates $b2$.

DynaMOSA uses the control dependency graph to identify which targets are independent from each other and which ones can be covered only after satisfying previous targets in the graph. Algorithm 6 highlights the test case evolution in DynaMOSA.

Algorithm 7: DynaMOSA

```
input  : $U = \{u_1, ..., u_m\}$ the set of coverage targets of a program
         Population size $M$
         $G = \langle N, E, s \rangle$: control dependency graph of the program
         $\phi : E \rightarrow U$ : partial mapping between edges and targets
output : A test suite $T$

1  begin
2      $U^* \leftarrow$ targets in $U$ with not control dependencies
3      $t \leftarrow 0$
4      $P_t \leftarrow$ RANDOM-POPULATION($M$)
5      $archive \leftarrow$ UPDATE-ARCHIVE($P_t$, Ø)
6      $U^* \leftarrow$ UPDATE-TARGETS($U^*$, $G$, $\phi$)
7      while $not(search\_budget\_consumed)$ do
8          $Q_t \leftarrow$ GENERATE-OFFSPRING($P_t$)
9          $archive \leftarrow$ UPDATE-ARCHIVE($Q_t$, $archive$)
10         $U^* \leftarrow$ UPDATE-TARGETS($U^*$, $G$, $\phi$)
11         $R_t \leftarrow P_t \cup Q_t$
12         $F \leftarrow$ PREFERENCE-SORTING($R_t$, $U^*$)
13         $P_{t+1} \leftarrow 0$
14         $d \leftarrow 0$
15         while $(|P_{t+1}| + |F_d| \leqslant M)$ do
16             CROWDING-DISTANCE-ASSIGNMENT($F_d$, $U^*$)
                $P_{t+1} \leftarrow P_{t+1} \cup F_d$
17             $d \leftarrow d + 1$
18         Sort($F_d$) // according to the crowding distance
19         $P_{t+1} \leftarrow P_{t+1} \cup F_d[1 : (M - |P_{t+1}|)]$
20         $t \leftarrow t + 1$
21     $T \leftarrow archive$
```

The main difference with MOSA is how the initial target population is selected; with DynaMOSA, only the targets that are free from dependencies are part of this set. Then, in each iteration, the set of non-dependent targets is updated by using the following procedure:

Algorithm 8: UPDATE-TARGETS

```
input  : $G = \langle N, E, s \rangle$: control dependency graph of the program
         $U^* \subseteq U$: current set of targets
         $\phi : E \rightarrow U$ : partial mapping between edges and targets
output : $U^*$: updated set of targets to optimize
1  begin
2      for $u \in U$ do
3          if $u$ is covered then
4              $U^* \leftarrow U^* - \{u\}$
5              $e_u \leftarrow$ edge in $G$ for the target $u$
6              visit($e_u$)
7  function visit($e_u \in E$):
8      for each unvisited $e_n \in E$ control depenmdent on $e_u$ do
9          if $\phi(e_n)$ is not covered then
10             $U^* \leftarrow U^* \cup \{\phi(e_n)\}$
11             set $e_n$ as visited
12         else
13             visit($e_m$)
```

This routine update the sets of selected targets U^* in order to include any uncovered targets that are control dependent on the newly covered target. In the case of newly covered targets, the procedure iterates over the control graph to find all control dependent targets.

3.6 OCELOT and object-oriented versus procedural test case generation

Generally speaking, both EvoSuite and MOSA/DynaMOSA have been designed with the OO paradigm in mind, as well as most empirical software engineering tools for testing.

...expand on automated testing for procedural programs...

Optimal Coverage sEarch-based tooL for sOftware Testing, OCELOT [**DBLP:conf/ssbse/ScalabrinoGNOL16**] is a test case generation tool for C programs that implements both a multi-objective approach based on MOSA, and a new iterative single-target approach named LIPS, Linear Independent Path-based Search. OCELOT can automatically detect the input types of a C function, without requiring any specification of parameters; Additionally it can handle the different data types of C, including pointers and structs, and produces test cases based on the Check framework [2]. Similarly to other tools, OCELOT is not able to generate oracles. From an implementation point of view, the GA implementation in OCELOT was realized with the JMetal, a Java framework for multi-objective optimization with meta-heuristics [3]. Furthermore, JNI is used as interface with the target program to try different combinations of test data. SBX Crossover, Polynomial Mutation and Binary Tournament Selection, for both the multi-objective and iterative approaches.

The iterative approach used in OCELOT, LIPS

Running OCELOT consists of two main phases: build and run [**DBLP:conf/kbse/ScalabrinoGNC**
as highlighted in figure 3.2:

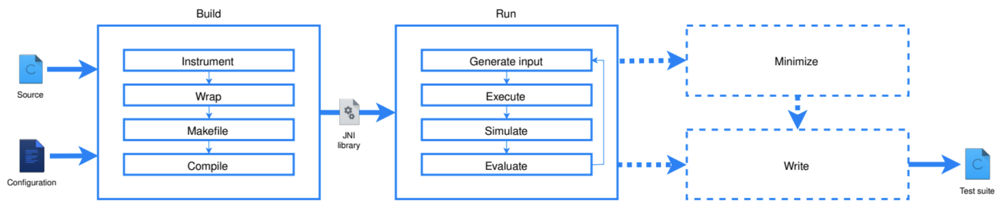

Figure 3.2: OCELOT components and workflow.

3.7 Test case generation for CyberPhysical Systems

Typically, in the development stages of a CPS, validation and verification happen according to the V-model approach.

AmbieGen based on NSGA-II. Markov chains used to assign values to different attributes

[4] [5]

Chapter 4

Conclusions

Bibliography

[1] Ross B. Girshick et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*. IEEE Computer Society, 2014, pp. 580–587. DOI: `10.1109/CVPR.2014.81`. URL: `https://doi.org/10.1109/CVPR.2014.81`.

[2] *Check: framework for unit testing in C*. URL: `https://libcheck.github.io/check/`.

[3] *JMetal Java framework*. URL: `http://jmetal.sourceforge.net/`.

[4] Dmytro Humeniuk, Foutse Khomh, and Giuliano Antoniol. "A search-based framework for automatic generation of testing environments for cyber-physical systems". In: *Inf. Softw. Technol.* 149 (2022), p. 106936. DOI: `10.1016/j.infsof.2022.106936`. URL: `https://doi.org/10.1016/j.infsof.2022.106936`.

[5] Walid M. Taha, Abd-Elhamid M. Taha, and Johan Thunberg. *Cyber-Physical Systems: A Model-Based Approach*. Springer US, 2021.

[6] Jindi Zhang et al. "Evaluating Adversarial Attacks on Driving Safety in Vision-Based Autonomous Vehicles". In: *IEEE Internet Things J.* 9.5 (2022), pp. 3443–3456. DOI: `10.1109/JIOT.2021.3099164`. URL: `https://doi.org/10.1109/JIOT.2021.3099164`.

[7] Vincenzo Riccio et al. "Testing machine learning based systems: a systematic mapping". In: vol. 25. 6. 2020, pp. 5193–5254. DOI: `10.1007/s10664-020-09881-0`. URL: `https://doi.org/10.1007/s10664-020-09881-0`.

[8] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[9] Edmund K. Burke and Graham Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer US, 2014, pp. 403–449.

[10] Stefan Edelkamp and Stefan Schrodl. *Heuristic Search: Theory and Applications*. Morgan Kaufmann, 2008, pp. 403–449.

[11] A. Author and A. Author. *Book reference example*. Publisher, 2099.

[12] A. Author. "Article title". In: *Journal name* (2099).

[13] *Example*. URL: `https://www.isislab.it`.

[14] A. Author. "Tesi di esempio ISISLab". 2099.