

# Hardware Performance Counters Can Detect Malware: Myth or Fact?

**Boyou Zhou**, Anmol Gupta, Rasoul Jahanshahi,  
Manuel Egele, Ajay Joshi

Email: {bobzhou, anmol.gupta1005, rasoulj, megele, joshi}@bu.edu

June 10, 2018



# Malware Explosion

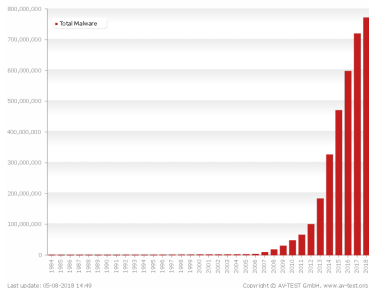


Figure: Exponential Growth in Total Number of Malware[av-test.org 2017]

# Malware Explosion

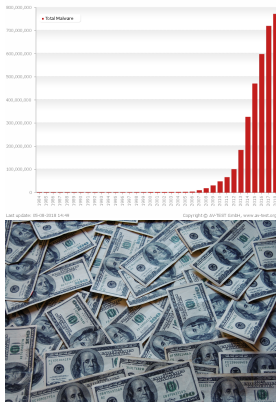


Figure: the Damage of Malware [av-test.org 2017] [verdict.co.uk 2017]  
[StrongArm.io][thehackernews.com 2018]

# Overview

- 1 Motivation
- 2 Prior Works
- 3 Contribution
- 4 Experimental Setup
- 5 Data Analysis
  - Event Selection
  - Data Division
  - Our Findings
- 6 Malware Example
- 7 Summary

# Distinguishing Malware and Benignware

- Malware detection method:
  - Signature-based analysis
  - Dynamic analysis

# Distinguishing Malware and Benignware

- Malware detection method:
  - Signature-based analysis
  - Dynamic analysis
- To decrease the anti-virus performance overhead, previous works propose to use Hardware Performance Counters (HPCs) to detect malware.

# Distinguishing Malware and Benignware

- Malware detection method:
  - Signature-based analysis
  - Dynamic analysis
- To decrease the anti-virus performance overhead, previous works propose to use Hardware Performance Counters (HPCs) to detect malware.
- HPCs have negligible performance overhead during information extraction.

# Distinguishing Malware and Benignware

- Malware detection method:
  - Signature-based analysis
  - Dynamic analysis
- To decrease the anti-virus performance overhead, previous works propose to use Hardware Performance Counters (HPCs) to detect malware.
- HPCs have negligible performance overhead during information extraction.
- Can the information of HPC values be used for malware detection?

# Hardware Performance Counters (HPCs)

- *Hardware Performance Counters (HPCs)* are the hardware units that count *micro-architectural events*:
  - cache misses/hits.
  - floating-point (fp) operations

# Hardware Performance Counters (HPCs)

- *Hardware Performance Counters (HPCs)* are the hardware units that count *micro-architectural events*:
  - cache misses/hits.
  - floating-point (fp) operations

Example 1:

```
def count_to_100():  
    count = 0;  
    while (count ≤ 100):  
        count = count + 1;  
        encrypt_file(file1, key);
```

Example 2:

```
def count_to_100():  
    count = 0.2;  
    while (count ≤ 100.2):  
        count = count + 1.0;  
        encrypt_file(random(), key);
```

# Hardware Performance Counters (HPCs)

- *Hardware Performance Counters (HPCs)* are the hardware units that count *micro-architectural events*:
  - cache misses/hits.
  - floating-point (fp) operations

Example 1:

```
def count_to_100():  
    count = 0;  
    while (count ≤ 100):  
        count = count + 1;  
        encrypt_file(file1, key);
```

Example 2:

```
def count_to_100():  
    count = 0.2;  
    while (count ≤ 100.2):  
        count = count + 1.0;  
        encrypt_file(random(), key);
```

- More cache hits in Example 1 - encryption on the same file
- More fp-operations in Example 2 - no fp-operations in the Example 1

# Hardware Performance Counters (HPCs)

- There are more than 130 micro-architectural events on Intel, but only 4 can be monitored at a time.
- AMD has 6 counters that can be monitored at a time.
- Previous works **have not** used time-multiplexing to monitor more events.

# Program Semantics

Example 3:

```
def save_to_keyvault():
```

# Program Semantics

Example 3:

```
def save_to_keyvault():  
    key=generate_key(seed);
```

# Program Semantics

Example 3:

```
def save_to_keyvault():  
    key=generate_key(seed);  
    encrypt_file(file, key);
```

# Program Semantics

Example 3:

```
def save_to_keyvault():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_cloud(ip1, key);
```

# Program Semantics

Example 3:

```
def save_to_keyvault():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_cloud(ip1, key);  
    print("Encryption Completed.");
```

# Program Semantics

Example 3:

```
def save_to_keyvault():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_cloud(ip1, key);  
    print("Encryption Completed.");
```

Example 4:

```
def ransomware():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_attacker(ip2, key);  
    print("Where is my money?");
```

# Program Semantics

Example 3:

```
def save_to_keyvault():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_cloud(ip1, key);  
    print("Encryption Completed.");
```

- HPC values do not distinguish between `ip1` and `ip2`.

Example 4:

```
def ransomware():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_attacker(ip2, key);  
    print("Where is my money?");
```

# Program Semantics

Example 3:

```
def save_to_keyvault():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_cloud(ip1, key);  
    print("Encryption Completed.");
```

- HPC values do not distinguish between **ip1** and **ip2**.
- The difference between ransomware and crypto-programs is **who** holds the key (user in Example 3 and attacker in Example 4).

Example 4:

```
def ransomware():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_attacker(ip2, key);  
    print("Where is my money?");
```

# Program Semantics

Example 3:

```
def save_to_keyvault():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_cloud(ip1, key);  
    print("Encryption Completed.");
```

- HPC values do not distinguish between **ip1** and **ip2**.
- The difference between ransomware and crypto-programs is **who** holds the key (user in Example 3 and attacker in Example 4).

Example 4:

```
def ransomware():  
    key=generate_key(seed);  
    encrypt_file(file, key);  
    upload_key_to_attacker(ip2, key);  
    print("Where is my money?");
```

It is counter-intuitive that high-level program behaviors would manifest themselves in low-level hardware behaviors.

# Overview

- 1 Motivation
- 2 Prior Works
- 3 Contribution
- 4 Experimental Setup
- 5 Data Analysis
  - Event Selection
  - Data Division
  - Our Findings
- 6 Malware Example
- 7 Summary

# Previous HPC malware detection system

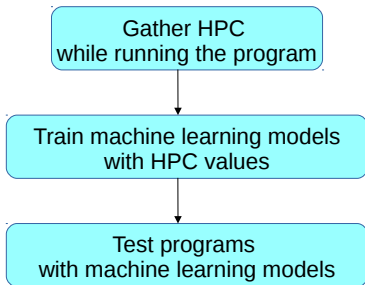


Figure: General Workflow

# Previous HPC malware detection system

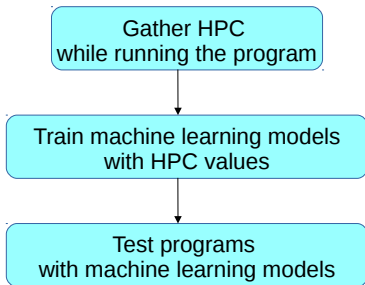


Figure: General Workflow

These listed works apply a general workflow to use HPCs to detect malware: [Demme 2013 ISCA] [Tang 2014 RAID] [Ozsoy 2015 HPCA] [Khasawneh 2015 RAID] [Wang 2016 TACO] [Kazdagli 2016 MICRO] [Singh 2017 AsiaCCS] [Khasawneh 2017 MICRO]

# Experimental & Analytical Drawbacks

Why do those works draw the conclusion that HPC can be used in malware detection?

# Experimental & Analytical Drawbacks

Why do those works draw the conclusion that HPC can be used in malware detection?

## Unrealistic Experimental Setup

- Virtual Machines [**Vincent 2013 ISPASS**]
- Few data samples
- Dynamic binary instrumentation

# Experimental & Analytical Drawbacks

Why do those works draw the conclusion that HPC can be used in malware detection?

## Unrealistic Experimental Setup

- Virtual Machines [**Vincent 2013 ISPASS**]
- Few data samples
- Dynamic binary instrumentation

## Biased Data Analysis

- Unrealistic data division
- No quantitative selection of events
- No cross-validations, insufficient validations

# Overview

- 1 Motivation
- 2 Prior Works
- 3 Contribution**
- 4 Experimental Setup
- 5 Data Analysis
  - Event Selection
  - Data Division
  - Our Findings
- 6 Malware Example
- 7 Summary

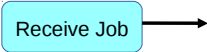
# Contributions

- We identify the unrealistic assumptions and the insufficient analysis used in prior works.
- We perform thorough experiments with a program count that exceeds prior works by a factor of  $2\times \sim 3\times$ .
- We compare the effects of the experimental settings (division of data) on the quality of machine learning.
- Finally, we make all code, data, and results of our project publicly available.

# Overview

- 1 Motivation
- 2 Prior Works
- 3 Contribution
- 4 Experimental Setup**
- 5 Data Analysis
  - Event Selection
  - Data Division
  - Our Findings
- 6 Malware Example
- 7 Summary

# Our Experiment Workflow



Receive Job

**Figure:** Our workflow of benignware/malware experiments

# Our Experiment Workflow

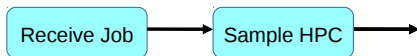


**Figure:** Our workflow of benignware/malware experiments

# Our Experiment Workflow

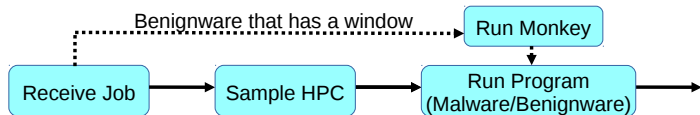


Figure: Our workflow of benignware/malware experiments

# Our Experiment Workflow

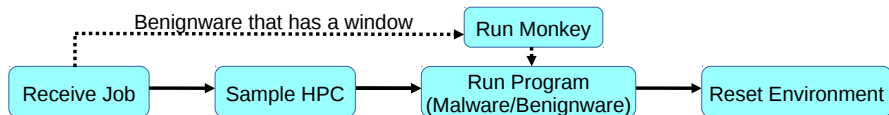


Figure: Our workflow of benignware/malware experiments

# Our Experiment Workflow

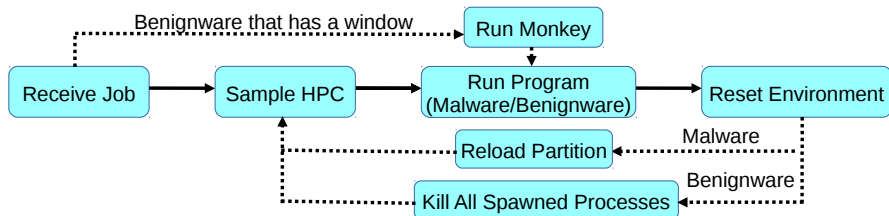


Figure: Our workflow of benignware/malware experiments

# Overview

- 1 Motivation
- 2 Prior Works
- 3 Contribution
- 4 Experimental Setup
- 5 Data Analysis**
  - Event Selection
  - Data Division
  - Our Findings
- 6 Malware Example
- 7 Summary

# Data Analysis - Event Selection

- Previous works selected events based on “expert intuition”.
- Without “expert intuition”, we found out that our quantitative selected events have many overlapping events with previous works.

# Data Analysis - Event Selection

- Previous works selected events based on “expert intuition”.
- Without “expert intuition”, we found out that our quantitative selected events have many overlapping events with previous works.

Table: Description of the Selected Events

| Events         | Definition                                                                    |
|----------------|-------------------------------------------------------------------------------|
| <b>0x04000</b> | <b>The number of accesses to the data cache for load and store references</b> |
| <b>0x03000</b> | <b>The number of CLFLUSH instructions executed</b>                            |
| 0x02B00        | The number of System Management Interrupts (SMIs) received                    |
| <b>0x02904</b> | <b>The number of Load operations dispatched to the Load-Store unit</b>        |
| <b>0x02902</b> | <b>The number of Store operations dispatched to the Load-Store unit</b>       |
| 0x02700        | The number of CPUID instructions retired                                      |

# Data Analysis - Data Division

## Training-testing Approach (TTA)

- TTA1: Testing on traces produced by the same program sample.

# Data Analysis - Data Division

## Training-testing Approach (TTA)

- TTA1: Testing on traces produced by the same program sample.  
→ not realistic

# Data Analysis - Data Division

## Training-testing Approach (TTA)

- TTA1: Testing on traces produced by the same program sample.  
→ not realistic
- TTA2: Testing on traces produced by the program from same category/family.

# Data Analysis - Data Division

## Training-testing Approach (TTA)

- TTA1: Testing on traces produced by the same program sample.  
→ not realistic
- TTA2: Testing on traces produced by the program from same category/family.  
→ realistic

# Data Analysis - Data Division

## Training-testing Approach (TTA)

- TTA1: Testing on traces produced by the same program sample.  
→ not realistic
- TTA2: Testing on traces produced by the program from same category/family.  
→ realistic

# Data Analysis - Data Division

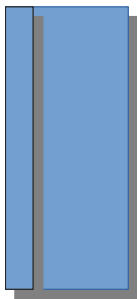
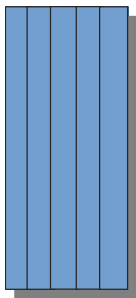


Figure: Data Division

# Data Analysis - Data Division



1

Figure: Data Division

# Data Analysis - Data Division

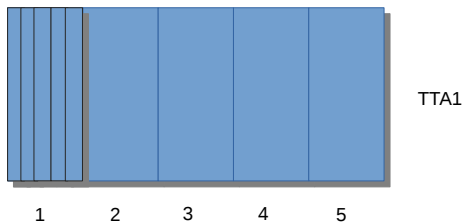


Figure: Data Division

# Data Analysis - Data Division

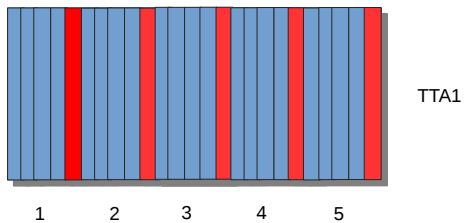


Figure: Data Division

# Data Analysis - Data Division

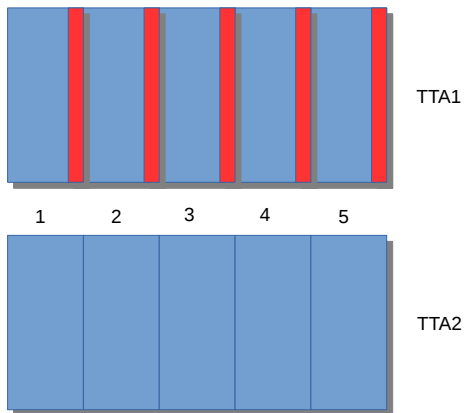


Figure: Data Division

# Data Analysis - Data Division

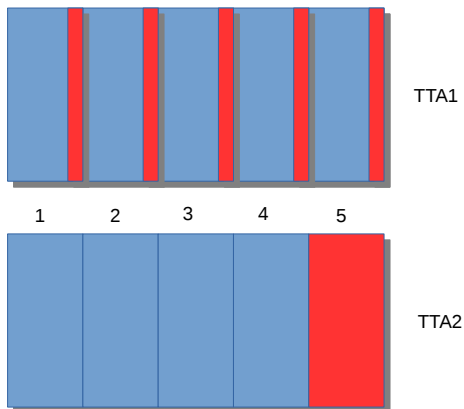


Figure: Data Division

# Data Analysis - Our Findings

## High False Positives (FP)

- Our detection False Discovery Rate (FDR) is 15%.  $FDR = \frac{FP}{FP+TP}$

# Data Analysis - Our Findings

## High False Positives (FP)

- Our detection False Discovery Rate (FDR) is 15%.  $FDR = \frac{FP}{FP+TP}$
- If we deploy this system to a Windows 7 file system, among 1,323 executables, 198 would be flagged as malware.

# Data Analysis - Our Findings

## High False Positives (FP)

- Our detection False Discovery Rate (FDR) is 15%.  $FDR = \frac{FP}{FP+TP}$
- If we deploy this system to a Windows 7 file system, among 1,323 executables, 198 would be flagged as malware.

## Large Standard Deviation (STD)

- We cross-validate our models in both data division types.

# Data Analysis - Our Findings

## High False Positives (FP)

- Our detection False Discovery Rate (FDR) is 15%.  $FDR = \frac{FP}{FP+TP}$
- If we deploy this system to a Windows 7 file system, among 1,323 executables, 198 would be flagged as malware.

## Large Standard Deviation (STD)

- We cross-validate our models in both data division types.
- TTA2 results in  $1.762\times$  larger STD than the results from TTA1.

# Experimental & Analytical Drawbacks

Why do those works draw the conclusion that HPC can be used in malware detection?

## Unrealistic Experimental Setup

- Virtual Machines
- Few data samples
- Dynamic binary instrumentation

## Biased Data Analysis

- Unrealistic data division
- No quantitative selection of events
- No cross-validations

# Experimental & Analytical Drawbacks

Why do those works draw the conclusion that HPC can be used in malware detection?

## Unrealistic Experimental Setup

- Virtual Machines
- Few data samples
- Dynamic binary instrumentation

## Biased Data Analysis

- Unrealistic data division
- No quantitative selection of events
- No cross-validations

# Overview

- 1 Motivation
- 2 Prior Works
- 3 Contribution
- 4 Experimental Setup
- 5 Data Analysis
  - Event Selection
  - Data Division
  - Our Findings
- 6 Malware Example**
- 7 Summary

# An Example - Ransomware

- During our experiments, we observed the variations in HPC values.

# An Example - Ransomware

- During our experiments, we observed the variations in HPC values.
- We write a simple malware that can hide from the HPC malware detection,

# An Example - Ransomware

- During our experiments, we observed the variations in HPC values.
- We write a simple malware that can hide from the HPC malware detection, by infusing a malware (ransomware) into benignware (Notepad++).
- We train traces from the original ransomware (with injected into Notepad++) and benignware in our detection system. The detection system fails to detect our malware.

# Overview

- 1 Motivation
- 2 Prior Works
- 3 Contribution
- 4 Experimental Setup
- 5 Data Analysis
  - Event Selection
  - Data Division
  - Our Findings
- 6 Malware Example
- 7 Summary

# Summary

- We identify the unrealistic assumptions and the insufficient analysis used in prior work.
- We provide guidelines for future works in malware detection:
  - Run experiments on bare-metal machines (no VM, DBI) with more program samples
  - Select events based on quantitative analysis
  - Divide training and testing dataset based on program samples (TTA2)
  - Perform cross-validations
- We open-source our work in the following link:  
[https://github.com/bu-icsg/Hardware\\_Performance\\_Counters\\_Can\\_Detect\\_Malware\\_Myth\\_or\\_Fact](https://github.com/bu-icsg/Hardware_Performance_Counters_Can_Detect_Malware_Myth_or_Fact)



# Backup

# Principal Component Analysis

In order to avoid *curse of dimensions*, we reduce the feature dimension by applying Principal Component Analysis.

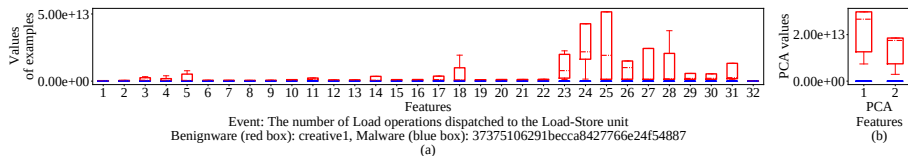


**Figure:** Distributions of sampled values before (a) & after (b) the reduction of dimensions.

# Reduction of Approximation Error

We only use the main components during PCA, which introduces approximation error. Thus, we minimize the approximation error by selecting the events with minimum approximation error.

$$A = V\lambda V^{-1} \approx V'\lambda V'^{-1} \quad (1)$$

$$AV = \sum_{i=1}^m v^{(i)}\lambda^{(i)} + \sum_{i=m+1}^n v^{(i)}\lambda^{(i)} \quad (2)$$

$$= \sum_{i=1}^m v^{(i)}\lambda^{(i)} + \epsilon(\alpha v\lambda) \quad (3)$$

# Reduction of Approximation Error

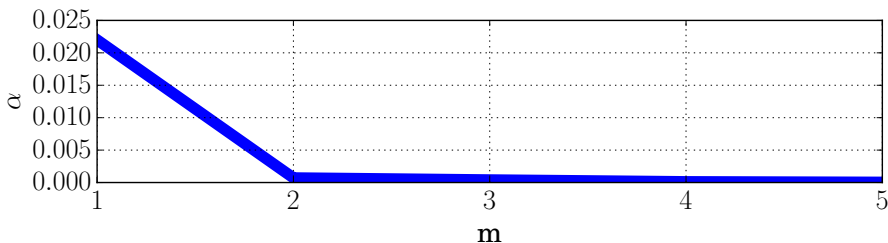
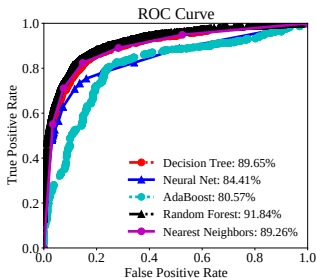
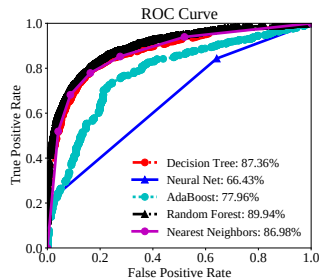


**Figure:** Error Bound vs the Number of Eigenvectors Plot: when choosing different number of eigenvectors for reduction in dimensions, the error bound  $\alpha$  changes according to  $m$  eigenvectors.

# Roc curves



(a)



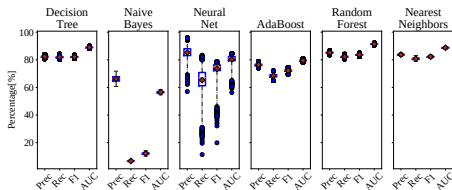
(b)

Figure: Receiver Operating Characteristic (ROC) curve of 5 models.

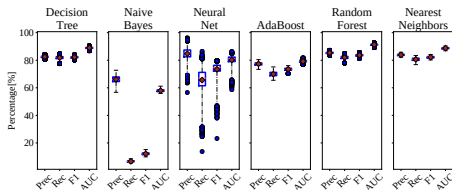
**Table:** Detection Rates with TTA1 and TTA2: **Red** means the value is less than 50% and **bold** means that the value is more than 90%

| Models            | TTA1         |             |              |              | TTA2         |              |              |        |
|-------------------|--------------|-------------|--------------|--------------|--------------|--------------|--------------|--------|
|                   | Precision[%] | Recall[%]   | F1-Score[%]  | AUC[%]       | Precision[%] | Recall[%]    | F1-Score[%]  | AUC[%] |
| Decision Tree     | 83.04        | 83.75       | 83.39        | 89.65        | 83.21        | 77.44        | 80.22        | 87.36  |
| Naive Bayes       | 70.36        | <b>7.97</b> | <b>14.32</b> | 58.11        | 56.72        | <b>5.425</b> | <b>9.903</b> | 58.38  |
| Neural Net        | 82.41        | 75.4        | 78.75        | 84.41        | <b>91.34</b> | <b>22.16</b> | <b>35.66</b> | 66.43  |
| AdaBoost          | 78.61        | 71.73       | 75.01        | 80.57        | 75.78        | 65.6         | 70.32        | 77.96  |
| Random Forest     | 86.4         | 83.34       | 84.84        | <b>91.84</b> | 84.36        | 78.44        | 81.29        | 89.94  |
| Nearest Neighbors | 84.84        | 82.37       | 83.59        | 89.26        | 82.7         | 77.88        | 80.22        | 86.98  |

# Distributions of Cross-validations



(a)



(b)

Figure: Box plots of distributions of 10-fold cross-validation experiments using (a) TTA1 and (b) TTA2.