TEMPSMITTER

Temperature As Covert Channel

Adhokshaj Mishra

Security Researcher - Malware - Linux

July 17, 2020

Agenda

1. Legal disclaimer
2. Who am I?
3. Introduction
   - 3.1 Motivation behind research
4. CPU, Instruction Set and Power
5. Physical Layer Communication
   - 5.1 Naive Method
   - 5.2 Timing Problem
   - 5.3 Synchronization Problem
   - 5.4 Noise and Error
   - 5.5 Error Detection and Correction

Agenda

1. Design of Covert Channel
   - 1.1 Hunting Power Hungry Instructions
   - 1.2 Stressing the CPU
   - 1.3 Power De-optimization of Code
   - 1.4 Cooling the CPU Down
2. Encoding and Decoding Data
   - 2.1 Encoding data bits to heat pattern
   - 2.2 Running heat pattern on CPU
   - 2.3 Extracting data from CPU temperature
3. Demo
4. Countermeasures (seriously?)

Legal Disclaimer

The opinions expressed in the presentation and on the following slides are solely of mine, and not of my employer.

The technique(s) presented hereafter are offensive in nature; and are generally considered a criminal offence if practiced without proper authorization in place. It is presented here for educational purpose only. In other words, if you come to me saying that you are neck-deep in mess due to these techniques, I won't feel responsible at all.

Who am I?

- Independent security researcher, with deep interest in offensive malware techniques and cryptography.
- I love to attend and speak at various security conferences and meet-ups.
- You can contact me on LinkedIn (AdhokshajMishra), or via e-mail (me@adhokshajmishraonline.in)

Introduction: Motivation behind work

- "Fansmitter: Acoustic Data Exfiltration from (Speakerless) Air-Gapped Computers" [https://arxiv.org/abs/1606.05915]
- Limitations of Fansmitter: not all fans are software controlled; and those which are, need root/administrator privilege.

Introduction: Why Tempsmitter

- Because side channels are sexy.
- Fansmitter is for air-gapped systems. IPC between processes on same host is way more common, and so is networked infrastructure.
- Fansmitter needs specific privileges, and hardware support.

Tempsmitter tries to remove dependence on specific hardware, higher privileges; while providing a way to perform IPC on same host.

CPU

- A CPU is a physical is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions.
- Most CPUs used today are microprocessors (i.e. whole CPU is packed in a single IC)
- Most (if not all) CPUs used in complex machines used today are based on microprogrammed control unit, while other simple ones are based on hardcoded logic.

Instruction Set

- An instruction set is the complete set of all the instructions in machine code that can be recognized and executed by a CPU. The instruction sets are dictated by ISA (instruction set architecture) which is abstract model of a computer which is targeted by various implementations of same ISA.
- It can be CISC (Intel/AMD), RISC (ARM), MISC (many 8-bit uC), VLIW (SHARC), EPIC (under research).
- The abstract model can be a register machine, a stack machine (JVM), or a mix of both (Intel/AMD).

Power

- Thumb rule: the more work you do per unit time, the more power you need.
- The more complex ISA, the more power you need to run an implementation. This is why mobiles are mostly powered by ARM (RISC), while laptops, workstations and servers are powered by Intel and AMD (CISC/VLIW).
- The more complex instruction, the more power it will draw.
- Drawback: you cannot keep drawing high power for long time, otherwise magic smoke will leak from CPU (unless you have fancy liquid cooling).

Naive Method

- Dead simple mechanism: High = 1, Low = 0. It can be voltage, current, tension, luminosity or something else that can be controlled.
- Pretty simple design: all you need is an encoder (bits to high/low levels), a decoder (high/low levels to bits), and a clock on both ends.

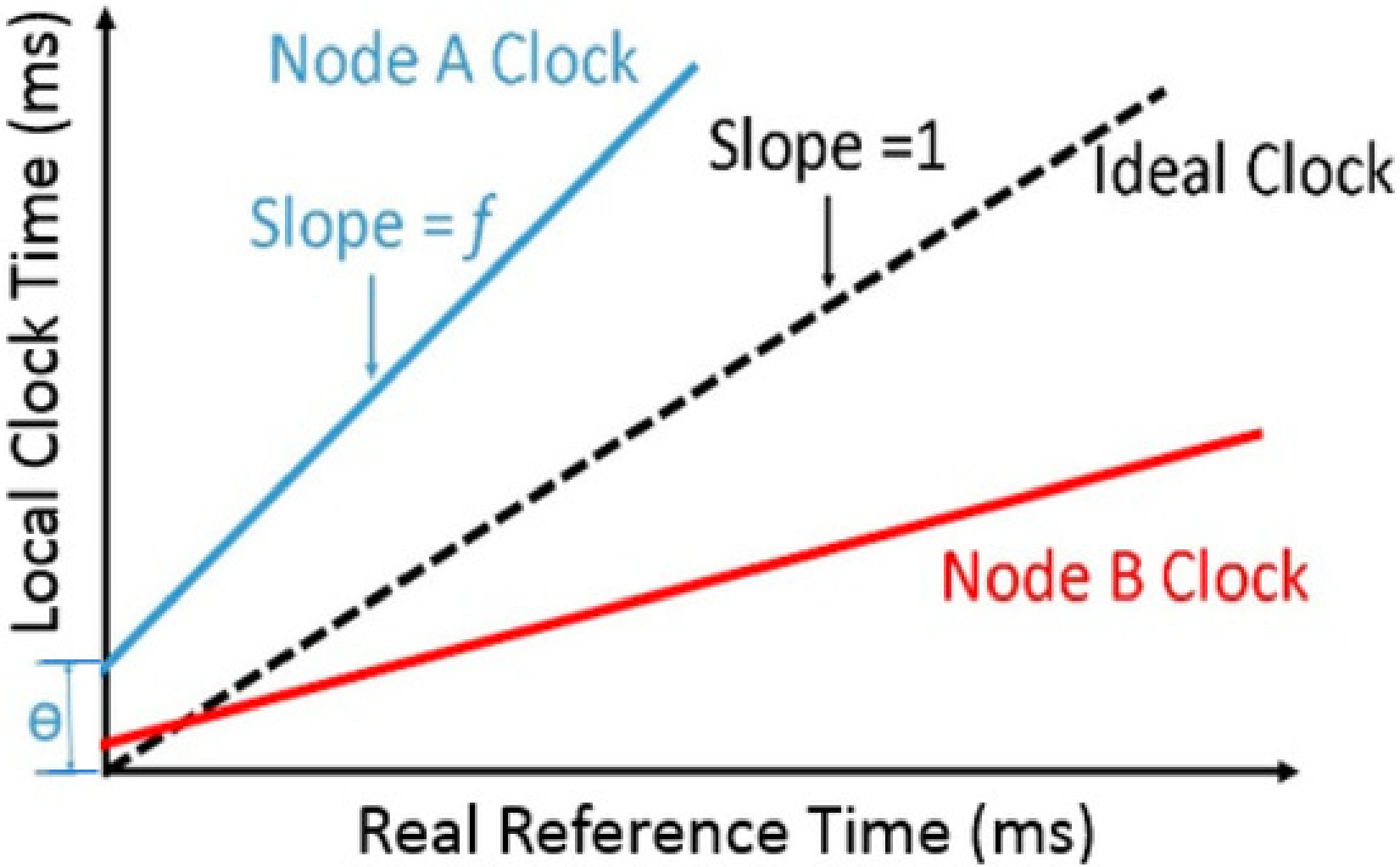
How hard can it be in practice, really?

Timing Problem

It turns out, it will be pretty hard.

- An ideal clock does not drift, but a practical clock drifts. By drift, we mean that the clock does not run at exact same frequency with an ideal reference clock. With long data transmission, you will either get repeated bits, or you will miss bits; depending upon which way your clocks are drifting.

Timing Problem



Clock Drift

Timing Problem

Since we ran into problem with two different clocks at different ends, let us just use one clock, which can send timing signal at both ends.

Problem solved, right?

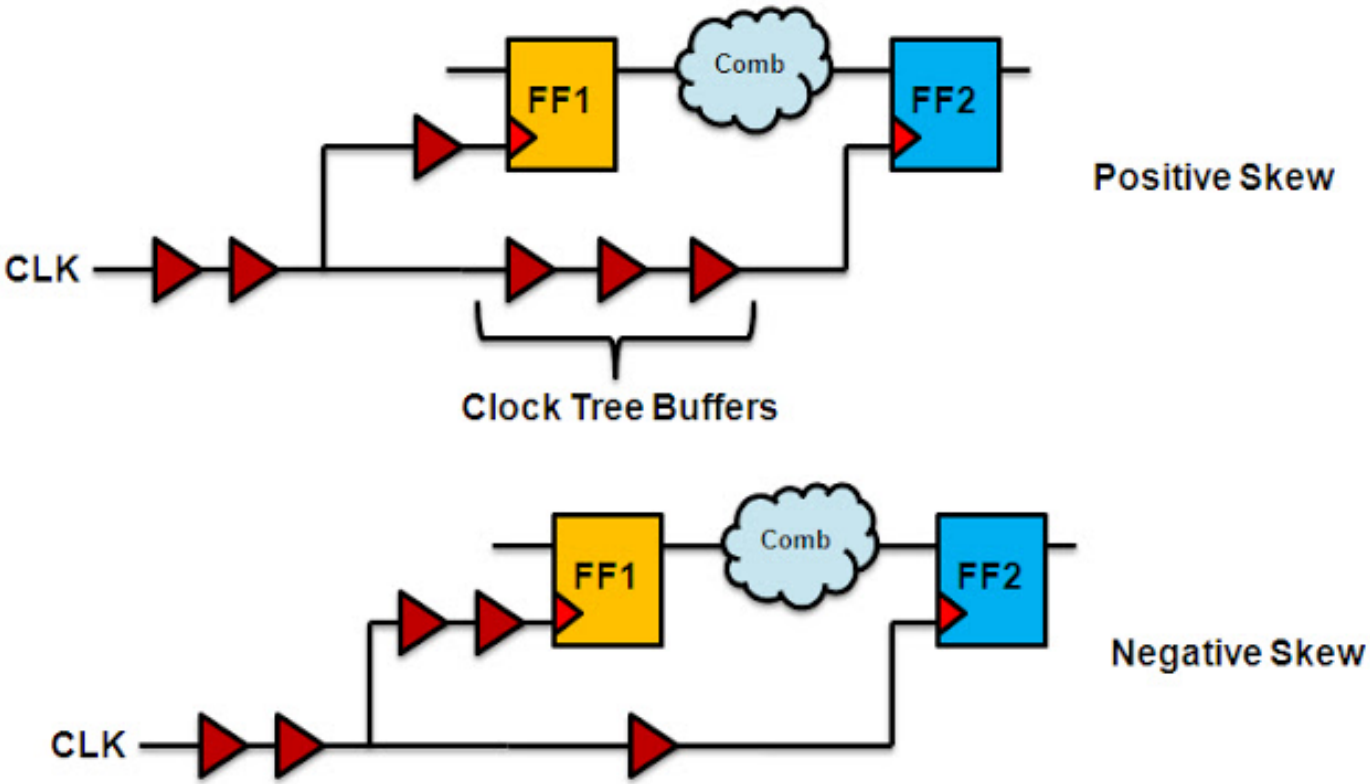
RIGHT?

Timing Problem

Unfortunately, using same clock source at both ends leads us to different problem: timing skew.

Timing skew is a phenomenon in synchronous digital circuit systems in which the same sourced clock signal arrives at different components at different times. The instantaneous difference between the readings of any two clocks is called their skew.

Timing Problem



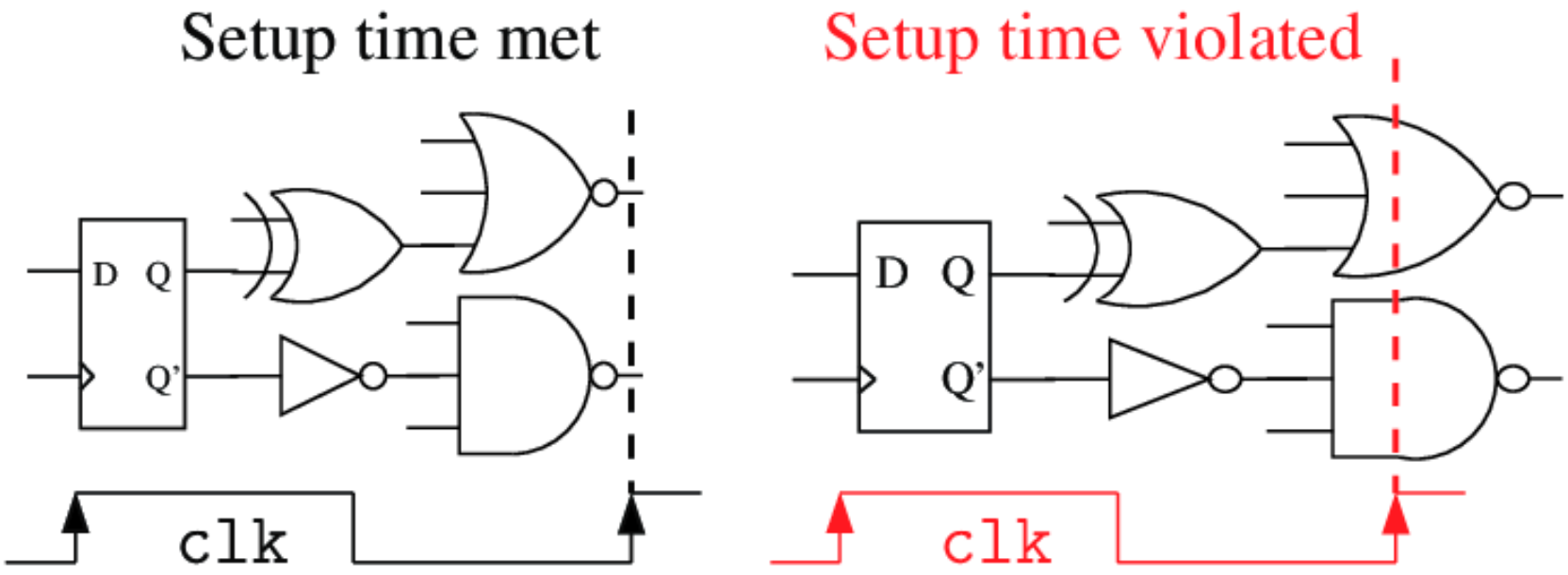
Timing Skew

Timing Problem

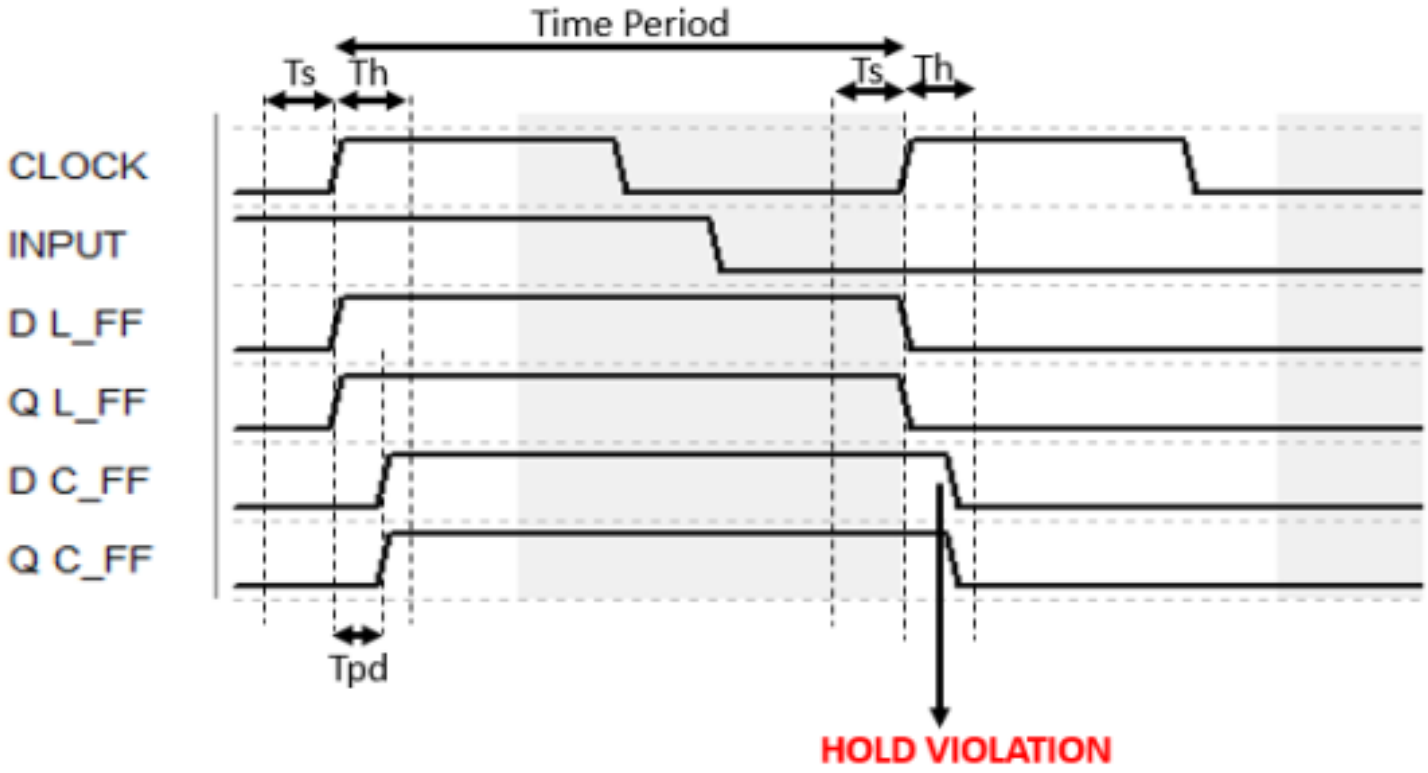
An ideal edge-triggered latch samples the data line instantaneously on one of the edges of the clock. However, in practice, the data must be valid for some finite amount of time around the clock edge. The time it must be fixed before the clock edge is called the setup time, and the time it must be fixed after the clock edge is called the hold time.

Violation of setup time is called setup violation (which is called by negative skew). Similarly, violation of hold time is called hold violation (which is caused by positive skew).

Timing Problem



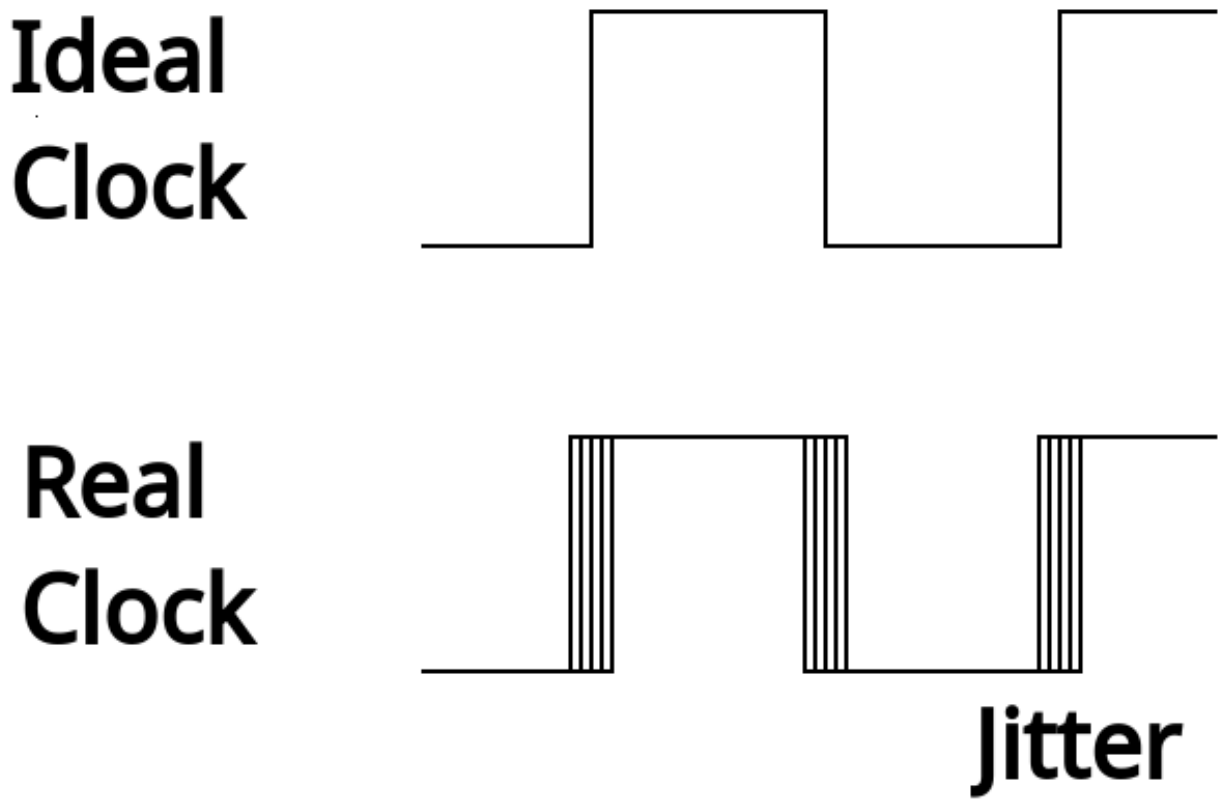
Timing Problem



Timing Problem

Due to static differences in the clock latency from the clock source to each clocked register, no clock signal is perfectly periodic, so that the clock period or clock cycle time varies even at a single component, and this variation is known as clock jitter. This is yet another variable in timing leading to uncertainties.

Timing Problem



Timing Problem

Can we even solve all these problems?

Timing Problem: Solution

Self-Clocking Signalling

Problems we have to solve now:

- How to send timing information to a reciever without a synchronized clock?
- How to send data WITH timing information reliably?

Self-Clocking Signalling

Sending timing information:

- A reference wave of constant frequency (f) can be used to transmit timing information. Reciever can sample the incoming signal at >2f, and can derive timing information from that.

Self-Clocking Signalling

Sending data information:

- Since we are relying on constant frequency of reference wave for timing information, we cannot use frequency modulation to encode data.
- Amplitude modulation is an option, but that is prone to interference. This will result in unreliable decoding due to noise, and therefore is not suitable.
- Phase modulation is another option, which is not prone to interference (not at level of amplitude modulation).

Self-Clocking Signalling

Modulation/De-modulation scheme:

- Since we are going to transmit digital bitstream over analog carrier signal, we need some digital modulation scheme.
- Since phase modulation is the only option to send data, we will settle with Phase Shift Keying.

Binary Phase Shift Keying

- It is simplest form of Phase Shift Keying, and uses two phases separated by 180^o.
- It handles the highest noise level or distortion before the demodulator reaches an incorrect decision.
- It can modulate only 1 bit per symbol, and therefore is not suitable for high data rate channels.
- Data is often differentially encoded before modulation.

Manchester Code

- Developed at University of Manchester, and used to store data on magnetic drums of Manchester Mark 1.
- It is a special case of binary phase-shift keying, where the data controls the phase of a square wave carrier whose frequency is the data rate.
- It ensures frequent state transitions, directly proportional to the clock rate; this helps clock recovery.

Manchester Code

Conventions for representing data:

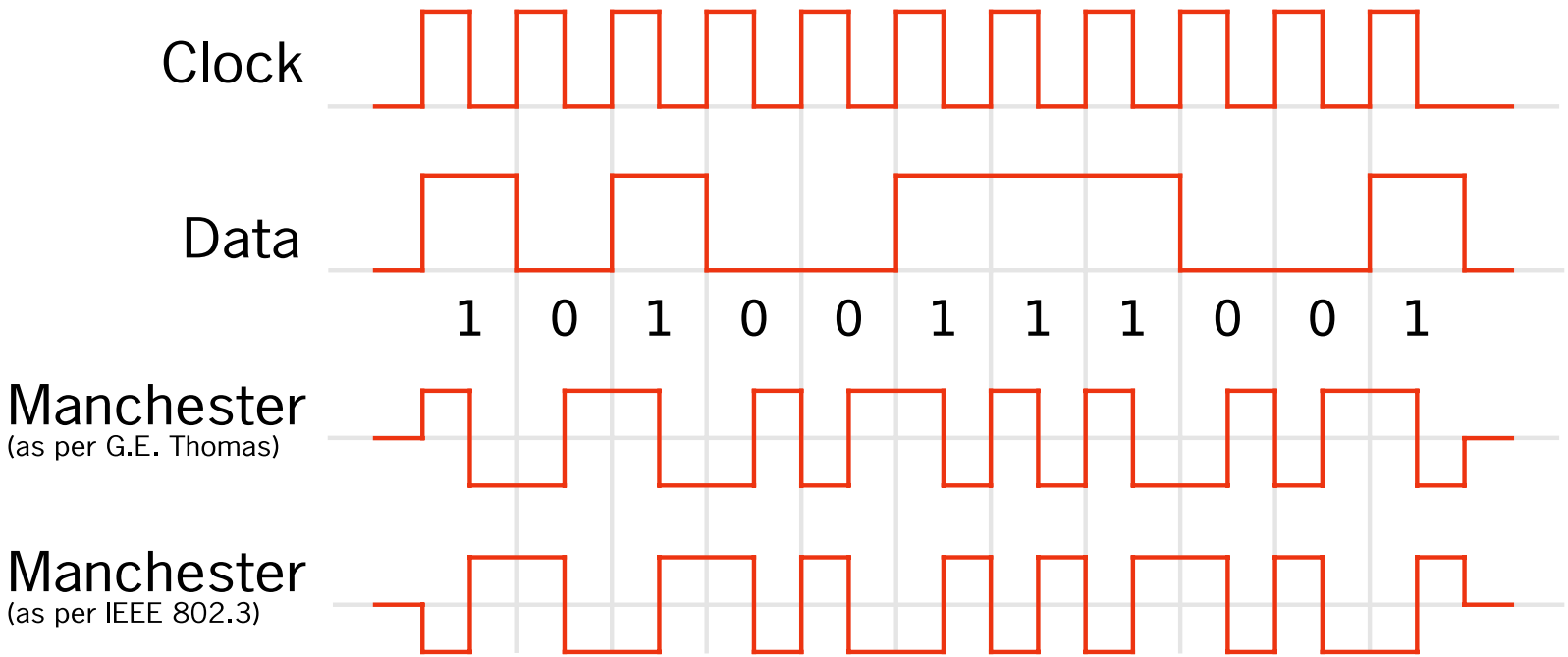
As per IEEE 802.3

- Logic 0: High $\rightarrow$ Low signal sequence
- Logic 1: Low $\rightarrow$ High signal sequence

As per G. E. Thomas

- Logic 0: Low $\rightarrow$ High signal sequence
- Logic 1: High $\rightarrow$ Low signal sequence

Manchester Code



Synchronization

How do we know exactly when data transmission starts?

Synchronization

- We can tell decoder where data starts via out of band signalling. This requires another signalling mechanism to transmit synchronization information.
- We can use in-band signalling to tell decoder when to start reading data. We can set a “magic signal sequence” which signals start of data transmission. Similarly, we can also have another “magic signal sequence” which signals stop of data transmission.

Self-Synchronizing Code

A self-synchronizing code is a uniquely decodable code in which the symbol stream formed by a portion of one code word, or by the overlapped portion of any two adjacent code words, is not a valid code word. In other words, a set of strings over an alphabet is called a self-synchronizing code if for each string obtained by concatenating two code words, the substring starting at the second symbol and ending at the second-last symbol does not contain any code word as substring.

Self-Synchronizing Code

Self-Synchronizing Code

In very simple terms, we need to figure out only two magic sequences, both of which are not valid encoding of any data. One of them can denote beginning of data, and other can denote end of data.

Self-Synchronizing Code

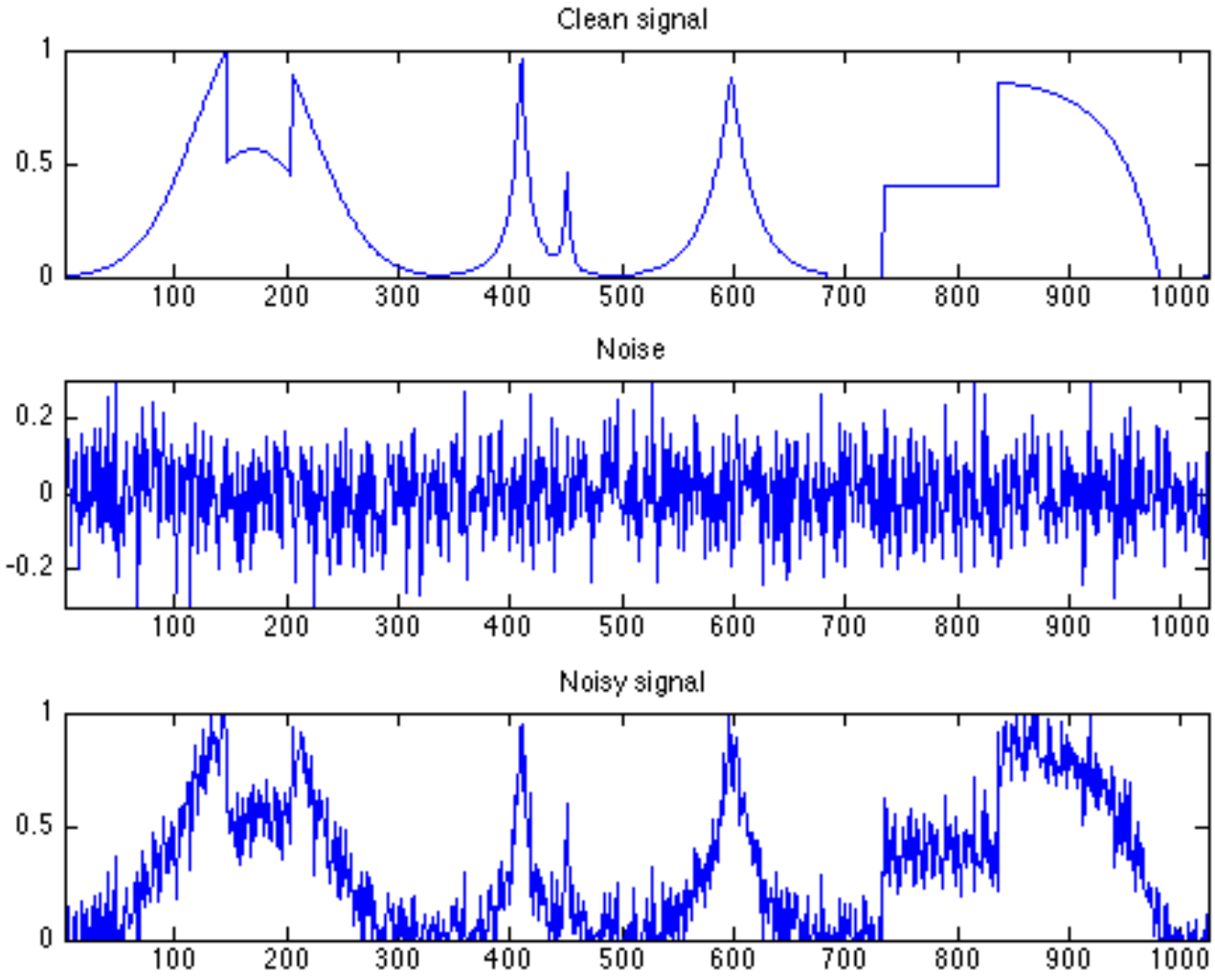
Examples:

- In UTF-8, bit patterns 0xxxxxxx and 11xxxxxx are used to mark the beginning of the next valid character

Noise

- It is an error or undesired random disturbance of a useful information signal.
- It may result from intermodulation, crosstalk, atmosphere, or cosmic radiations.
- It will corrupt the parameters of modulated waveform.

Noise


Clean signal
1
0.5
0
100
200
300
400
500
600
700
800
900
1000
Noise
0.2
0
-0.2
100
200
300
400
500
600
700
800
900
1000
Noisy signal
1
0.5
0
100
200
300
400
500
600
700
800
900
1000

Shannon-Hartley Theorem

Shannon-Hartley Theorem gives us a hard limit of speed at which we can communicate over a noisy channel.

The channel capacity C (the theoretical tightest upper bound on the information rate of data that can be communicated at an arbitrarily low error rate) using an average received signal power S through an analog communication channel subject to additive white Gaussian noise (AWGN) of power N:

$$C = B \log_2(1 + \frac{S}{N})$$

Shannon-Hartley Theorem

$$C = B \log_2(1 + \frac{S}{N})$$

Where

- C is the channel capacity in bits per second, a theoretical upper bound on the net bit rate (information rate) excluding error-correction codes;
- B is the bandwidth of the channel in hertz (passband bandwidth in case of a bandpass signal);
- S is the average received signal power over the bandwidth (in case of a carrier-modulated passband transmission, often denoted C), measured in watts (or volts squared);

Shannon-Hartley Theorem

$$C = B \log_2(1 + \frac{S}{N})$$

Where

- N is the average power of the noise and interference over the bandwidth, measured in watts (or volts squared);
- and S / N is the signal-to-noise ratio (SNR) or the carrier-to-noise ratio (CNR) of the communication signal to the noise and interference at the receiver (expressed as a linear power ratio, not as logarithmic decibels).

Noisy Channel Coding Theorem

Given a noisy channel with channel capacity C and information transmitted at a rate R, then if $R < C$ there exist codes that allow the probability of error at the receiver to be made arbitrarily small.

Noisy Channel Coding Theorem

If $R > C$, an arbitrarily small probability of error is not achievable. All codes will have a probability of error greater than a certain positive minimal level, and this level increases as the rate increases. So, information cannot be guaranteed to be transmitted reliably across a channel at rates beyond the channel capacity.

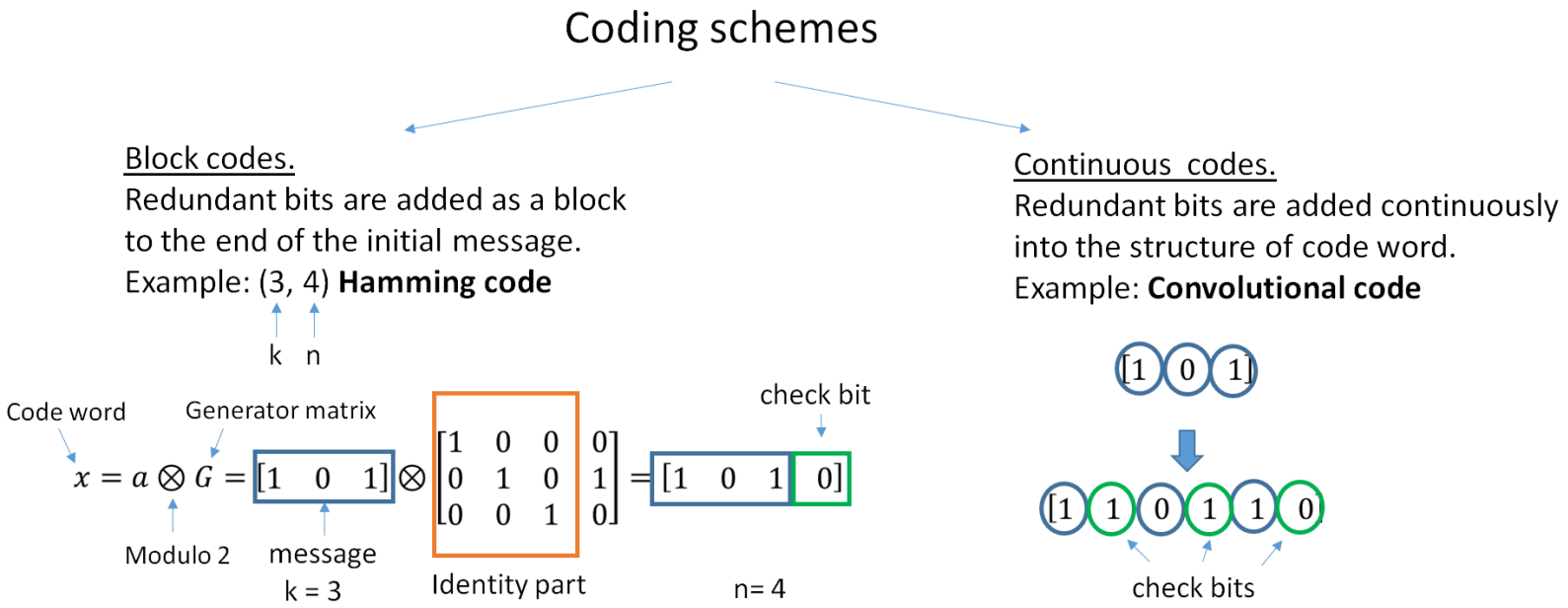
Noisy Channel Coding Theorem

In very easy terms, Noisy Channel Coding Theorem proves existence of error correction codes, which allow us to transmit data over noisy medium without corruption. This theorem has wide-range applications in communication, and data storage. It is of fundamental importance in information theory.

Error Correction Code

- Error correcting code (ECC) is used for controlling errors in data over unreliable or noisy communication channels. The central idea is the sender encodes the message with a redundant in the form of an ECC.
- In general, a stronger code induces more redundancy that needs to be transmitted using the available bandwidth, which reduces the effective bit-rate while improving the received effective signal-to-noise ratio.

Error Correction Code



Error Correction Code

Examples:

- Hamming Code
- Reed Solomon Code
- Convolutional Code

Covert Channel

In computer security, a covert channel is a type of attack that creates a capability to transfer information objects between processes that are not supposed to be allowed to communicate by the computer security policy.

Elements of Covert Channel

- A medium with controllable parameter. This can be timestamp of a file, metadata of some other component, system load, resource utilization etc.
- A mechanism to control chosen parameter with respect to time. In simple terms, we should be able to reliably affect timestamp/metadata/system load/resource utilizatione or whatever parameter we are choosing.

Elemets of Covert Channel

- Modulation and demodulation scheme. This is required to map data bits in state of parameter at sender's as well as receiver's side.
- Clocking and synchronization

Temperature as Covert Channel

- Medium: CPU Dice Temperature
- Controllable Parameter: Temperature
- Control Mechanism: Stressing the CPU, and relieving the stress
- Modulation and Demodulation scheme: Use any suitable scheme you are happy with
- Clocking and synchronization: Use any suitable scheme you are happy with

Hunting the Power Hungry Instructitons

Instruction Set Extensions supported by Intel Chips:

- x86
- AMD64
- SSE (Streaming SIMD Extensions)
- MMX
- AVX

Hunting the Power Hungry Instructions

Thumb Rule 1: The more work you do in per unit time, the more power it will draw.

Thumb Rule 2: The more complex instruction set, the more power it will daw.

Thumb Rule 3: The more complex instruction, the more power it will draw.

Hunting the Power Hungry Instructions

Some more useful tidbits:

- Intel instruction sets generally increase in complexity with time.
- CPUs mostly run underclocked @ ~800MHz-1.2GHz. If the load increases, CPU increases clock speed upto base clock speed (SpeedStep). If load persists further, CPU overclocks itself for short time (Turbo Boost).
- Power consumtion is roughly proportional to 3^{rd} power of frequency. Therefore, if CPU frequency increases, thermal dissipation will also increase.

Hunting the Power Hungry Instructions

For design of our covert channel, we settle with AVX instruction set that is supported by newer Intel CPUs.

To decide which instructions draw highest power, use instructions one by one in a loop, run it, and see how hot CPU becomes. This is the approach I took.

Hunting the Power Hungry Instructions

We settle with following instructions:

- VFMADD132PD (Multiply packed double-precision floating-point values from ymmX and ymmZ, add to ymmY and put result in ymmX)
- VBROADCASTSD
- VMULPD/SD
- VSUBPD/SD
- VADDPD/SD

Power De-optimization of Code

Problem statement: generate maximum heat using chosen assembly instructions.

Power De-optimization of Code

Any ideas?

Power De-optimizaton of Code

Pro tips:

- Floating point arithmetic is costlier than integer arithmetic.
- Division is costlier than multiplication, which is costlier than addition/subtraction, which is costlier than logical boolean operations.
- Ratio of costly instructions to cheap instructions should be as high as possible. In other words, DO NOT loop around a single costly instruction.
- Always attempt to run same code in different threads (1 thread per CPU core).

Cooling the CPU Down

Time to head back to school level physics.

Newton's law of cooling: the rate of change of the temperature of an object is proportional to the difference between its own temperature and the ambient temperature (i.e. the temperature of its surroundings).

Cooling the CPU Down

In other words, temperature difference between heat sink and CPU dice should be large enough to cool CPU down quickly.

Which means, we cannot keep CPU hot for long time, as that will start heating contact surface of heat sink, which will decelerate cooling of CPU. Even worse, CPU may throttle rather aggressively if it runs hot for long time. At worst case, if CPU dice reaches peak temperature, system will turn off immediately to prevent meltdown.

Cooling the CPU Down

Cooling mechanism is simple: release all the load at once, and let cooling system handle everything else.

Constraints on Covert Channel

- Should be able to heat CPU within 2-3 seconds by sufficient margin
- CPU should be able to cool down within 3-4 seconds back to old temperature
- Heating and cooling should be reliable with respect to time (i.e. it should take same time to heat to same temperature)
- System load from all other processes running on the system is almost static.

Designing the Hot Loop

To put everything in code, we have three choices:

- Use assembly to write hot loop
- Use inline assembly to write hot loop, mixed with C++ code
- Use intrinsics to write hot loop

We will use Intel intrinsics to write our hot loop, as there is less chance of error (in writing incorrect assembly), and the code does not look ugly.

Tools of Trade

All we need is:

- A linux box (you can do it on Windows/MacOS/BSD etc as well)
- GNU G++ Compiler

Designing the Hot Loop

The Setup

```
register __m256d r0,r1,r2,r3,r4,r5,r6,r7,r8,r9,rA,
                 rB,rC,rD,rE,rF;

r0 = _mm256_set1_pd(XXXX);
r1 = _mm256_set1_pd(XXXX);
.
.
rF = _mm256_set1_pd(XXXX);
```

XXXX is a placeholder for arbitrary double precision floating point values.

Designing the Hot Loop

The hot loop

```
r0 = _mm256_mul_pd(r0,rC);
r1 = _mm256_add_pd(r1,rD);
r2 = _mm256_mul_pd(r2,rE);
r3 = _mm256_sub_pd(r3,rF);
...
r8 = _mm256_mul_pd(r8,rC);
r9 = _mm256_add_pd(r9,rD);
rA = _mm256_mul_pd(rA,rE);
rB = _mm256_sub_pd(rB,rF);
```

The hot loop contains more variations of similar code to ensure that CPU sees AVX instructions one after another to trigger SpeedStep (and possibly TurboBoost) to reach higher and higher frequencies.

Data I/O on Covert Channel

Writing data

- Convert data to binary bitstream
- Add error correction codes as deemed fit
- Encode with any binary phase shift keying based encoding scheme
- For every 1 in encoded bitstream, run the hot loop until CPU heats up.
- For every 0 in encoded bitstream, go to sleep state for some fixed time internal (roughly same as needed by hot loop)

Data I/O on Covert Channel

CPU temperature can be read from the following path under Linux (kernel: 5.3.11-arch1-1):

`/sys/class/hwmon/hwmonX/tempY_input`

X and Y are integers. You have to check exact path on your system to ensure that you pick CPU dice temperature sensors instead of other sensors (GPU, System etc).

Data I/O on Covert Channel

Reading data

- Sample CPU dice temperature in a loop at sufficiently high frequency ($>$ 2x frequency of modulation)
- Demodulate the temperature data to get encoded bitstream
- Decode the bitstream by inverse of encoding scheme used in writing data
- Use error correction codes to determine and recover errors in bitstream
- Recover original bitstream after correcting all errors.
- Convert bitstream to suitable format

DEMO TIME

Test Setup

Hardware

- CPU: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
- Cooling: 3 heat-sinks cooled with two independently controlled fans
- Sensors: 1 per core x 4, 1 per CPU package x 1

Test Setup

Software

- OS: Arch Linux x64 (Kernel: 5.3.11-arch1-1)
- Compiler: GNU GCC 9.2.0 (invocation flags: `-O3 -fopenmp -march=native -mtune=native`)
- CPU Microcode version: 20191113-1

Test Setup

Configuration

- Cores enabled: 4
- Hyper-Threading: Enabled (total cores: 8)
- SpeedStep: Enabled
- TurboBoost: Enabled
- Frequency Scaling Governor: ONDEMAND

Hot Loop

Inner hot loop:

- vfmadd132pd: 18
- vmulpd: 6
- vaddpd: 6

The inner hot loop is run 210 million times per heating cycle.

Hot Loop

Outer hot loop:

- vaddpd: 11
- vbroadcastsd: 9
- vaddsd: 6
- vmulsd: 3

The outer hot loop is run 210 thousand times per heating cycle.

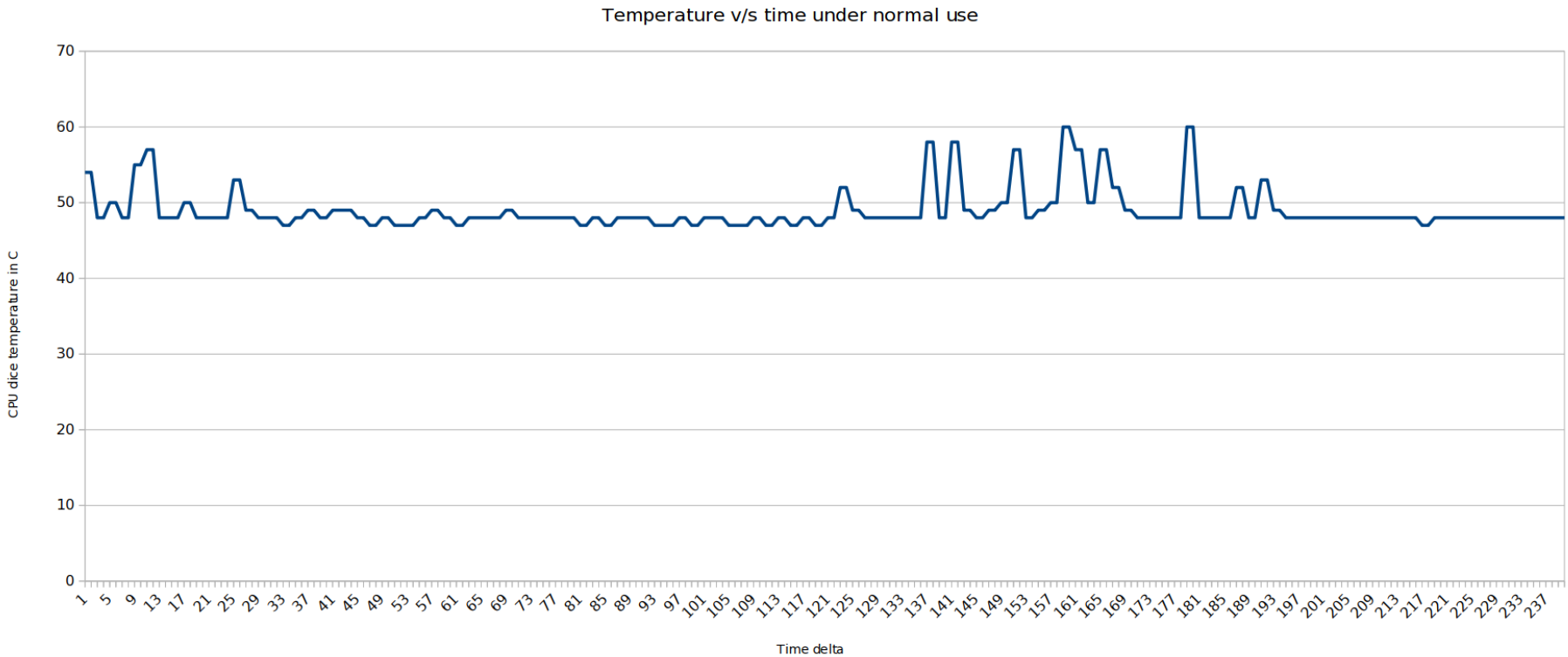
Data

- Timeperiod of 1 heating cycle: 2 seconds (approximated) / 1950ms to 2050ms (as observed)
- Timeperiod of 1 cooling cycle: 2 seconds (approximated) / 2000ms (as programmed)
- Temperature delta in heating cycle: $+25^{o}C$ (peak) / $+20^{o}C$ (average)
- Temperature delta in cooling cycle: $-25^{o}C$ (peak) / $-20^{o}C$ (average)

Countermeasures

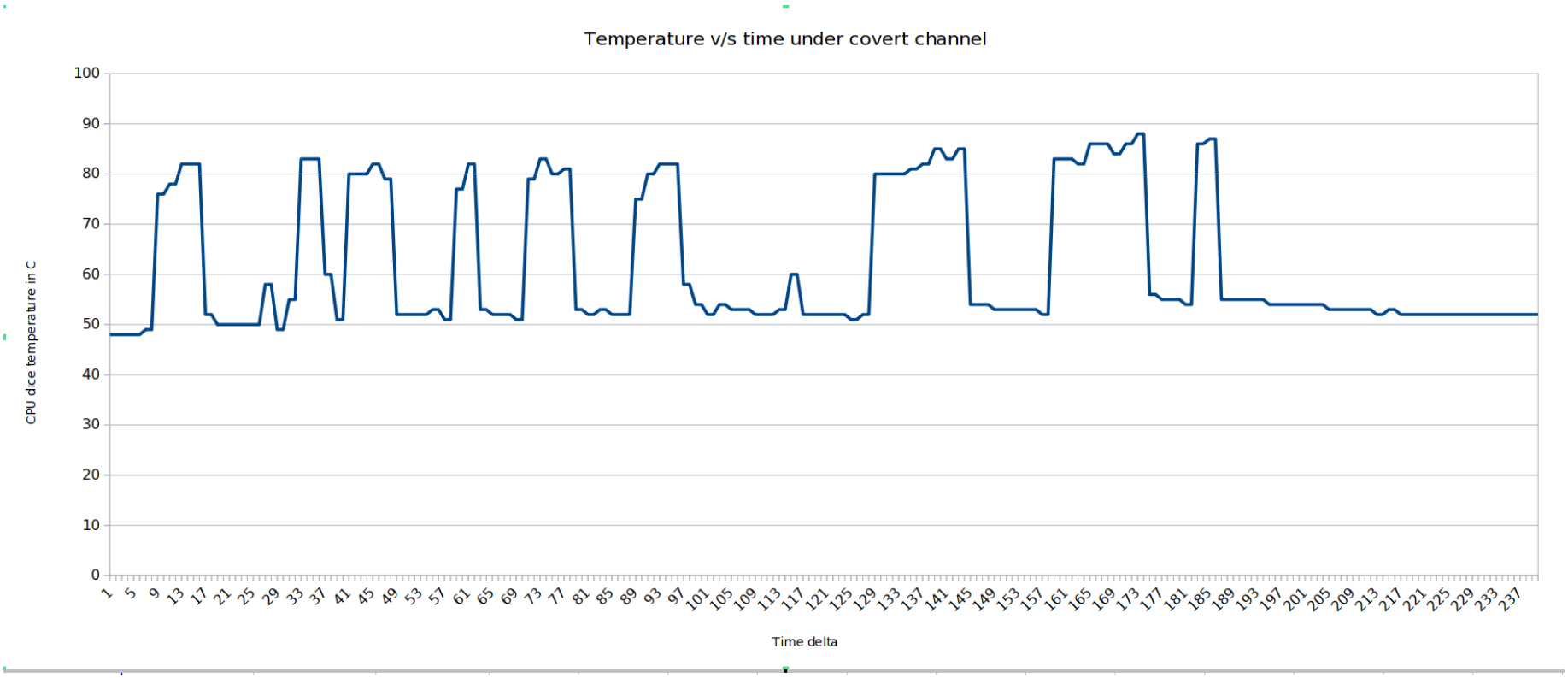
Monitor all the parameters of system for abnormal “patterns”.

Countermeasures



Temperature v/s Time under normal use

Countermeasures



Temperature v/s Time under covert channel

Countermeasures

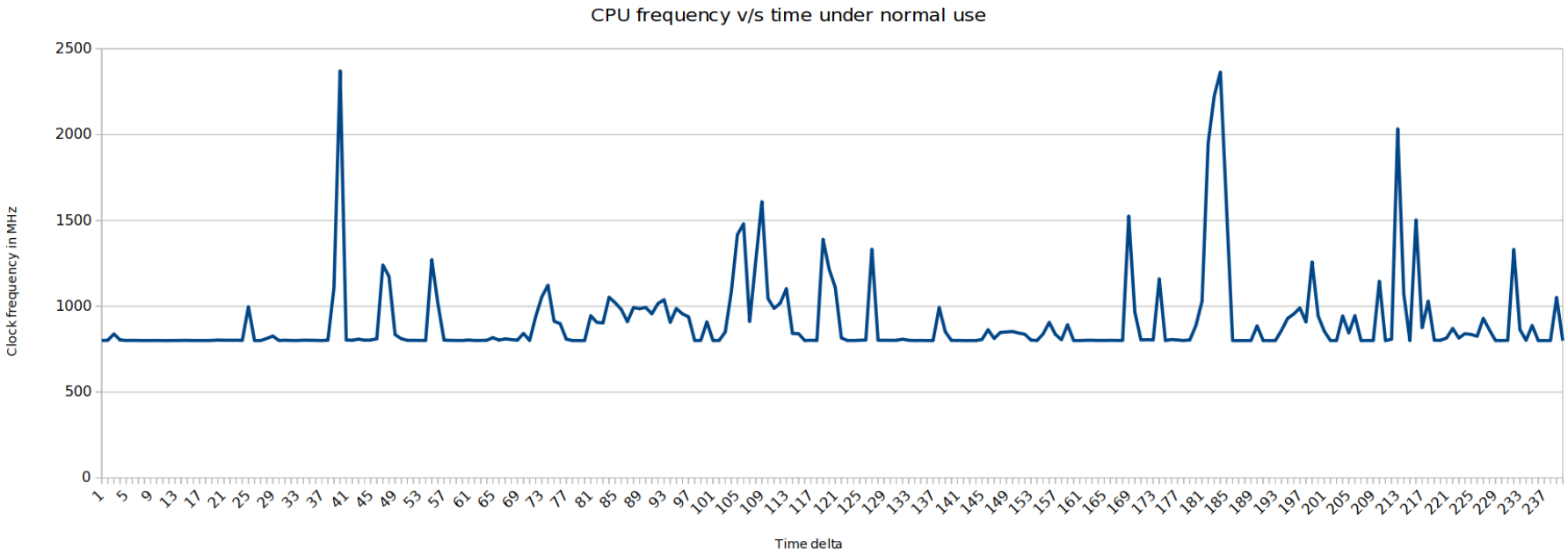
Temperature v/s Time graph

- Under normal use, the graph fluctuates within a narrow range, and has only a few spikes.
- Under covert channel, the graph fluctuates within a wider range, and has many rising and falling edges. Also, instead of spikes, it has flatter peaks.

Countermeasures

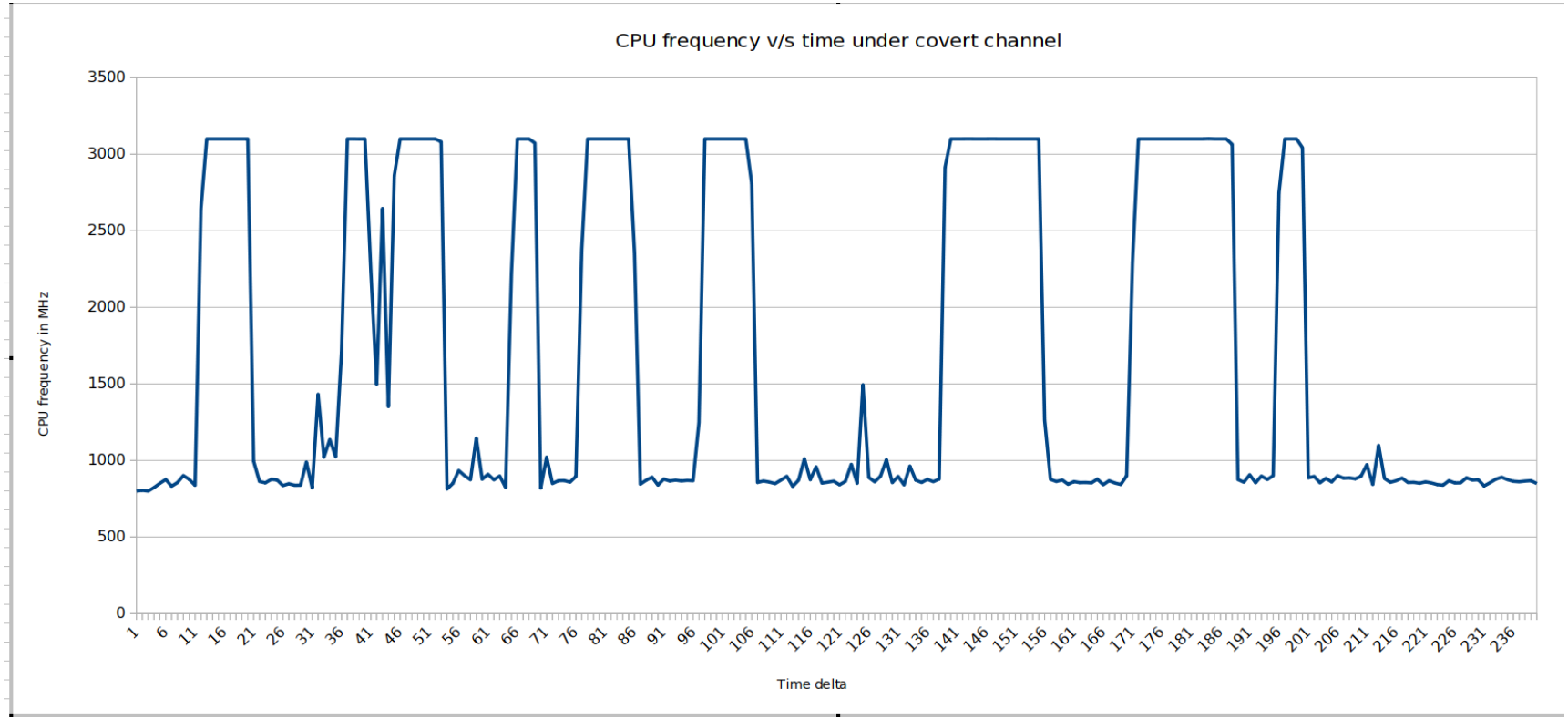
Just like tempearture, CPU clock frequency can also be monitored to reveal presence of something sneaky.

Countermeasures



Frequency v/s Time under normal use

Countermeasures



Frequency v/s Time under covert channel

Countermeasures

Frequency v/s Time graph:

- Graph fluctuates within wide range in both cases.
- Under normal load, there are many spikes in graph as CPU momentarily switches to higher frequencies, and comes down quickly.
- Under normal load, CPU remains below base clock frequency for most (if not all) part.
- Under cover channel, we again see rises and falls, and flat peaks instead of spikes just like temperature v/s time graph.
- Under covert channel, all peaks are at higher than base clock frequency, well within range of TurboBoost.

Countermeasures

Other useful parameters:

- Overall system load will show pattern similar to temperature and frequency. Using this, offending process can often be isolated without much effort.
- Such channels may or may not impact memory utilization and/or fan speed, depending upon how those are constructed. For example, Tempsmitter does not affect fan RPM.

Countermeasures

In general, it takes specialized “witch hunting” mission with careful examination of all such parameters to find a covert channel. These parameters may include electromagnetic intereference, LEDs, acoustic noises, brightness, sound-waves beyond capacity of human ear etc.

THANK YOU