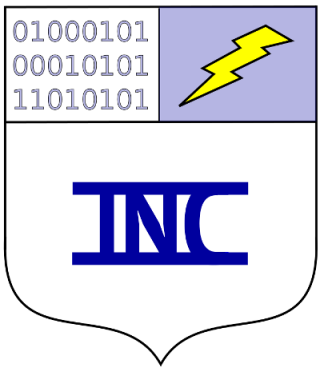
# Properties of Hacks

Brent Kirkpatrick *

August 17, 2020



Intrepid Net Computing




www.intrepidnetcomputing.com

*bbkirk@intrepidnetcomputing.com

# Document Revision History

**August 17, 2020** Published on www.intrepidnetcomputing.com.

**Abstract**

Hacking produces a variety of effects on the target computer, making hacking difficult to detect. In general, humans are better at detecting hacking than machines. On a given computer, a hack is given away by two things: 1) the uncertainty principle of computing and 2) indicators of compromise.

The uncertainty principle of computing is that the addition of any new machine code can produce unexpected results on a computer. The software that was already on the machine is perturbed in some fashion by the addition of the new machine code. Possibly the particular machine, software, and hack have not been tested together.

Indicators of compromise are used to determine whether a computer has been hacked. There is very little published on the topic of indicators of compromise, even though they are an industry-wide discussion.

Here, we introduce the uncertainty principle and categorize indicators of compromise as to the type of data and type of measurement that they represent. We introduce the distributed learning paradigm. This paradigm is useful for understanding network variables, for understanding the Internet, and for debugging network algorithms.

**Keywords:** learning, network algorithms, race condition, probability



# 1 Introduction

Hacking is defined as the introduction of machine code to a computer without permission. This can be accomplished from the terminal, from a resource, or from the network. While hacking is a verb, the discussion of the noun *hack* refers to the machine code that resides on the hacked computer or to the stand-alone exploit that introduces the machine code, which is called the *de novo exploit*. When the hack is *isolated*, either the version residing on the hacked computer, or the de novo exploit, this means that we have access to machine code.

Detecting whether a computer is hacked is a process of understanding the permissions and whether an unpermitted machine code was introduced. In principle, the proof of existence of unpermitted machine code need not require the isolation of that machine code. In the world of epianalysis, we generally look to isolate unpermitted machine code. In the world of analysis, we generally look for *de novo* exploits.

From the perspective of systems administrators, detection of hacking can be quite difficult. We have two tools at our disposal. The first is the uncertainty principle of computing and the second is indicators of compromise.

# 2 Background

Hacking and its detection are active conversations in computer systems. The main tool already in use is *indicator of compromise* (IOC) evebts. These are defined as measurable

events on computer systems. The IOC events are realized as variables. These consist of log records, user-experienced cases, network events, resource events, etc. For example, close examination of CPU resource use during an operating system (OS) update may reveal that there was resource use that was not approved by the operating system developers. This unpermitted resource use would be an indicator of compromise. The origin of the discussion of indicators of compromise appears is the applied areas of computing.

Digital forensics is done using sophisticated detection tools such as statistics on network traffic, statistics on firewalls, and searches of operating system files. These types of detection tools are slow, cumbersome, and used in relatively rare situations. The use of indicators of compromise is much more prevalent and is discussed in the setting of incident response.

The discussion of the uncertainty principle of computing is new. Here, we introduce the uncertainty principle and discuss categories of indicators of compromise. The formalisms that are fundamental to this discussion are formal methods, statistics, and formal languages. When we discuss how to assess a variable, we are referring to formal methods. When we discuss that variable is measurable, we are referring to measure theory. When we discuss the static or dynamic properties of a variable (i.e. constant or variable), we are referring to formal languages and statistics together. We conclude the discussion by introducing the *distributed learning* paradigm.

# 3 Results

We introduce the uncertainty principle of computing, first. Then we discuss useful categories of indicators of compromise. These categories help us understand potentially how to measure different indicators of compromise.

Once we have an event to consider, we discuss a novel paradigm for learning. This *distributed learning* paradigm lends itself to analysis of the Internet, to discovery of network properties, and to debugging network traffic. Furthermore, there are applications to computer security, such as detecting hidden channels.

## 3.1 Uncertainty Principle of Computing

Typically an operating system is shipped and installed in a stable state. This is the state somewhat matching the state for which the developers aimed. The introduction of new software, layered on top of the operating system, can always produce unexpected results. For example, a buggy application could crash the entire operating system. In the case of computer security, the introduction of foreign machine code can produce unexpected results.

The unexpected results introduced by new machine code is called, in natural language, the *uncertainty principle* of computing. This is seen from an absence of stability, rather than the presence of any particular event. Stability is the state that the software developers aim for when they release the software. The introduction of hacks or foreign machine code can produce an unstable situation.

Instability of the operating system is not necessarily measurable. For example system crashes might not be measured. If a system crash occurs without being noticed, like a tree falling unobserved in a forest, then the system crash might not measurable in seconds. However, the event of a system crash might be noticed in a broad time window such a day.

Similar non-measurable instabilities include: blue screens, wobbly mouse cursors, strange resource consumption, network anomalies, etc. An experienced computer user or systems administrator might notice some of these instabilities. An experienced user might tell the administrator that the computer is "strange" or "weird". Such word choice might mean that the user noticed some non-measurable instabilities.

For example, one can argue that in certain circumstances, network time can be unstable. In the case where network time diverges [1, 2], one can argue that it is not measurable. It is straightforward to demonstrate that network time can diverge. This is left as an excise to the reader.

As for a wobbly cursor, there is not computer code that can measure the precise movement of the cursor. The movement of the cursor is a complicated function of the graphical user interface (GUI) code and the graphics rendering engine. There is currently no way to measure the cursor's oddities. Between the resolution of the displays and the resolution of cameras that could video capture the cursor, there would be no way to measure the deviations.

In terms of resource consumption, there are few ways to get a base-line on a system. This means that the dynamic resource consumption is difficult to measure and evaluate statistically. However, an experienced systems administrator can often watch/manage the resources and observe problems by eye. The problems that are detected by eye are typically not measurable.

Doing digital forensics is provably undecidable even in the case of instabilities. In general, this means that there is no algorithm that can detect hacks [3]. In the case of specific instabilities or even measurable events, the problem of digital forensics is provably undecidable.

This discouraging result does not mean that we give up. On the contrary, it means that we look for feasible ways to precede, and we accept incremental progress. For example, when deciding if a particular instability is hacking related, we need to consider the possibility that hardware failures might be responsible for the observed instabilities. Sometimes this can be tested for statistically, if one knows the failure rate of a particular piece of hardware.

## 3.2 Properties of Indicators of Compromise

Indicators of compromise are measurable events that can be detected on computers in a network. The network is modeled as a directed graph with with nodes being computers and edges being network communication channels. The edges proceed from a computer/router/server that has network address translation (NAT) to the computers to which it delivers. This means that end-point computers are computer terminals having in-degree only and no out-degree.

The computers on which indicators of compromise are measured are often end-points, as is well known. In the case that the computer is a server, the measurable events might

| | **static** | **dynamic** |
|---|---|---|
| **network** | | time |
| | | date |
| | DNS domains | |
| | IP Addresses | |
| **node** | | user-experienced cases |
| | | pings |
| | | packet events |
| | | port events |
| | | protocol events |
| | | handshakes |
| | | queue saturation |
| | | system crashes |
| | file names | |
| | file hashes | |
| | settings | |
| | log events | |
| | host name | |
| | user names | |

Table 1: There are many useful indicators of compromise. These fall into several categories, network or node, and static or dynamic.

be quite sophisticated, such as queue saturation in the event of a denial of service (DOS) situation.

There are two dichotomies to look at with regards to indicators of compromise. These two dichotomies consider whether where a variable's value is stored and whether it is static or dynamic in memory.

One dichotomy is whether the variable is stored on the compute node or transmitted over the network. A node variable is stored locally on the compute node in a semi-static state. A network variable is transmitted in a dynamic fashion over the network. Network variables are subject to convergence properties on their values [1].

The other dichotomy is whether the variable is static or dynamic in memory. Variables that are static are relatively fixed or constant. On the other hand, variables that are dynamic are variable, meaning they are updated in real-time or in run-time. For example, the domain name service (DNS) domains names are static network variables, the Internet protocol (IP) addresses are static network variables, and pings hitting a firewall are dynamic node variables.

The type of variable hints at how we can assess its change on a computer as that computer goes from a not hacked state to a hacked state. A dynamic variable needs to be captured in time. A static variable is more amenable to assessment.

A dynamic variable might be assessed over time in a log event, and can be measured from memory with a memory dump (the OS call). In rare cases, flash freezing the memory might be a useful way to measure the instantaneous state of a dynamic variable.

On the other hand, a static variable is easier to asses when it changes. If there is a back-up server with the static values stored, and check-pointed, we might even know which day or hour the static value changed. One difficulty with storing static variables is knowing whether the storage or back-up mechanism is a transaction. If the storage option is a transaction, we can consider the record to be a measurement. However, if the storage option is not a transaction, then we are unable to do reliable statistics.

Certainly, we still need to consider alternative hypotheses. For example, the IP address, when assigned dynamically, changes every time the hardware address of the network card changes. This means that we need an alternative hypothesis to hacking when the IOC event occurs where an IP address changes. In the case of other indicators of compromise, we need to consider various alternative hypotheses.

We will briefly discuss several of these IOC events. For example, user-experienced cases are important because they are a sequence of program calls initiated and experienced by the user. These differ from use-cases, because they are from the execution path actually experienced by the user. As another example, queue saturation is interesting because it discusses a full queue in the network layer. The saturation event is when the queue becomes full. For yet another example, packet events are a category of events involving data packets on the network. These are actually many and varied, including packets from various protocols, ports, and applications. The final example given here is system crashes. These are difficult to assess, because the computer may crash before it dumps the contents of the memory. This means the system crashes may need to be documented by hand.

Network variables and node variables are different to assess. A node variable can be stored on a back-up server that does check-pointing. However, a network variable is more difficult to asses. In the case of network time, we already know that the time delivered by data packets to a single node might well diverge [1, 2]. In such a case, the divergence is mechanical and algorithmic, and has little to do with hacking. Other times the network time might oscillate. In this case, the oscillations might hide criminal activity. Even DNS look-up can have difficulties of convergence when there is DNS poisoning.

## 3.3 Analyzing IOC Events on One Node

When we consider IOC events and their events on one node in the Internet, we need to carefully correct for multiple testing. For example, we might monitor the traffic on a single port across time. When we we use several routers to track the same data-packet event (i.e. protocol, source, and destination), we need to correct for correlations. These correlations may come from the routing protocol, a streaming protocol, a service such as SSL, or some other source. At minimum, we need to use the Bonferroni correction for multiple testing.

Misinterpretation of the IOC events on a single node can lead to many false positives in detection. Generally, the power to detect hacking decreases as the number of multiple tests increases. This leads to false positives as the data points in time-driven events are misinterpreted as more important than they actually are.

There are many corrections for multiple testing, some are frequentist and some are Bayesian. If we test one queue several times, we can model the entire queue over time

and correct using the Poisson model. Alternatively, one could use permutation testing to correct for multiple testing. The Bonferroni correction is suggested for the ease of implementation and its guarantees. The Bonferroni correction is the most stringent, because it is based on the worst-case scenario, that the tests are completely independent.

The frequentist setting for the Bonferroni correction is that we have run $n$ correlated tests, and the results are perfectly correlated. The correction to the p-values is conservative. Let $p_i$ be the p-value obtained from one test that is run $n$ times. This means that we take the expectation $\mathbb{E}[p_i] = np_i = n$. Now, we can use this Bonferroni correction of the threshold, $t$, for the test:

$$t' = t/n.$$

Only the corrected test should be used when determining the results against the alternative hypothesis. This means that the Bonferroni correction removes the false positives.

In general multiple tests should be avoided, as they can obscure the signal. In particular, the Bonferroni is the most conservative correction for multiple testing. Running multiple tests is one way of looking for a signal in big data. To more easily find the signal, one should carefully choose the events, choose the space, and properly model the correlations.

## 3.4 Analyzing IOC Events on Multiple Nodes

The difficulty with comparing IOC events on multiple nodes in the Internet is that those IOC events are correlated. If we take two back-bone/infrastructure routers and look at data-packet events, we might see that some data packets pass through both routers. In this case, this leaves us with a correlation that must be corrected when computing statistics. Even in the case of hidden channels (i.e. those created by Tor), the explaining away phenomenon via the network topology, needs to be considered and corrected. Fortunately, the field of machine learning has the existing framework of graphical models which can model the correlations across the network.

**Distributed Learning Paradigm.** Many of our tests on multiple nodes can be phrased in terms of distributed learning. We will define this as a learning paradigm that draws data from distributed network sources. In some cases the distributed learning paradigm can be executed in a distributed fashion. In other cases, it might be executed on one node while drawing data from the network. In the first case, the distributed learning paradigm can use message passing across nodes via a dedicated port. This means that each node would be updating its part of the model, and, after convergence, the posterior can be computed. In the second case, the distributed algorithm might be an off-line algorithm, like an hidden Markov model (HMM), where the computation is done on a single centralized node that draws data from the network on a dedicated port. In the other cases, the paradigm looks more similar to standard machine learning.

**Distributed Learning of Network Variables.** Let's consider the case of network variables. In this setting we learn the marginal probabilities of a network variable at each node

in the network. This could be either a dynamic network variable or a static network variable. This setting lends itself to online learning.

The main difficulty is that the latency is not measurable. So the message passing algorithm will have race conditions. Hence, we recommend a dedicated port.

Another difficulty is obtaining the partition function from the network. This is due to at least two problems. One problem is that nodes might crash or be removed from the network. Another problem is that map reduce becomes difficult to implement.

**Distributed Learning of Channels.** In this setting the goal is to detect a channel that is embedded in a background of data packets. For example, one might try to detect a hidden channel routed by Tor. Alternatively, one might try to detect a streaming data channel as it passes through several infrastructure routers. This might distinguish some data packets in that channel from the other data packets passing through the routers.

The main difficulty in this type of learning is the partition function. The model can be of arbitrary complexity to model the topology of a portion of the Internet. However, computers are added and removed from the Internet in real-time, effectively changing the topology instantly. This means several things: 1) the partition function may not be constant, and 2) the probabilities may not be measurable, and 3) an off-line model may not be accurate. This means that the explaining-away phenomenon is quite important and a mistake in the topology of the network can be misinterpreted.

For example, we might use a Kalman filter to detect a channel, where the hidden states are the network topology, and the observations are a collected data set. In this case, min-cut/max-flow of the hidden-state model is critical to the running-time of the algorithm. One difficulty in using this filter is to model the background distribution.

Online learning is more difficult in this setting, as the implementation has to be at real-time speeds. That means that each update step needs to be extremely fast. An algorithm such as AdaBoost might be feasible, provided that the implementation is fast enough.

# 4 Conclusions

We have presented a new paradigm for learning, called distributed learning. This paradigm is potentially useful for understanding the Internet. It might help us debug network algorithms, understand routing protocols, examine the effectiveness of DNS, and help with computer security.

We have also presented a new discussion of how to detect hacking. This discussion is about the uncertainty principle of computing. This follows on an existing discussion of indicators of compromise. We add to the discussion of indicators of compromise by discussing categories of indicators of compromise.

New research directions include the possibilities of measuring indicators of compromise and using them for statistical analysis of network-wide hacking events. The collection of IOC events broadly across the network is already under consideration by the field. However,

analyzing that data statistically will require further research into frameworks for taking measurements and for modeling the data.

# Biography

Dr. Kirkpatrick has a bachelor's in computer science from Montana State University-Bozeman, a master's and a Ph.D. in computer science from the University of California, Berkeley with post-doctoral experience at the University of British Columbia. Dr. Kirkpatrick is a Professor Emeritus of Computer Science from the University of Miami and is an expert in deterministic and statistical computer algorithms. His main application area is the field of computational biology. Recently, he has specialized in algorithms for computer systems and computer security.

# References


[1] B. Kirkpatrick. Properties of network time. *Technical Report*, 2020.

[2] Y. Weiss. Correctness of local probability propagation in graphical models with loops. *Neural Computation*, 12:1–41, 2000.

[3] B. Kirkpatrick. Computer security is algorithmically intractable. *Technical Report*, 2018.