# Tripwires: a rule-based detection system for advanced persistent threats


Candidate no. 1054467

Word count: 4932

Submitted in partial completion of

*Part B of the Final Honour Schools of Computer Science*

Trinity 2023

# Contents

# List of Figures



# List of Tables

# 1 Introduction

In an age that is increasingly digital and interconnected, fully secure computer systems still remain an aspiration[17], with the detection of attacks forming one major component of this issue. Cyber attacks are having increasingly large impacts - the attacks on the Colonial Pipelines across the Southern United States in 2021 cost Colonial $4.4 million in ransom payments, but also caused fuel prices in surrounding states to increase by as much as 21 cents per gallon[37]. Additionally, the World Economic Forum reported that 93% of cyber leaders and 86% of business leaders believe that it is likely that geopolitical instability will lead to a catastrophic cyber event in the next two years[10]. As a result, detecting cyber threats effectively and taking precautions to hinder such attacks from occurring is of paramount importance. Cyber threats can take many forms; this project will explore using *tripwires* to detect *advanced persistent threats*.

## 1.1 Advanced Persistent Threats

The term *advanced persistent threat* (APT) was coined in 2006 by United States Air Force analysts[9]. The term describes the nature of APT attacks perfectly:

- Advanced refers to specifically targeted attacks of a covert nature, often utilising multiple attack vectors and techniques in order to gain access to their target devices/networks. Groups behind APT attacks typically have access to significant funding, technological know-how and resources in order to launch targeted attacks

through a range of mediums.

- Persistent means that the attacks gain a medium to long term foothold within their target network, opting for a protracted attack over a long period of time instead of more rapid data extraction or damage. Since activity is spread over a longer period APTs tend to be significantly harder to detect than other intrusions; one specific attack analysed by Mandiant had an intrusion time of nearly 5 years[26].

- APTs represent significant threats to their targets, allowing the attacker to either exfiltrate confidential data or cause damage to physical infrastructure, most famously seen in the Stuxnet attack on the Iranian nuclear program[24].

APTs typically are multi-stage attacks, initially penetrating into system before gaining further privileges and information in order to reach its goal[8], often reporting back to a command and control centre where the malware can be modified, updated or tasked with new targets. They commonly follow the *Intrusion Kill Chain*[21] (IKC) which provides a common attack methodology. In spite of this, detecting APTs remains a significant issue, with CrowdStrike reporting an average dwell time (the duration for which an adversary has unfettered access to a compromised system) of 79 days in 2021[13]. One of the primary reasons behind this is that APTs attempt to hide 'under the radar' with low amounts of background traffic over a protracted period of time, making it challenging to detect anomalous behaviours within a large and complex system.

## 1.2 Attack Pattern Tripwires

*Attack pattern tripwires*(tripwires) are an *signature-based detection system* that use a use known behavioural patterns of different attacks to detect intrusion[2]. Attacks can be broken down into *attack trees*, which describe the processes taken by attacks through a number of individual action nodes (eg. accessing certain websites, creating/modifying system files or exploiting known vulnerabilities) connected by transitions. Known APTs are good candidates for modelling using attack trees and therefore could be detected using tripwires, however this has yet to be done.

## 1.3 Aim of the Project

The aim of this project is to apply the attack-pattern tripwire methodology to APTs and discern how well attack pattern based rules can detect APTs within systems. Current work exists using decision trees to detect APTs[30] and combining attack-trees and IKC to detect malicious insider attacks[14], however no work exists using attack-trees to detect external APT attacks.

In this project an analysis will be carried out on a number of different APTs with their attack methodologies studied in order to derive precise attack trees. These attack trees will then form the foundation of an *intrusion detection system* (IDS), being tested against publicly available datasets and some semi-synthetic datasets we generate to detect attacks when they occur. A novel IDS will be developed from scratch, building upon the further work outlined by Zhang et al.[40] by implementing multithreading.

## 1.4 Contributions

- We show the feasibility of using attack pattern tripwires to detect advanced persistent threats.
- We develop highly specific attack trees for Eternal Blue and Stuxnet that can be used for detection by a rule-based detection system.
- We develop a novel intrusion detection system to implement these tripwires for testing.
- We extend other tripwire-based systems with multithreading to improve performance.

# 2 Literature Review

There are currently two common approaches for detecting cyber-attacks; these are *rule-(or signature-)based detection* and *anomaly-based detection*. Anomaly-based detection typically makes use of machine learning in order to analyse large quantities of network and log data, allowing the system to then identify potentially suspicious or malicious behaviour. By contrast, rule-based (or signature-based) detection uses predefined behavioural patterns or signatures of attacks in order to detect intrusions. Both of these approaches can be applied to the detection of APTs, but they each have their distinct drawbacks.

## 2.1 Anomaly-Based Detection

Anomaly-based detection determines a baseline indicative of normal behaviour and then flags deviations from this norm. The majority of these systems are based on machine learning techniques due to the large quantities of data involved.

Anomaly-detection systems typically rely on system monitoring data in the form of log files. These log files record events as they occur, with these normally consisting of one system entity performing an action on another entity - a process accessing a network connection or creating a file for example. Since an APT requires multiple steps in order to achieve its goals, it will necessarily leave behind a series of footprints in system logs. Gao et al. try to exploit this behaviour in their paper presenting SAQL, a stream based query system for anomaly detection[18]. This allows users to define a series of thresholds to detect anomalous behaviour indicative of an APT attack using their query language, however there are performance limitations in the number of concurrent queries, limiting its effectiveness when attempting to detect a wider range of threats. It is also rather sensitive to minor changes in system behaviour, which restricts its utility within large enterprises due to the constant evolution in device and systems usage.

Han et al. [19] developed a system using provenance (information regarding to file accesses, writes, processes etc) graphs specifically to detect APTs. Their system UNICORN uses both machine learning and dynamic analysis, developing upon similar systems by allowing for more flexible transitions between states and also real-time analysis; this is quite rare given the quantity of provenance data produced by even

moderate use. However, it has limited scalability within larger environments, both due to greater overheads and the increased difficulty in effectively detecting malicious anomalies.

Similar anomaly-based detection methods are also possible on subsets of network data with Zhao et al.[41] developing IDnS, a system that used DNS activity to detected APT command and control (C2C, see below) domains. This first extracted a range of potentially malicious features from DNS requests, such as particularly short Time To Live values and addresses with a third-level domain (eg. a is the third-level address in a.b.com), before using machine learning to develop a reputation engine which could be used to detect APT infections with a 96% accuracy. This highlights that importance of using a wide range of data fields in detecting APTs: any APT that uses IP addresses instead of domains to communicate with its C2C servers is immediately undetectable by this system. The ideal solution to this is to have a diverse range of IDSs focussing on different data in order to minimise the risk of an anomaly being detected.

One issue that is shared across most anomaly-based detection systems (and particularly those using machine learning) is the trade-off between false positives and false negatives. A high sensitivity to anomalies will often lead to false positives due to natural variations in system usages, however this is preferable to a lower sensitivity and therefore higher false negative rate, with false negatives already being a major issue due to the covert attack methods of APTs. By contrast, well designed signature-based systems tend to have a lower false positive rate due to the more prescriptive definition of what constitutes a alert.

Some research has attempted to produce hybrid IDSs through combining signature-based detection with other methods. An example of this is Alshammari et al.[5] who attempted to improve on pure rule-based detection by using supervised machine learning methods. Their hybrid intrusion detection system achieved accuracy rates of over 96% when tested against datasets of different attacks on vehicles. However it is quite small in scope, focussing just on attacks on the Controller Area Network bus and with much more simplistic threats than most APTs.

## 2.2 Rule-Based Detection

In order to develop rule-based detections systems, it is important to model a range of threats and attacks so that rules and tripwires can be derived from the models. The most common models used in APT detection are the *Intrusion Kill Chain*[31][29] model or the *Attack Life Cycle*[36] model which are broadly similar. The different attack steps of the Attack Kill Chain model are as follows:

1. **Reconnaissance:** The target is researched identifying potential vulnerabilities, existing entry points and other potential weaknesses for exploitation.

2. **Weaponisation:** The attacker uses the knowledge gained from the reconnaissance stage to target an attack; this can include developing new malware and tools or modifying pre-existing tools.

3. **Delivery:** The attacker now attempts to establish access to the target's network by some delivery techniques including attacks on public servers, phishing attempts

or infected devices to name a few.

4. **Exploitation:** Once the attacker has established an initial foothold into the network they then exploit it by executing code or by jumping laterally across the network to reach their desired target.

5. **Installation:** Malware and other cyberweapons are installed on the target device(s).

6. **Command and Control (C2):** The malware installed is then remotely controlled by the attackers, allowing the attackers to modify their attack approach throughout and to maintain a presence inside the network.

7. **Action on Objectives:** The attacker now begins to achieve their initial objectives making use of the system access they have achieved. These objectives can include anything from data exfiltration to ransomware attacks to damage to physical systems.

Using the IKC framework, Atapour[6] was able to identify a series of weaknesses that APTs have at different stages of the IKC. This was then mostly focussed towards behaviour at the C2 stage that are detectable solely by network logs: namely HTTP GET and POST request loops and custom encryption protocols. Further features were then identified to further increase the confidence that the activity detected were due to an APT, with these including non-ASCII characters in responses and downloading executable files as of particular risk. Atapour also highlighted how the majority of

research into APT detection has been either anomaly-based or using machine learning approaches, indicating that a more signature-based approach could be used with their modelling approach.

Agrafiotis et al.[3] explored the use of rule-based detection in the area of insider attacks. This was done through the development of a tripwire grammar that was able to formalise typical security policies and known attack patterns. This was then later implemented and tested by Zhang et al.[40] who developed a series of attack trees for common attacks. These trees were then used these in a state machine to detect insider attacks from system log data, demonstrating the efficacy of tripwires as an attack detection method. This exact approach of using attack trees is yet to be applied to external threats and APTs, which is what this project will focus on. One issue when trying to apply this methodology to external threats is that the nature of insider threats tend to be significantly more static whereas the attack methodologies of APTs are constantly changing; to mitigate this we will give examples of known APTs being detected with existing datasets, whereas the attack trees used by such a system would need to be updated regularly as new attack data became available.

## 2.3 Advanced Persistent Threats

Advanced persistent threats are a sophisticated subset of cyber threats, with these types of attacks often having some of the most damaging impacts[20]. They differ from other threats such as malware or ransomware as they are significantly more methodical, slower to execute and more specifically targeted[4]. Most APT groups are believed

to be attached in some form to their corresponding nation state, however many still attack corporations for financial, espionage or geopolitical reasons[27]. One result of this association is most major APT groups have access to significant funding and resources, in turn allowing for rigorous reconnaissance and targeting of attacks compared to more generic malware.

APTs prove to be a significant challenge to detect/block by traditional firewalls or other traditional antivirus software due to the novel techniques that they make use of[12], with Stuxnet making use of four separate zero-day exploits[16]. Though it is typically better to prevent APT attacks from occurring through education and best practice[23], it is also important to develop techniques to detect APTs once initial breaches have occurred. Since APTs are persistent, there is typically a time period within which they can be detected before significant damage or data exfiltration occurs. In spite of attempts to remain covert, APTs will have detectable behaviour such as the creation/moving of files, upticks in activity and new network behaviours to list just a few. The majority of approaches to detecting APTs so far have used anomaly-based detection and of the rule-based approaches none have attempted to use attack pattern tripwires; this project will address this gap.

# 3 Methodology

## 3.1 Attack Trees

An attack tree is a directed acyclic graph $T$ consisting of a finite number of *states* $\Sigma$ and *transitions* $\Phi$ with transitions directed from one state to another. In the context of tripwires, the states are used to represent the current progress of an attack on a specific device with the transitions representing the individual tripwires that exist.

States $S$ are defined by the tuple $\langle id, tree \rangle$, where $id$ is a unique identifier and $tree$ indicates which attack tree the state belongs to. For simplicity, we will let $S_0$ be the universal starting state and also let $\Sigma_f$ be the set of final states reached at the end of their respective attack patterns.

We will define a transition $\phi$ as a 5-tuple $\langle S_{origin}, S_{to}, trigger, timeout, tree \rangle$ where:

- $S_{origin} \in \Sigma - \Sigma_f$ is the starting state for the transition $\phi$.
- $S_{to} \in \Sigma - \{S_0\}$ is the state that the transition $\phi$ directs to.
- $trigger : (log\ data) \rightarrow \{0, 1\}$ is the attack step that must occur to cause the transition to be taken in the attack tree. This takes the form of a function that takes a input of log data and returns a Boolean indicating whether the relevant attack step has occurred.
- $timeout : \mathbb{N}$ is the timeout value for the transition; if the no trigger event occurs within this timeframe then the attack tree is deemed to have expired and we generate a return to the state state $S_0$.

- *tree* indicates the attack tree that the transition and states belong to.

We also permit transitions to use conjunctions within our attack trees; eg. the following is a more complex but valid transition (and can have additional triggers) $\phi_a : S_x \wedge S_y \rightarrow S_z$.

Figure 1 depicts an example attack tree for the main installer of Stuxnet using file read and write data. Stuxnet is chosen here as it is probably the most famous example of an APT. The attack tree is formally $T = (\{S_0, S_1, S_2, S_3, S_4\}, \{\phi_0, \phi_1, \phi_2, \phi_3, \phi_4\})$, where the states are of the form $(S_n, T)$ and $\Sigma_f = \{S_4\}$. Each of the transitions $S_{origin}$, $S_{to}$ and descriptors of their triggers can be found below[16]:

- $\phi_0 : S_0 \rightarrow S_1$: read registry value "NTVDM TRACE" in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion \MS-DOS Emulation.
- $\phi_1 : S_1 \rightarrow S_2$: SetSecurityDescriptorDacl function called (Windows XP).
- $\phi_2 : S_1 \rightarrow S_3$: SetSecurityDescriptorSacl API used (Windows Vista and later).
- $\phi_3 : S_2 \rightarrow S_4$: *%SystemDrive%\inf\mdmcpq3.PNF* and *%SystemDrive%\inf\oem6C.PNF* files created.
- $\phi_4 : S_3 \rightarrow S_4$: *%SystemDrive%\inf\mdmcpq3.PNF* and *%SystemDrive%\inf\oem6C.PNF* files created.

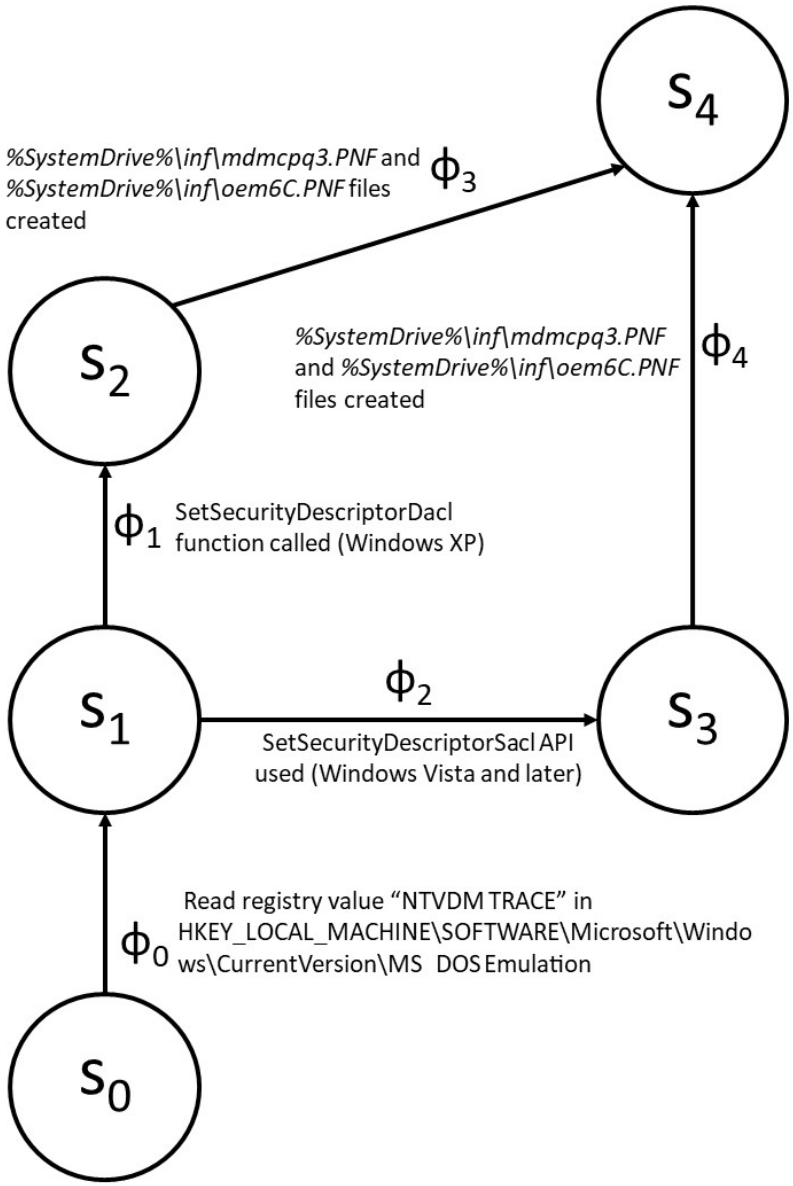


Figure 1: An attack tree for the installation of Stuxnet

The effectiveness of an IDS based on attack trees is strictly limited by how good the attack trees are; poor quality attack trees will result in either false positives (attack trees that also fit benign behaviour) or false negatives (attack trees that are insufficient to detect breaches). To minimise these erroneous results, there are three key design principles for attack patterns for this system:

- Sufficient Detail: Attack trees should capture enough detail within their triggers and have enough states such that a final state being reached is strongly indicative of malicious behaviour and unlikely to be a false positive.

- Minimal Size: Attack trees should be large enough to encapsulate sufficient detail indicating an attack, but no larger in order to reduce the likelihood of false negatives.

- Multiple Trees: One attack can have multiple detectable attack patterns; these should each have individual attack trees to increase certainty in detections.

## 3.2 Detection System

The detection system developed we develop here will be based upon that outlined by Zhang et al.[40], however it has been extended and reimplemented from scratch. It consists of a set $\mathbb{T} = \{T_1, ..., T_n\}$ of attack trees and a *model* recording the current detection status of each of the attack trees. The IDS will be given system logs as inputs and will maintain the current progress of each of the different attack trees as the logs are processed. If an attack tree reaches a final state, indicating a completed attack, an

alert of the form $\langle time, tree, trace \rangle$ will be generated.

In order to maintain the current progress of the attack trees, we will use a tuple *model*: $\langle$*state-model*, *transition-model*$\rangle$. The *state-model* consists of a series of records $s \in$ *state-model* of the form $\langle state, time \rangle$ which indicate which states have been reached and the most recent time at which these states have been reached eg. $\langle S_1,$ 01/01/2023 15:30$\rangle$ . By contrast, the *transition-model* consists of records of the form $\langle S_{origin}\ S_{to},\ time \rangle$ stating which transitions have occurred and at what time.

# 4 Implementation

The implementation of the IDS follows the principles introduced in the previous chapter. It is based upon the system proposed by Zhang et al.[40] but has been implemented from scratch and extended to include multithreading as was suggested in the further work.

Our approach to implementing multithreading focuses on improving the performance of the `update` function without changing its functionality. We approach this using a modified version of the Bag of Tasks design pattern with a number of `worker` threads running a version of the `update` function, using an `AtomicInteger` and a lockable `model` variable in order to correctly distribute tasks and the correct program state between the threads. We also extended the model tuple with the index of the log entry that caused the most recent transition. The pseudocode for the worker threads is below in Figure 2 and generally similar to the single-threaded `update` function.

```
WORKER THREAD
(i, m) <- controller.getAndIncIM

while (i < length of log file)
  updated <- False // model hasn't been updated yet by this iteration
  STATEMODEL <- m.state
  TRANSITIONMODEL <- m.transition
  UPDATEDMODEL <- m // variable to update if transitions occur
  now <- data(i).time // get log timestamp
  targetTransitions <- []
  timeoutTransitions <- []

  for each s in STATEMODEL: // s-tuple of state and commitTime
    for each c in ATTACKPATTERN[s.id].children: // child state of s
      if ATTACKPATTERN[s.id].trigger[c](data[i]): // transition occurred
        updated <- true
        if data(i).timestamp < s.commitTime + ATTACKPATTERN[s.id].timeout[c]: // expired so state deletion required
          timeoutTransitions.add({"origin": s.id, "to": c.id})
        //STATEMODEL.remove(all tuples with tuple.id = t.origin
        else: // not expired so new state and transition required
          targetTransitions.add({{"origin": s.id, "to": c.id}})

  for each t in timeoutTransitions:
    // state has timed out so can be deleted
    STATEMODEL.remove(the tuple with tuple.id = t.origin)

  for each t in targetTransitions:
    // Transition can be added/updated with most recent time it occurred at
    TRANSITIONMODEL.update(t.origin + ' ' t.to, now)
    // Add new state to STATEMODEL or update the commitTime to now
    STATEMODEL.update(t.to, now)
    if t.to.isOutput:
      // Reached an output state so trace can be generated
      UPDATEDMODEL = (STATEMODEL, TRANSITIONMODEL)
      trace <- getTrace(t.to, UPDATEDMODEL)
      output the trace

  if updated:
    // send updated model to the controller
    UPDATEDMODEL = (STATEMODEL, TRANSITIONMODEL)
    controller.sync(i, UPDATEDMODEL)

  controller.getAndIncIM // move onto next log entry
```

Figure 2: The pseudocode for the worker threads

The `controller` object offers two functions in order to co-ordinate the workers. `incAndGetIM` is used when the worker thread is ready to process the next log entry, returning the index of that log entry as well as the current state and transition models. `sync` is called when a worker thread completes an update and the `updatedModel` needs to be passed to the controller in case the `updatedModel` should overwrite the `model`. This overwriting is done if the index of the `updatedModel` is less than that of the `model` in order to ensure correctness; an example of this can be see in Figure 3.

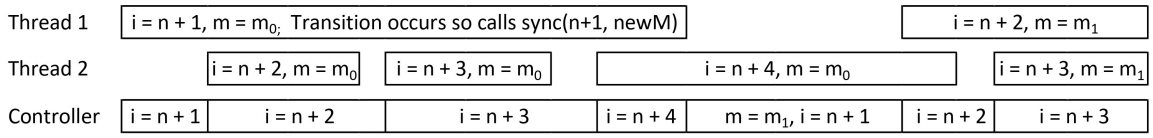


Figure 3: A diagram showing the behaviour of 2 threads and the controller when a `controller.sync` is called

The pseudocode for the controller can be found on the following page in Figure 4.

```
CONTROLLER Object
// store most upto date values of the index and model for distribution
    and synchronization between threads
aInt <- AtomicInteger(-1)
LOCALMODEL <- m
modelLock <- Lock

// returns the most upto date model and the next index
def incAndGetIM:
  modelLock.acquire
  out <- (aInt.incrementAndGet, LOCALMODEL)
  modelLock.release
  return out

// takes an updated model and checks if this should be set as the
    central model, resetting the index to the one immediately
    following the model
def sync(newVal, updatedModel):
  modelLock.acquire;
  if (newVal <= aI.get):
    LOCALMODEL <- updatedModel
    aInt <- newVal
  modelLock.release
```

Figure 4: The pseudocode for the controller class

Both functions complete all their reads and writes with the `modelLock` lock acquired; the only action completed without the lock is to return the values in `incAndGet`. This is to ensure thread safety - ie. that only a single thread can read or write to the `modelLock` variable at any one time. This protection is unnecessary for the operations on `aInt` since it is atomic and thread safety is guaranteed by hardware, however these operations are included inside the `modelLock` in order to avoid race conditions by allowing them to only be updated or gotten as a pair.

As a concurrent system, this system fulfils the required correctness properties. It satisfies liveness as the worker threads iterate through the log file data, processing

it before returning the final state of the model once all of the log file has been iterated over. We will demonstrate safety through an invariant. For simplicity, we will let $thread[0..noWorkers)$ be an array containing the worker threads and $thread[x].i$, $thread[x].m$, $thread[x].updated$ and $thread[x].updatedModel$ return the integer and model received by the thread when it last called `controller.getAndIncIM`, whether a transition occurred and the new model produced(if applicable) respectively. We will also let $complete[0..noWorkers)$ be the event that the corresponding thread is about to complete its latest iteration but hasn't yet called `controller.getAndIncIM` Finally, we will let $C(data, attackPattern)$ and $Sd(data, attackPattern)$ return the models obtained when running the concurrent and single threaded algorithms respectively on log file `data` and a given attack tree and let $LM$ and $aP$ be abbreviations for `LOCALMODEL` and `attackPattern`.

$$Invariant : \forall x.\ (0 <= thread[x].i < data.length + noWorkers) \wedge$$

$$(0 <= aInt.get <= data.length + noWorkers) \wedge$$

$$\forall ind.\ ((0 <= ind < data.length) \wedge$$

$$C(data[0..ind),\ aP) == S(data[0..ind),\ aP)) \wedge$$

$$completed[x] \rightarrow (min(thread[x].i) \rightarrow ((\neg updated \wedge LM == C(data[0..i),\ aP)) \vee$$

$$(updated \wedge LM == thread[x].updatedModel == C(data[0..i),\ aP))))$$

# 5 Evaluation

## 5.1 Datasets

Due to the advanced and highly specific nature of APTs and the difficulty in anonymising complex network data few APT specific datasets exist[36], with this leading researchers to either use more general datasets or create their own synthetic datasets. We use a combination of both during our testing, with each dataset consisting of either a large pcap or large log file which is then parsed and input into the system.

We also produced our own semi-synthetic dataset in order to test APT behaviours from other phases of the IKC. As identified by Atapour et al.[6] HTTP GET/POST loops during the C2C stage are an identifiable and detectable aspect of APTs. We exploit this by taking a network capture containing benign behaviour and use a python script to randomly insert HTTP entries as though one of the network computers was infected by Stuxnet. Stuxnet used *www.mypremierfutbol.com* and *www.todaysfutbol.com* as its command and control servers, hence HTTP communications between a device and either address would have been almost certain indicators of infection[16][39]. This attack tree is shown in Figure 5.

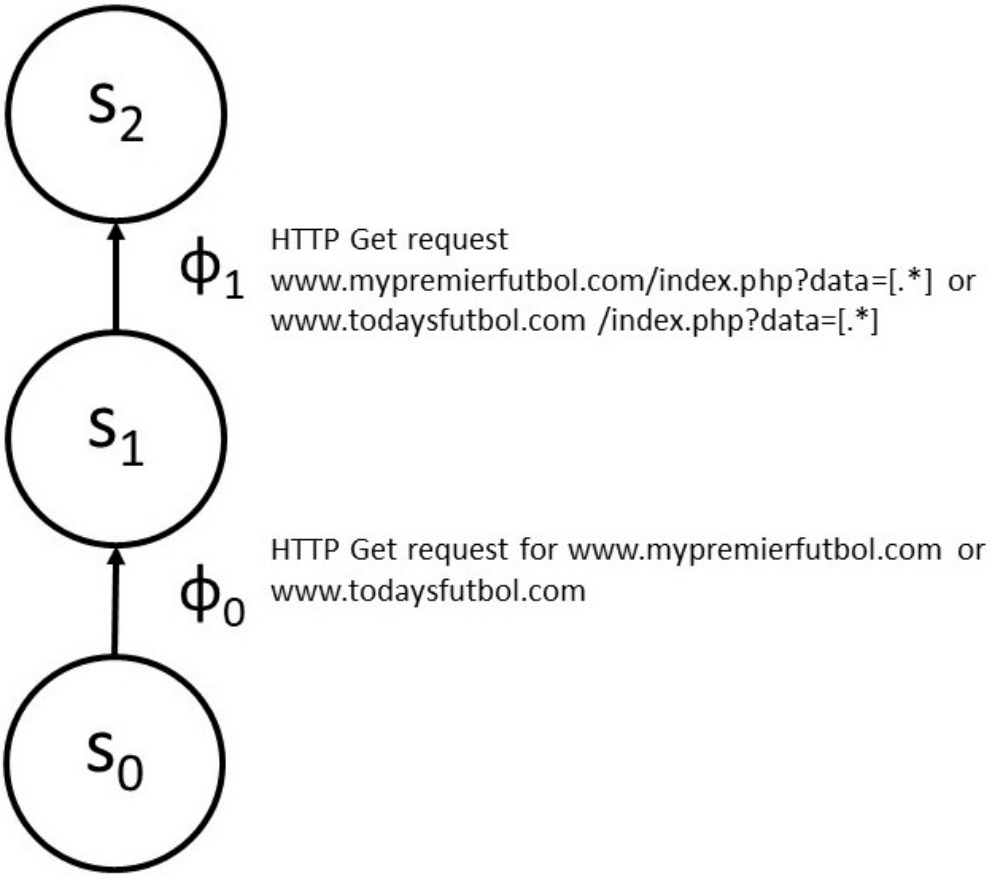


Figure 5: An attack tree containing Stuxnet's HTTP calls to its command centre

## 5.2 EternalBlue

We will now focus on EternalBlue as an example of how we use log data to detect APTs. This test scenario for the IDS involves detecting use the EternalBlue exploit, a zero-day exploit in the SMB protocol implementation on Windows first publicly leaked in 2017 by the "Shadow Brokers"[7]. It was most famously exploited by the WannaCry and NotPetya ransomwares[28], but has been exploited by APT groups such as Threat Group-3390[15] and APT28[35][11]. Though EternalBlue is just an exploit and not an

APT, it is used here as an example of how smaller features of APTs can be effectively detected by our IDS, allowing the APT itself to be detected.

Analysis of the EternalBlue exploit gives a series of clear attack steps[22] [32]:

1. After the initial SMB handshakes a large *NT Trans Request* packet is sent by the attacker, consisting of a sequences of NOPs is sent to the target.

2. The large size of the initial *NT Trans Request* request leads to a number of *Trans2 Secondary Request* packets being sent.

3. A *Trans2 Response* is received by the attacker with *NT Status: STATUS_INVALID_PARAMETER* indicating that the buffer overflow has been triggered correctly.

4. Another *Trans2 Response* packet is received by the attacker, this time with *Multiplex ID: 82* indicating that the payload has been successfully installed.

These attack steps allow us to generate the attack tree in Figure 6 in order to detect EternalBlue exploits. To test this, we will use three different known positive datasets and three known negative datasets all with significant SMB traffic. Of the positive datasets, one was generated by WannaCry[11] and the other two were only the successful exploitations of the vulnerability[38][32]

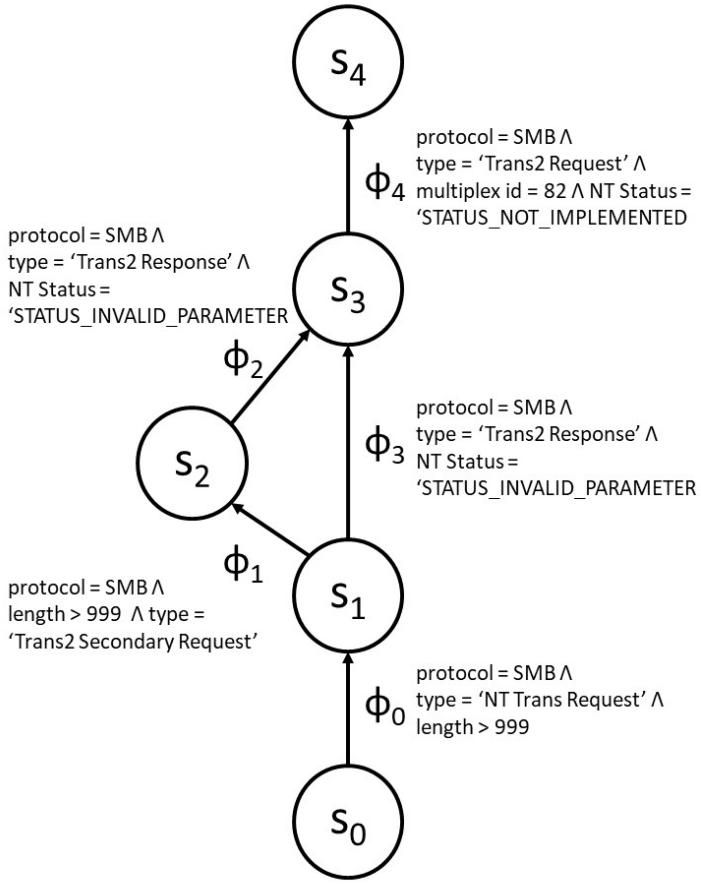


Figure 6: The attack tree developed for the EternalBlue vulnerability with the tripwires displayed

# 6 Discussion

## 6.1 Evaluation Results

The attack trees above (and others not included within this report) were encoded within the IDS. The IDS was then tested on a range of input data from single attack only datasets to significantly larger and more noisier network logs as well as our own

Table 1: Evaluation results using the EternalBlue attack tree in Figure 6

| Dataset | Exploit Present | Number of transitions $\lvert\phi_0\rvert$ | $\lvert\phi_1\rvert$ | $\lvert\phi_2\rvert$ | $\lvert\phi_3\rvert$ | Detected |
|---|---|---|---|---|---|---|
| WannaCry Attack[11] | Yes | 14 | 220 | 18 | 7742 | Yes |
| EternalBlue.pcp[38] | Yes | 3 | 48 | 6 | 1 | Yes |
| EternalBlue Success[32] | Yes | 2 | 31 | 2 | 1 | Yes |
| SMB Torture[1] | No | 455 | 205 | 0 | 0 | No |
| CIC-IDS2017-Thursday[34] | No | 0 | 0 | 0 | 0 | No |
| CIC-IDS2017-Friday[34] | No | 0 | 0 | 0 | 0 | No |
| Semi Synthetic | No | 0 | 0 | 0 | 0 | No |

semi-synthetic dataset. The attack trees were developed and designed without view of the log files to avoid overfitting.

As can be seen in Table 1, the detection system was able to detect the attacks it was intended to (ie. had attack trees for) with no false positives or negatives. This is predominantly a result of the design principles behind the attack trees - the trees very specifically capture key attack steps so that a false positives are very rare; for example the EternalBlue example will only return a complete attack tree if the vulnerability has been fully exploited. Capturing only unavoidable attack steps and details inside the attack trees significantly reduces the likelihood of false negative for attacks that we have trees for. The trade off of having rigorously defined, specific attack trees is that these are then typically unable to detect novel attacks which use new exploits. However due to well documented issues with patching large systems[33][25] there is significant value in being able to detect known exploits with a high degree of accuracy and precision. This limitation could also be addressed by extending the IDS into a hybrid system that has a machine-learning based anomaly detection component, for example attack trees that alert if enough HTTP GET/POST loops occur to a suspicious domain.

It is also worth noting that the lack of suitable datasets did limit our ability to test how effective our IDS and attack trees were. The semi-synthetic log files that we created did model the documented HTTP behaviour of a device infected with Stuxnet accurately enough to be detected, though this is obviously more limited than a larger quantity of logs or an actual real-world data capture.

## 6.2 Scalability

Our implementation has two significant performance advantages over Zhang et al.'s IDS, with the first being use case. Since we are applying the tripwire method to external APT attacks across a system, we do not need to maintain either a model per user or models of historical behaviour. This reduces the memory complexity from $O(|U|(3|T|+|S|))$ to $O(|T|+|S|)$ where U was previously the number of employees and S and T are the size of the state and transition models respectively. Additionally, compared to detecting insider threats, we see far fewer transitions do to our attack trees being significantly more specific, resulting in the system needing to keep track of far fewer states and hence further reduced memory use. The computational complexity of each `update` remains the same at $O(|S * T|)$ since in the worst case we iterate over every possible transition for every state, however the real number of transitions is significantly lower.

The other key advantage our implementation has is multithreading, allowing for far more efficient processing of the log files. The advantages are significant as can be seen below, though these vary based on the inputs as the concurrent implementation performs better when there are fewer transitions occurring within the log file. The

speed ups achieved through multithreading are limited by the fact that the threads effectively need to sync on every transition, but this is still faster than single-threaded performance in testing as seen below.

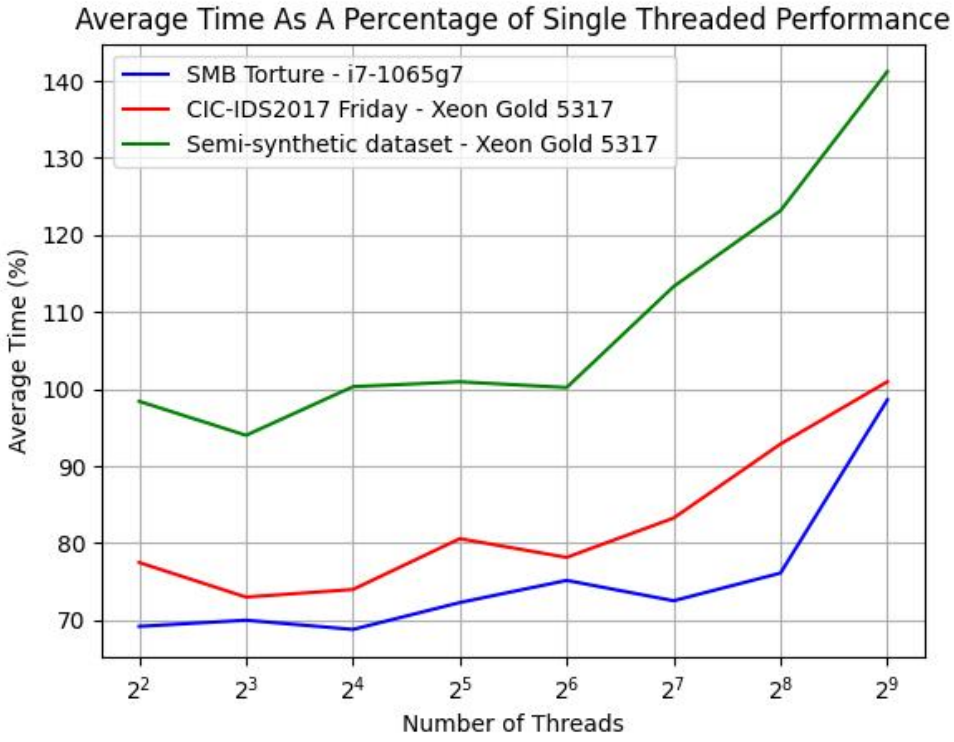


Figure 7: The performance of the multithreaded system across multiple datasets and processors compared to equivalent single-threaded performance

# 7 Conclusion

In this project we have successfully developed an intrusion detection system that uses attack trees in order to detect advance persistent threats. The system completes some of Zhang's further work by implement multithreading to ensure good performance within our system, whilst also ensuring that the multithreading is safe and that there is no risk

of deadlocking. Through testing and evaluation, we demonstrated that the system can perform efficiently and effectively, detecting known behaviours of APTs to a very high degree of accuracy. For future work improving the visual interface offered by the detection system would make it significantly more usable by non-specialists. Additionally, the tripwires approach could be used in conjunction with and anomaly-based detection system in order make the system significantly more effective at detecting novel APTs.

# References


[1] May 2023. URL: https://wiki.wireshark.org/SampleCaptures#server-message-block-smb-common-internet-file-system-cifs.

[2] Ioannis Agrafiotis et al. "A Tripwire Grammar for Insider Threat Detection". In: MIST '16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 105–108. ISBN: 9781450345712. DOI: 10.1145/2995959.2995971. URL: https://doi.org/10.1145/2995959.2995971.

[3] Ioannis Agrafiotis et al. "Formalising policies for insider-threat detection: A tripwire grammar". In: 8 (Jan. 2017), pp. 26–43.

[4] Feras Almatarneh. "Advanced Persistent Threats and its role in Network Security Vulnerabilities". In: *International Journal of Advanced Research in Computer Science* 11.1 (2020), pp. 11–20. ISSN: 0976-5697. DOI: 10.26483/ijarcs.v11i1.6502. URL: http://www.ijarcs.info/index.php/Ijarcs/article/view/6502.

[5] Abdulaziz Alshammari et al. “Classification Approach for Intrusion Detection in Vehicle Systems”. In: *Wireless Engineering and Technology* 09 (Jan. 2018), pp. 79–94. DOI: 10.4236/wet.2018.94007.

[6] Cheyenne Atapour, Ioannis Agrafiotis, and Sadie Creese. “Modeling advanced persistent threats to enhance anomaly detection techniques”. In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 9 (Dec. 2018), pp. 71–102. DOI: 10.22667/JOWUA.2018.12.31.071.

[7] BetaFred. *Microsoft Security bulletin MS17-010 - critical*. 2017. URL: https://learn.microsoft.com/en-us/security-updates/securitybulletins/2017/ms17-010.

[8] Parth Bhatt, Edgar Toshiro Yano, and Per Gustavsson. “Towards a Framework to Detect Multi-stage Advanced Persistent Threats Attacks”. In: *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. 2014, pp. 390–395. DOI: 10.1109/SOSE.2014.53.

[9] Beth E. Binde, Russ McRee, and Terrence J. O’Connor. *Assessing Outbound Traffic to Uncover Advanced Persistent Threat*. May 2011. URL: https://web.archive.org/web/20130626233122/https://www.sans.edu/student-files/projects/JWP-Binde-McRee-OConnor.pdf.

[10] Gretchen Bueermann and Seán Doyle. Jan. 2023. URL: https://www3.weforum.org/docs/WEF_Global_Security_Outlook_Report_2023.pdf.

[11] Hongsong Chen, Zhaoshun Wang, and Aaron Zimba. *Data for: Bayesian network based weighted apt attack paths modeling in cloud computing*. Mar. 2019. URL: https://data.mendeley.com/datasets/8ts23dp2zg/1.

[12] Copado. Dec. 2022. URL: https://www.copado.com/devops-hub/blog/11-characteristics-of-advanced-persistent-threats-apts-that-set-them-apart.

[13] Crowdstrike. Dec. 2020. URL: https://www.crowdstrike.com/resources/reports/crowdstrike-services-cyber-front-lines-2020/.

[14] Adrian Duncan, Sadie Creese, and Michael Goldsmith. "A Combined Attack-Tree and Kill-Chain Approach to Designing Attack-Detection Strategies for Malicious Insiders in Cloud Computing". In: *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. 2019, pp. 1–9. DOI: 10.1109/CyberSecPODS.2019.8885401.

[15] Robert Falcone and Tom Lancaster. *Emissary panda attacks Middle East Government Sharepoint servers*. Mar. 2020. URL: https://unit42.paloaltonetworks.com/emissary-panda-attacks-middle-east-government-sharepoint-servers/.

[16] Nicholas Falliere, Liam O. Murchu, and Eric Chien. Nov. 2010. URL: https://www.wired.com/images_blogs/threatlevel/2010/11/w32_stuxnet_dossier.pdf.

[17] Dennis Fisher. *Fully secure systems don't exist*. Apr. 2015. URL: https://threatpost.com/fully-secure-systems-dont-exist/112380/.

[18] Peng Gao et al. "SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection". In: *27th USENIX Security Symposium (USENIX*

*Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 639–656. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/gao-peng.

[19] Xueyuan Han et al. "UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats". In: *CoRR* abs/2001.01525 (2020). arXiv: 2001.01525. URL: http://arxiv.org/abs/2001.01525.

[20] Simon Heron. *Five notable examples of advanced persistent threat (APT) attacks*. URL: https://www.getsafeonline.org/business/blog-item/five-notable-examples-of-advanced-persistent-threat-apt-attacks/.

[21] Eric Hutchins, Michael Cloppert, and Rohan Amin. "Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains". In: *Leading Issues in Information Warfare & Security Research* 1 (Jan. 2011).

[22] Ali Islam, Nicole Oppenheim, and Winny Thomas. *SMB exploited: WannaCry use of "EternalBlue"*. May 2017. URL: https://www.mandiant.com/resources/blog/smb-exploited-wannacry-use-of-eternalblue.

[23] Kaspersky. Apr. 2023. URL: https://www.kaspersky.com/resource-center/threats/advanced-persistent-threat.

[24] David Kushner. "The real story of stuxnet". In: *IEEE Spectrum* 50.3 (2013), pp. 48–53. DOI: 10.1109/MSPEC.2013.6471059.

[25] Robert Lemos. *Three years after WannaCry, ransomware accelerating while patching still problematic*. May 2020. URL: https://www.darkreading.com/attacks-breaches/three-years-after-wannacry-ransomware-accelerating-while-patching-still-problematic.

[26] Mandiant. 2013. URL: https://www.mandiant.com/sites/default/files/2021-09/mandiant-apt1-report.pdf.

[27] MITRE. 2023. URL: https://attack.mitre.org/groups/.

[28] MITRE. 2023. URL: https://attack.mitre.org/techniques/T1210/.

[29] Hamad Al-Mohannadi et al. "Cyber-Attack Modeling Analysis Techniques: An Overview". In: *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*. 2016, pp. 69–76. DOI: 10.1109/W-FiCloud.2016.29.

[30] Daesung Moon et al. "DTB-ids: An intrusion detection system based on decision tree using behavior analysis for preventing apt attacks". In: *The Journal of Supercomputing* 73.7 (2015), pp. 2881–2895. DOI: 10.1007/s11227-015-1604-8.

[31] Weina NIU et al. "Modeling Attack Process of Advanced Persistent Threat Using Network Evolution". In: *IEICE Transactions on Information and Systems* E100.D (Oct. 2017), pp. 2275–2286. DOI: 10.1587/transinf.2016INP0007.

[32] OTW. *Network Forensics, part 2: Packet-level analysis of the NSA's EternalBlue exploit*. Jan. 2020. URL: https://www.hackers-arise.com/post/2018/11/30/network-forensics-part-2-packet-level-analysis-of-the-eternalblue-exploit.

[33] Cooper Quintin and Eva Galperin. *Why the patching problem makes us Wannacry.* July 2021. URL: https://www.eff.org/deeplinks/2017/05/why-patching-problem-makes-us-wannacry.

[34] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization". In: *International Conference on Information Systems Security and Privacy*. 2018.

[35] Lindsay Smith and Ben Read. *APT28 Targets Hospitality Sector, Presents Threat to Travelers.* Aug. 2017. URL: http://web.archive.org/web/20170829065819/https://www.fireeye.com/blog/threat-research/2017/08/apt28-targets-hospitality-sector.html.

[36] Branka Stojanović, Katharina Hofer-Schmitz, and Ulrike Kleb. "APT datasets and attack modeling for automated detection methods: A review". In: *Computers & Security* 92 (2020), p. 101734. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2020.101734. URL: https://www.sciencedirect.com/science/article/pii/S0167404820300213.

[37] Catherine Thorbecke. *Gas hits highest price in 6 years, fuel outages persist despite Colonial Pipeline restart.* May 2021. URL: https://abcnews.go.com/US/gas-hits-highest-price-years-fuel-outages-persist/story?id=77735010.

[38] Johannes Ullrich. *ETERNALBLUE: Windows SMBv1 Exploit (Patched).* Apr. 2017. URL: https://isc.sans.edu/diary/ETERNALBLUE+Windows+SMBv1+Exploit+Patched/22304.

[39] Pennsylvania State University. URL: https://www.cse.psu.edu/%5C~trj1/cse443-s12/slides/cse443-lecture-22-stuxnet.pdf.

[40] Haozhe Zhang et al. "A State Machine System for Insider Threat Detection". In: *Graphical Models for Security*. Ed. by George Cybenko, David Pym, and Barbara Fila. Cham: Springer International Publishing, 2019, pp. 111–129. ISBN: 978-3-030-15465-3.

[41] G. Zhao et al. "Detecting APT Malware Infections Based on Malicious DNS and Traffic Analysis". In: *IEEE Access* 3 (2015), pp. 1132–1142. DOI: 10.1109/ACCESS.2015.2458581.