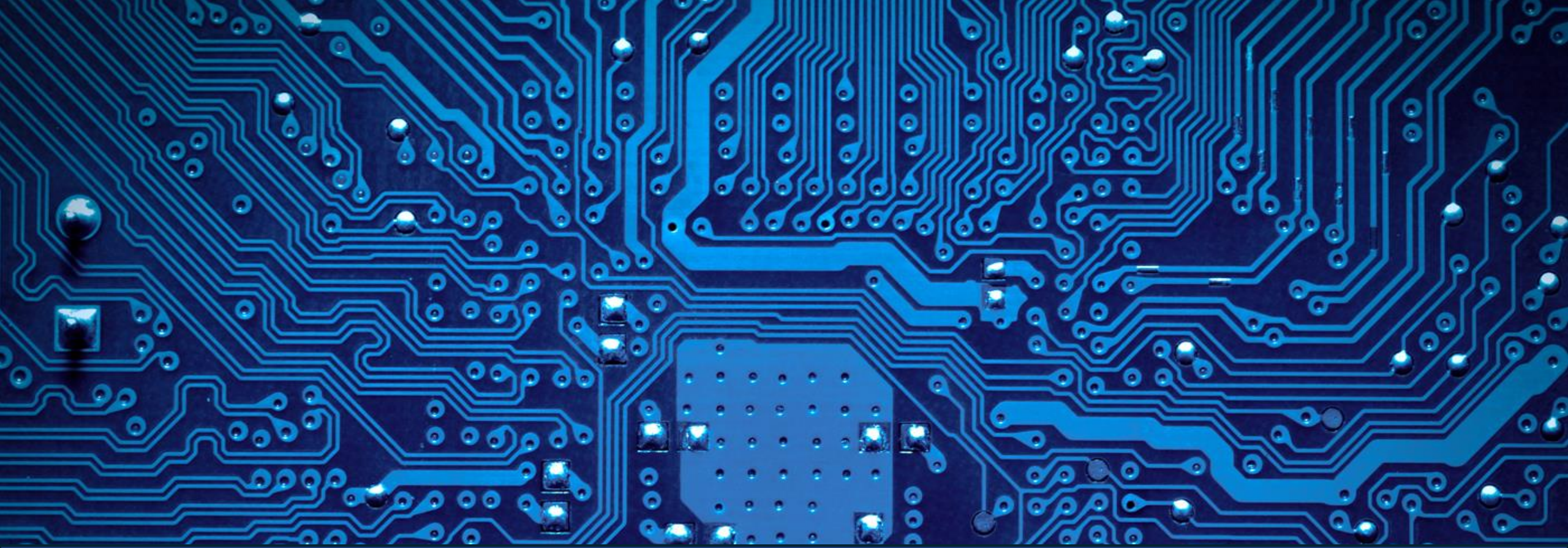


ECE 568: Computer Security

Part 2C: Other Vulnerabilities & Defenses

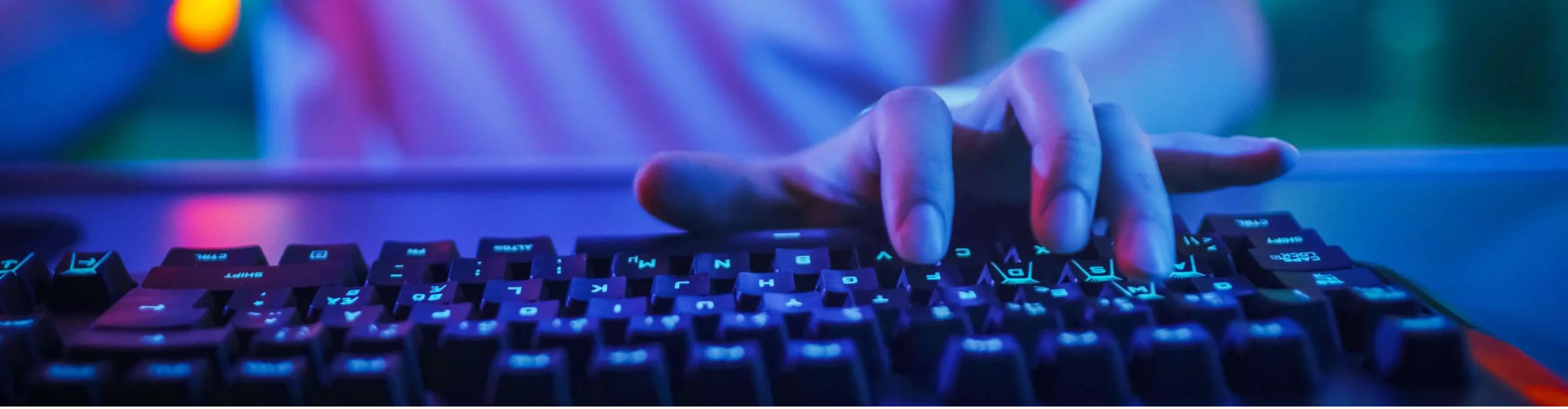
Courtney Gibson, P.Eng.

UNIVERSITY OF TORONTO

Part 2C

Other Vulnerabilities & Defenses

- Other Common Vulnerabilities
- IoT: Cyber-Physical Vulnerabilities
- Defenses



Other Common Vulnerabilities

UNIVERSITY OF TORONTO

Attacks without Code Injection

Until now, the attacks we have examined have involved overwriting the Return Address to point to injected code.

- Is it possible to exploit a program without injecting code?
- What does it mean to exploit a program without injecting code?

Return into **libc**

One exploit method that doesn't require code injection is to use code already present.

Many **libc** functions have code useful to the attacker

- e.g., the **`system`** library call looks a lot like shell code:

```
int system(const char *string);

// system() executes a command specified in string
// by calling /bin/sh -c string
```

Return into **libc**

Rather than inject shellcode, the attacker:

- Changes the return address to point to start of the **`system`** function
- Injects a stack frame onto the stack
- On return, **`system`** executes and expects its arguments at the top of the stack
- Argument contains the string the attacker wants to execute

Attacks without Overwriting the Return Address

Until now, attacks have overwritten return address.

Are there other exploit possibilities?

- **Function pointer overwrite**
- **Global Offset Table overwrite**

Function Pointers

A function pointer is a variable that can be dereferenced to call a function:

```
int foo(int arg1) {
   ...
}

int (*fp)(int arg1);   /* define a function pointer */
fp = &foo;     /* assign addr of foo to the pointer */
fp(6);                 /* call foo via the pointer */
```

An adversary can try to overwrite a function pointer.

Function Pointer Overwrite

- Common in object-oriented languages (*e.g.*, C++)
- Function pointers are often used in C also
  - Allows mimicking polymorphic features of OO languages (*e.g.*, qsort)
  - Used to support dynamically loaded libraries
  - Very common in OS kernels, where the kernel has to run with different modules or drivers without recompilation
  - Also common in other programs that use modules such as web servers, etc.
- Sometimes a buffer will be beside a function pointer rather a return address
  - By overwriting the function pointer, the attacker can cause execution to be redirected when the program calls the function pointer

Dynamic Linking

Program code needs to call functions such as **`printf`** in dynamic libraries

- These libraries are normally linked into the program at run time, at arbitrary locations, by a dynamic linker
- Typically, both the caller of a library function and the function itself are compiled to be **position independent**
- We need to map the position independent function call to the absolute location of the function's code in the library
  - The dynamic linker performs this mapping
  - It uses two tables: the **Procedure Linkage Table** (PLT) and the **Global Offset Table** (GOT)

Dynamic Linking: PLT/GOT

GOT is a table of pointers to functions:

- Contains the absolute memory location of each of the dynamically-loaded library functions

PLT is a table of code entries:

- One per each library function called by program
  - For example, sprintf@plt
- Somewhat similar to a switch statement
- Each code entry invokes the corresponding function pointer in GOT
  - For example, sprintf@PLT code may invoke “jmp GOT[k]”, where k corresponds to the sprintf function index in GOT

Dynamic Linking

All calls to dynamic libraries jump to PLT:

- The first time the function is called, the runtime linker is invoked to load the library
- The runtime linker updates the GOT entry, based on where the library is loaded
- Further calls invoke the function in the loaded library via the updated GOT entry
- The PLT/GOT contain commonly used library functions such as **`printf`**, **`fopen`**, **`fclose`**, etc..

Dynamic Linking: PLT/GOT

Suppose that an attacker is only able to overwrite a single chosen address location with a chosen value

- Then a good option is to overwrite a GOT function pointer

A binary utility like **`objdump -x`** allows disassembling an executable

- It provides the location of these structures
- PLT/GOT always appear at a **known** location

Return-Oriented Programming (ROP)

ROP is a method of creating “shellcode”, based on carefully-selected sequences of instructions, located at the end of existing functions (called “gadgets”)

```
…
mov   $0xb,%ebx
ret
```

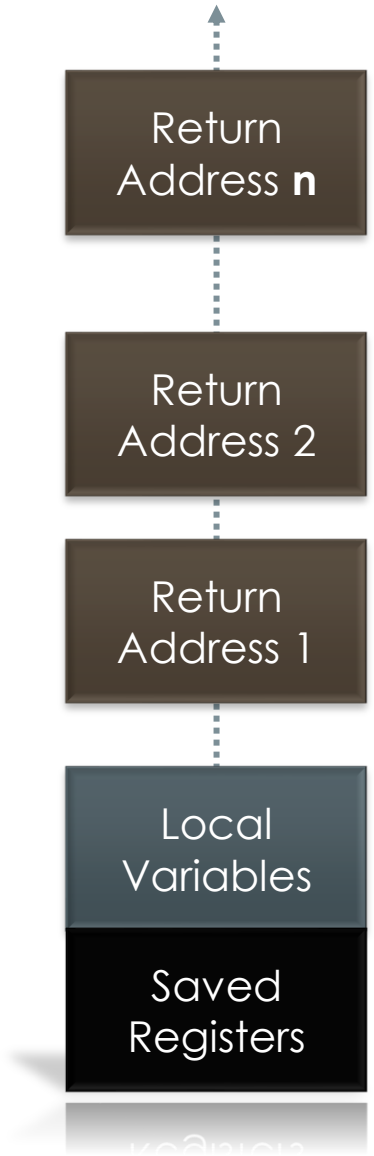
```
…
movl $0x0,%edx
ret
```

```
…
mov $0x0,-8(%ebp)
ret
```

```
…
movl %ebx,%eax
ret
```

Return-Oriented Programming (ROP)

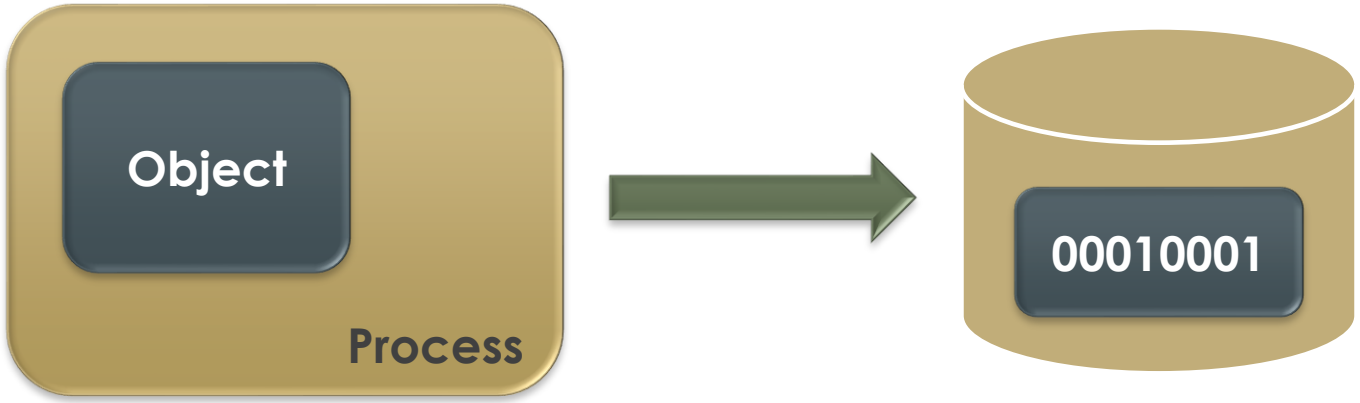
The basic approach is to overflow the stack, placing a sequence of Return Addresses that correspond to the sequence of gadgets



Deserialization Vulnerabilities

Serialization

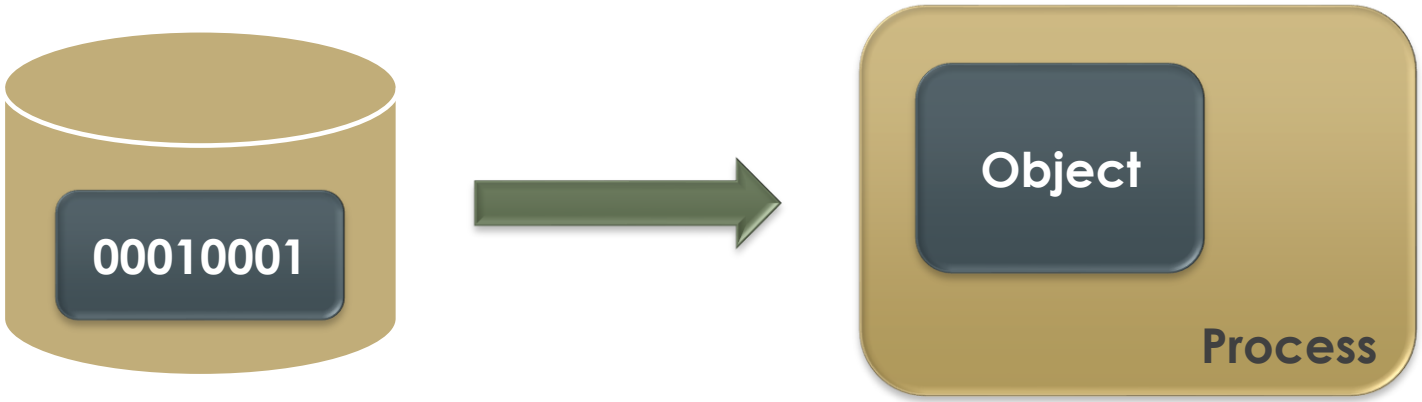
The process of translating objects into a format that can be stored or transmitted over a network.



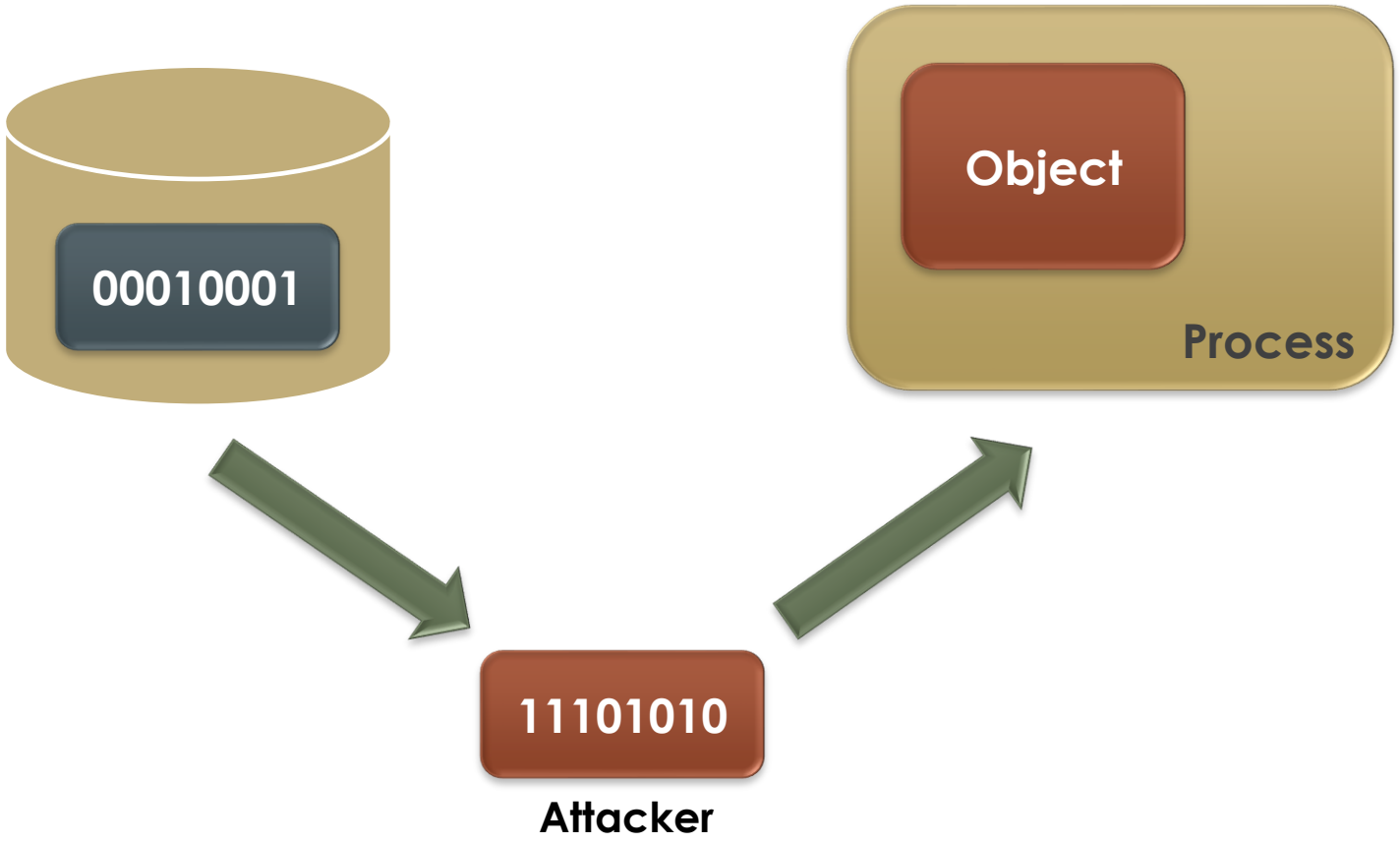
Deserialization Vulnerabilities

De-serialization

- Reconstructs the object in memory
- Should not result in code execution (ideally)
- However, a number of common libraries have vulnerabilities that can be exploited



Deserialization Vulnerabilities



Deserialization Vulnerabilities

Numerous languages and toolkits have had demonstrated exploits and reported CVEs.

- Java, Python, Scala, Websphere, JBOSS, WebLogic…

For more information:

- https://github.com/frohoff/ysoserial.git

Integer Overflows

A server processes packets of variable size:

- The first 2 bytes of the packet store the size of the packet to be processed
- Only packets of size <= 512 bytes should be processed

What's wrong with the code?

Hint: the third arg of memcpy is unsigned int

```
char* processNext(char* strm)
{
  char buf[512];
  short len = *(short*) strm;
  strm += sizeof(len);
  if (len <= 512) {
    memcpy(buf, strm, len);
    process(buf);
    return strm + len;
  } else {
    return -1;
  }
}
```

Argument Overwrite

Instead of changing the execution of a program (i.e. control flow), an attacker can cause unintended data modification

- For example, an attacker can hijack a program by overwriting the argument of a sensitive function such as **`exec`**

```
char buf[128] = “my_program” ;
char vulnerable[32];
        …
exec(buf);
```

The attacker can corrupt the argument **`buf`** by overflowing **`vulnerable`** and have the program execute something else

- Note that the program execution has not changed!

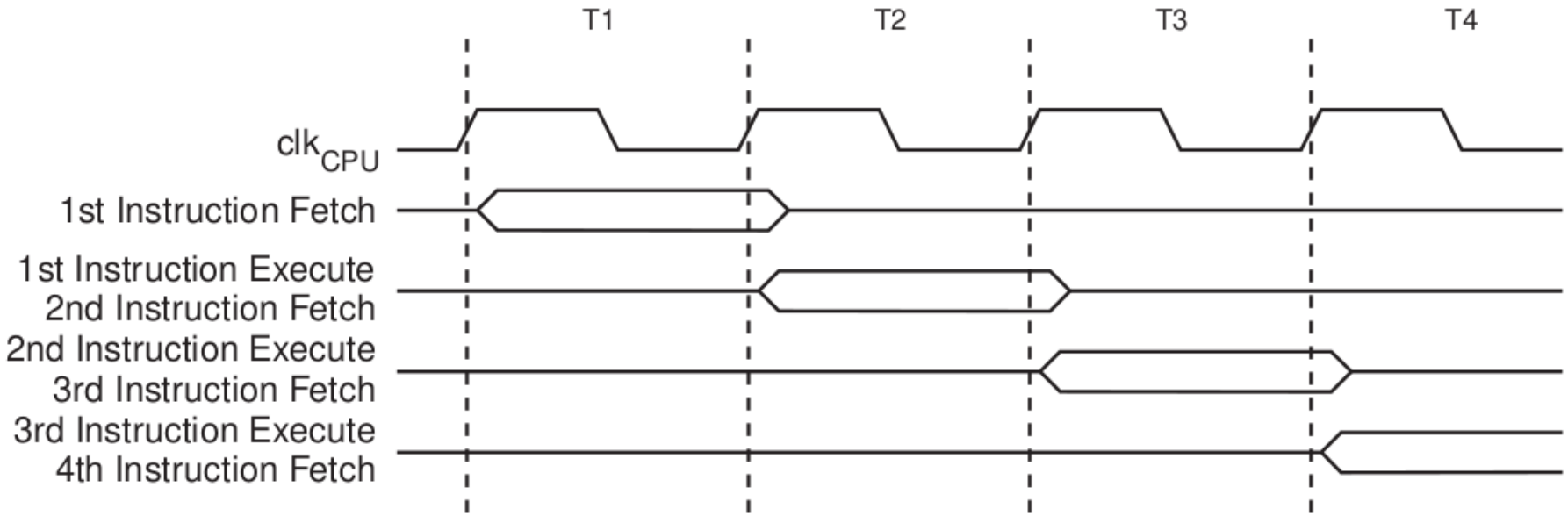
IoT: Cyber-Physical Vulnerabilities

UNIVERSITY OF TORONTO


Welcome Home
PROX
Texecom
6759

Fault Injection Attacks

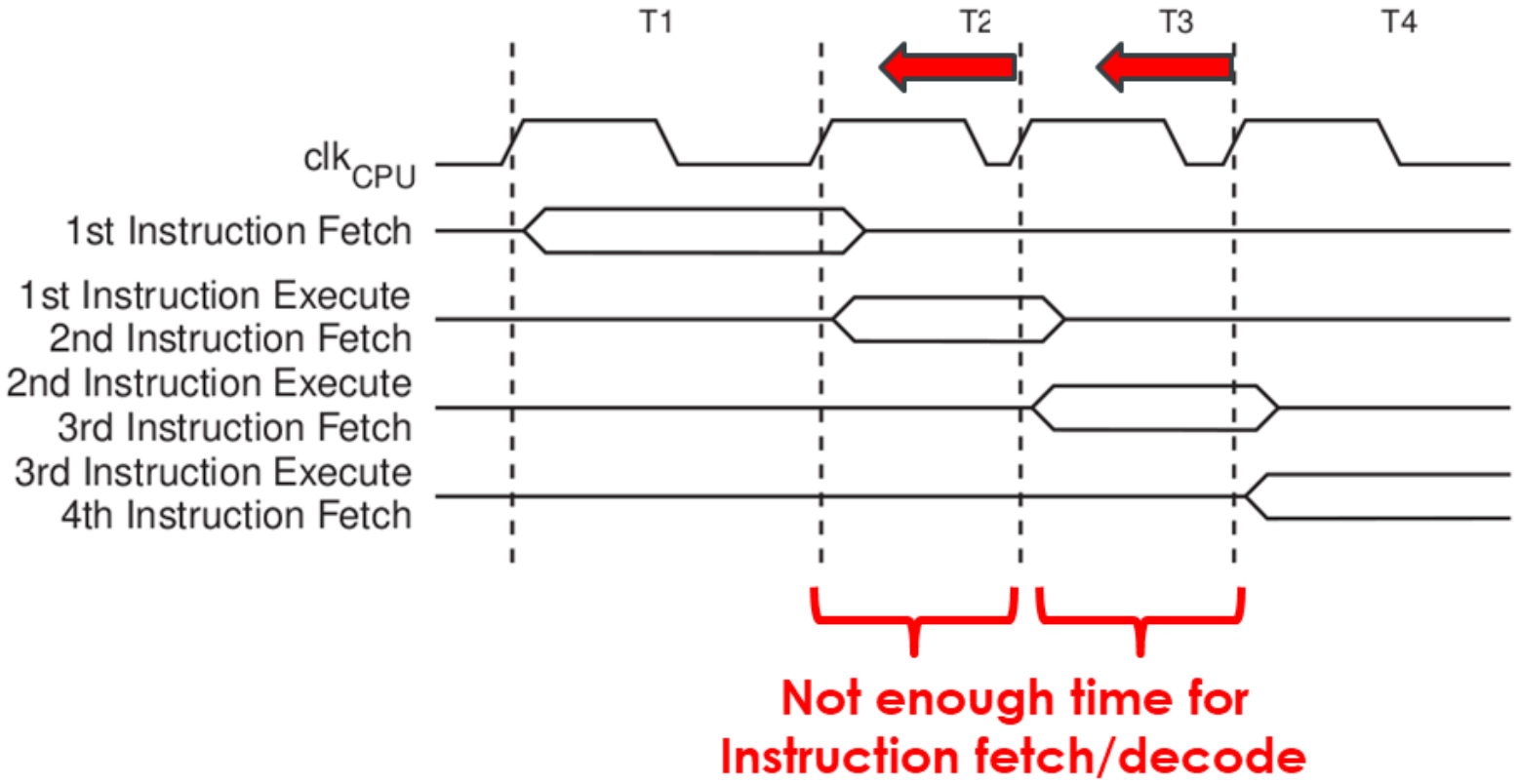
Observation: Proper CPU operation depends on stable power and clock inputs.



ATMega328P Instruction Pipeline

Clock Glitching

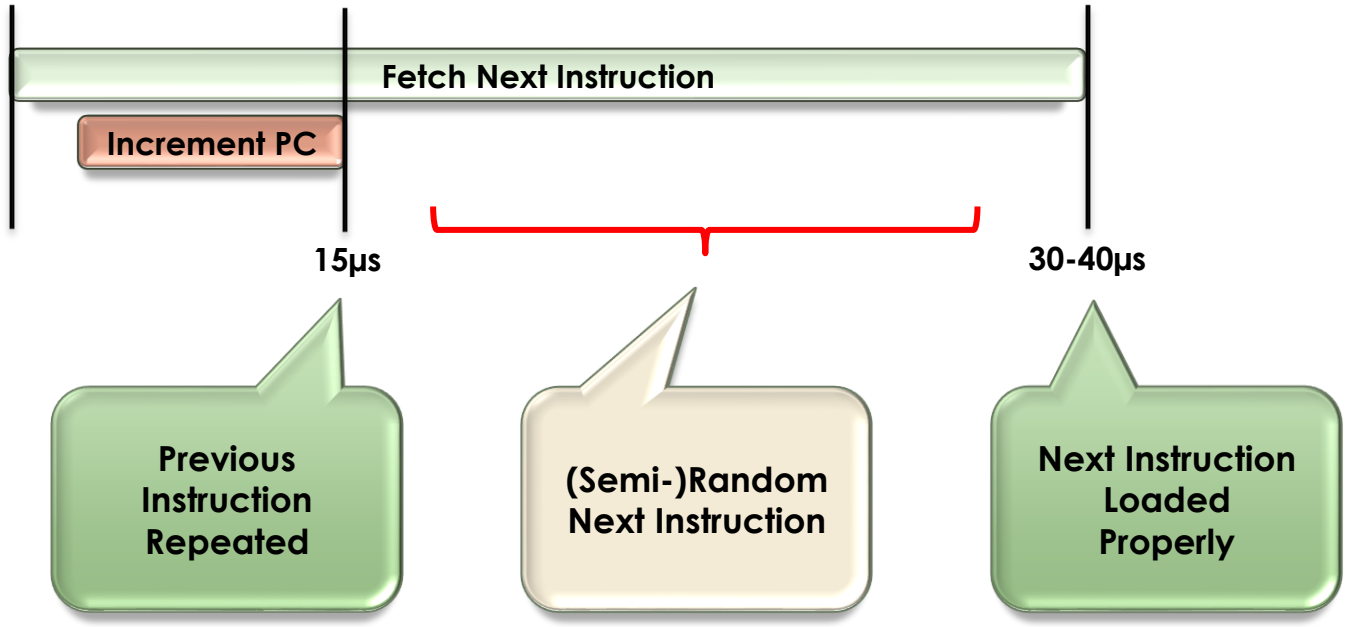
What happens if we introduce a very brief, rapid series of clock pulses? (*e.g.*, 30-40x the normal clock speed?)



If the glitch duration is:

1. **Longer** than the time to increment the Program Counter; and,
2. **Shorter** than the instruction fetch time

then we can start to see a special case: either **instruction skipping** or **instruction corruption**.

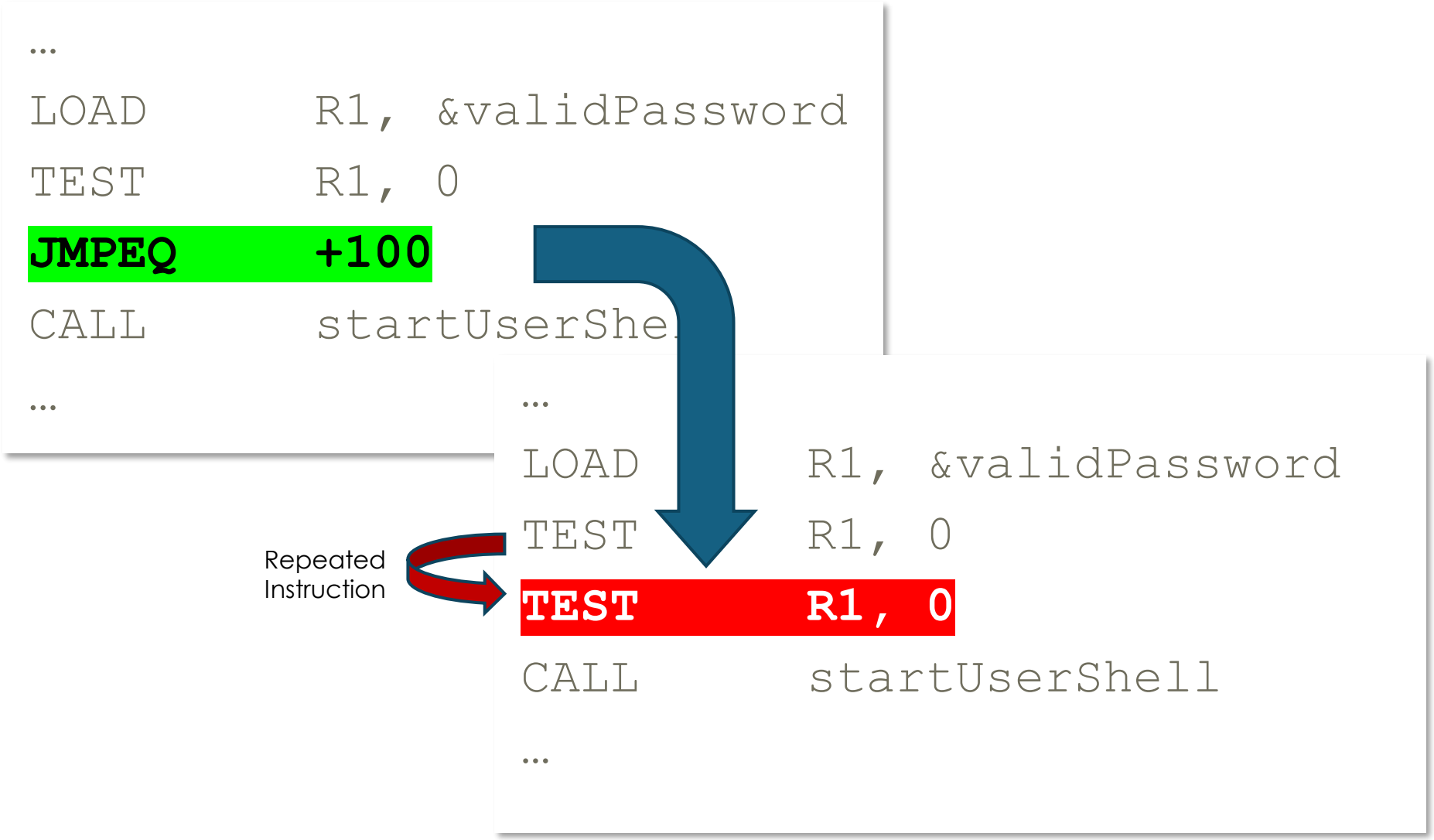


Instruction Skipping

```
…
if ( validPassword == true ) {
      startUserShell();
      …
}
```

```
…
LOAD        R1, &validPassword
TEST        R1, 0
JMPEQ       +0x100
CALL        startUserShell
…
```

Instruction Skipping

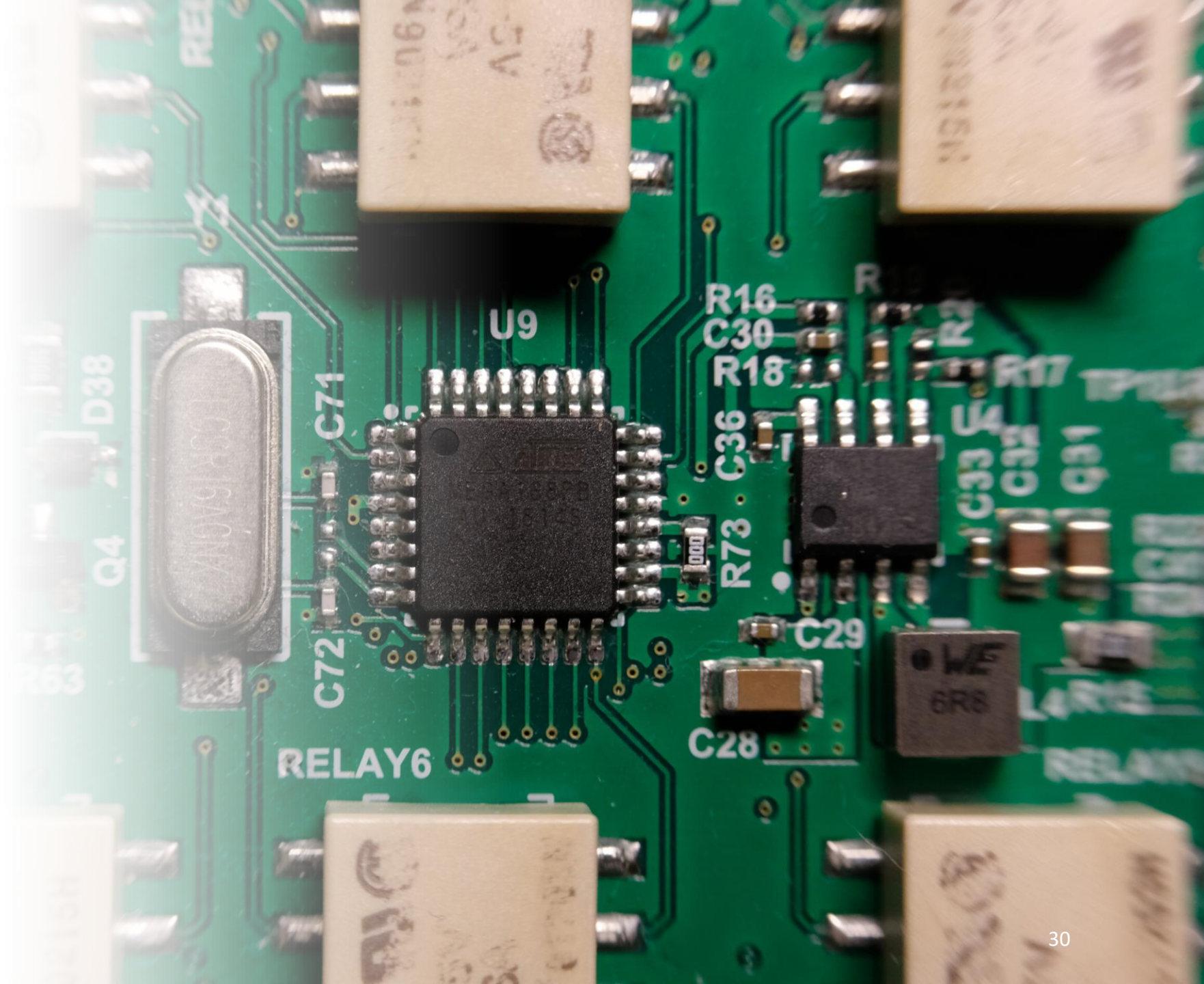


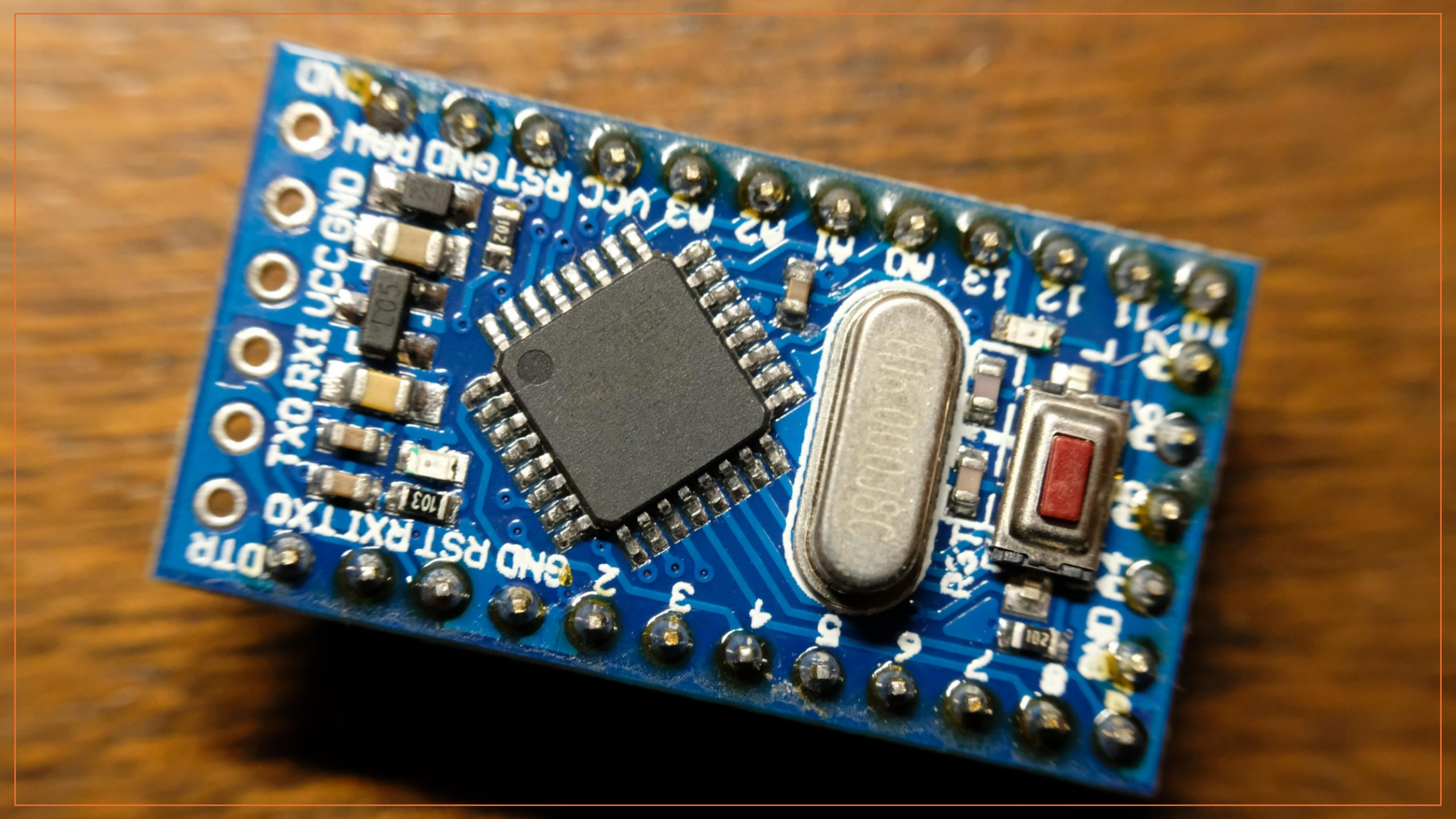
Demo

Clock and Power Glitching

ATMega328P Microcontroller

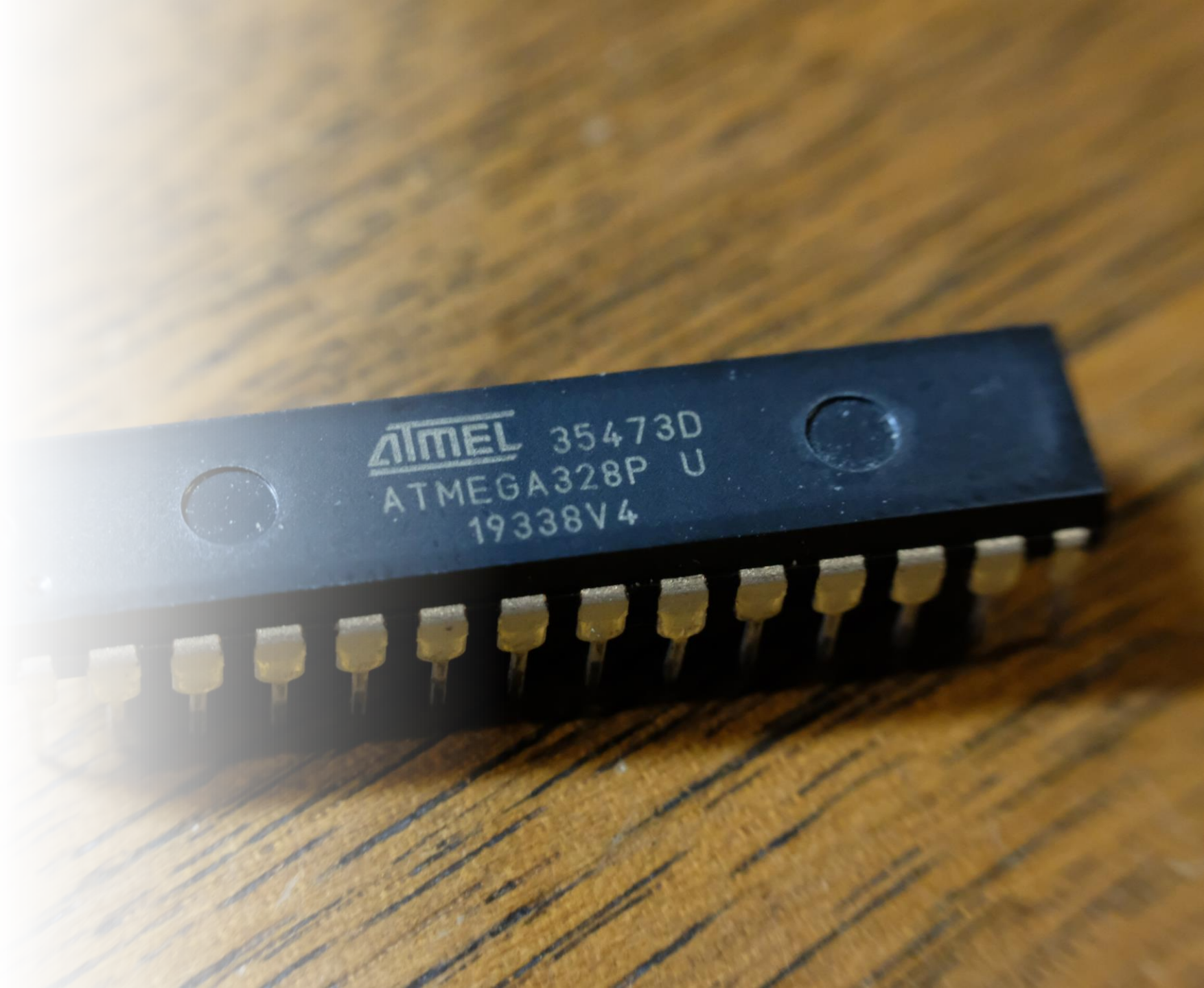
- Low-powered microprocessor + flash memory
- One application, no traditional operating system
- Common in **IoT** (home automation) and **embedded** (industrial control) applications



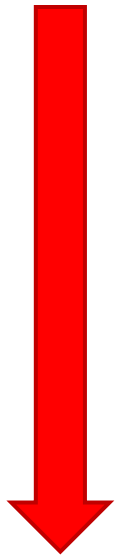

TXD RXI VCC GND
GND RST RXI TXD
DTR
VCC RST GND
103
102

Device Under Test (DUT)

- Our “victim” device will be the 28-pin version of the ATMega328P (easier to solder, but otherwise identical)
- Loaded with a test application that simulates a secure device with a login shell



```
login: root
Password: password

Login incorrect.

login: root
Password: s3cr3tP4ssw0rd&:-)@#

Last login: Wed Dec 31 12:34:56 2025 from 127.0.0.1
root@iotvictim:~#
```

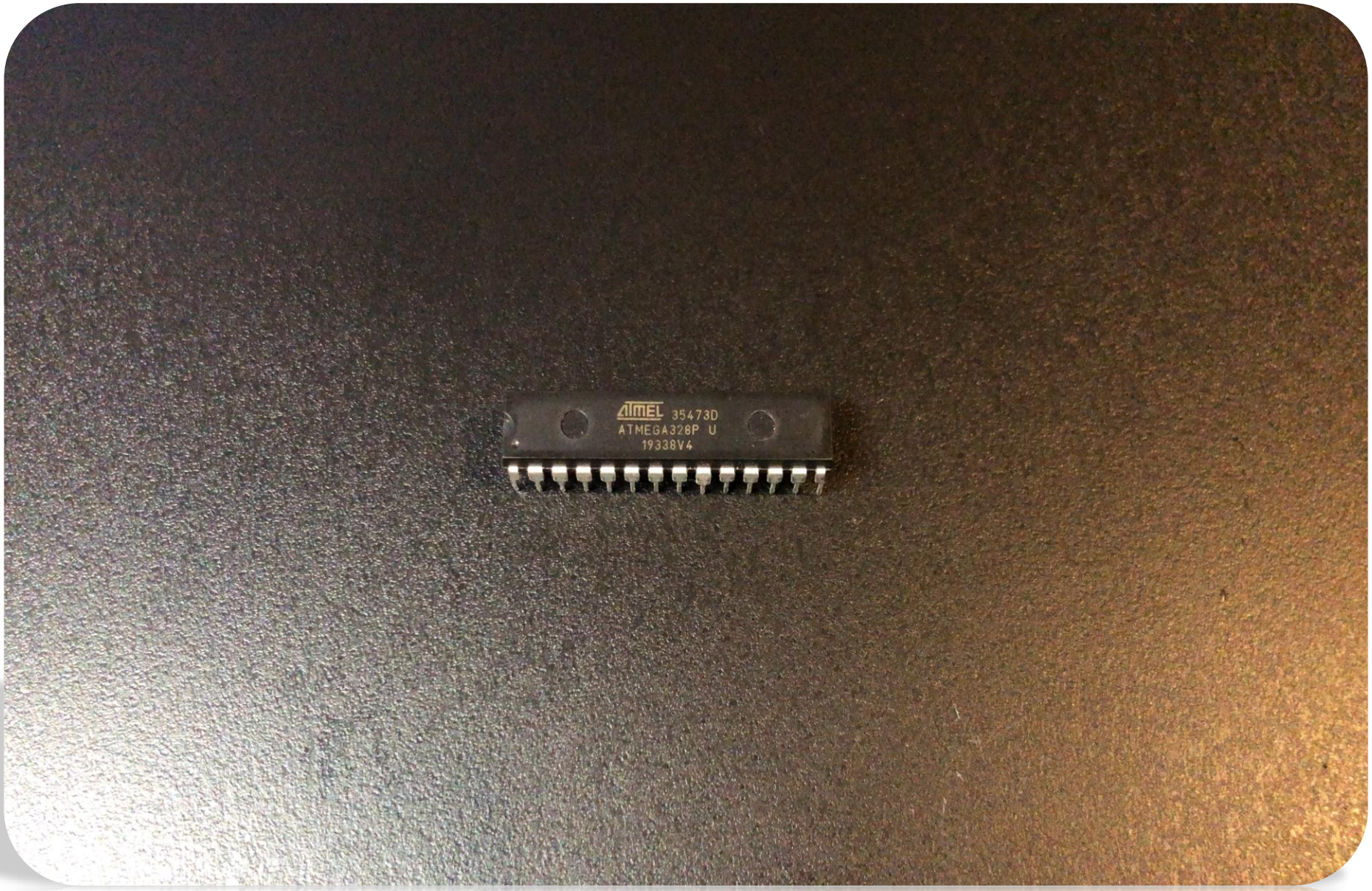


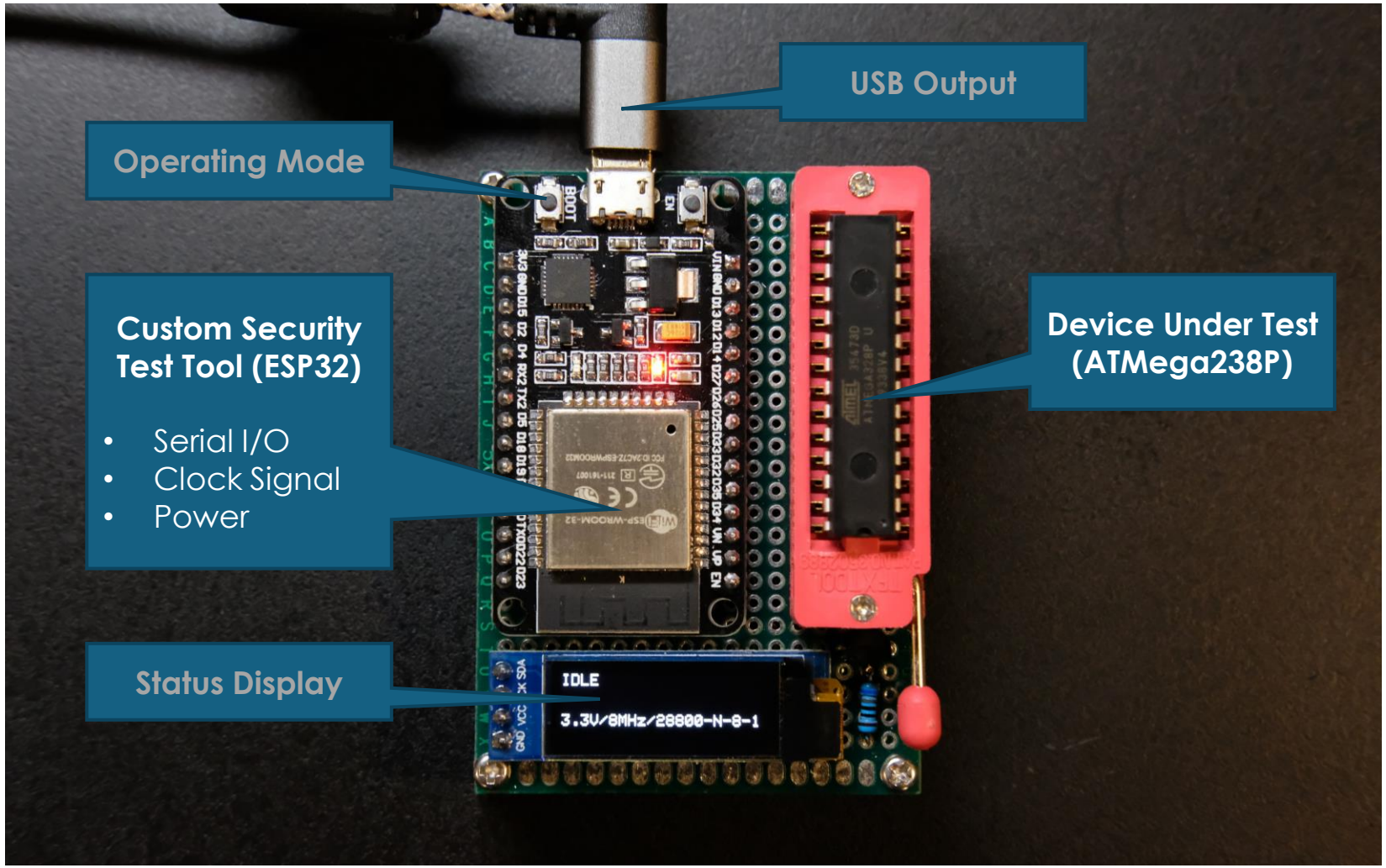
```
…
bool passwordIsValid =
     !strcmp("s3cr3tP4ssw0rd&:-)@#", buffer);

if ( !usernameIsValid || !passwordIsValid ) {

     // Login incorrect
     Serial.println("\n\nLogin incorrect.");
     return;
}

// Login correct
…
```

```
…
bool passwordIsValid =
     !strcmp("s3cr3tP4ssw0rd&:-)@#", buffer);

if ( !usernameIsValid || !passwordIsValid ) {

     // Login incorrect
     Serial.println("\n\nLogin incorrect.");
     return;
}

// Login correct
…
```

GOAL: SKIP THIS!

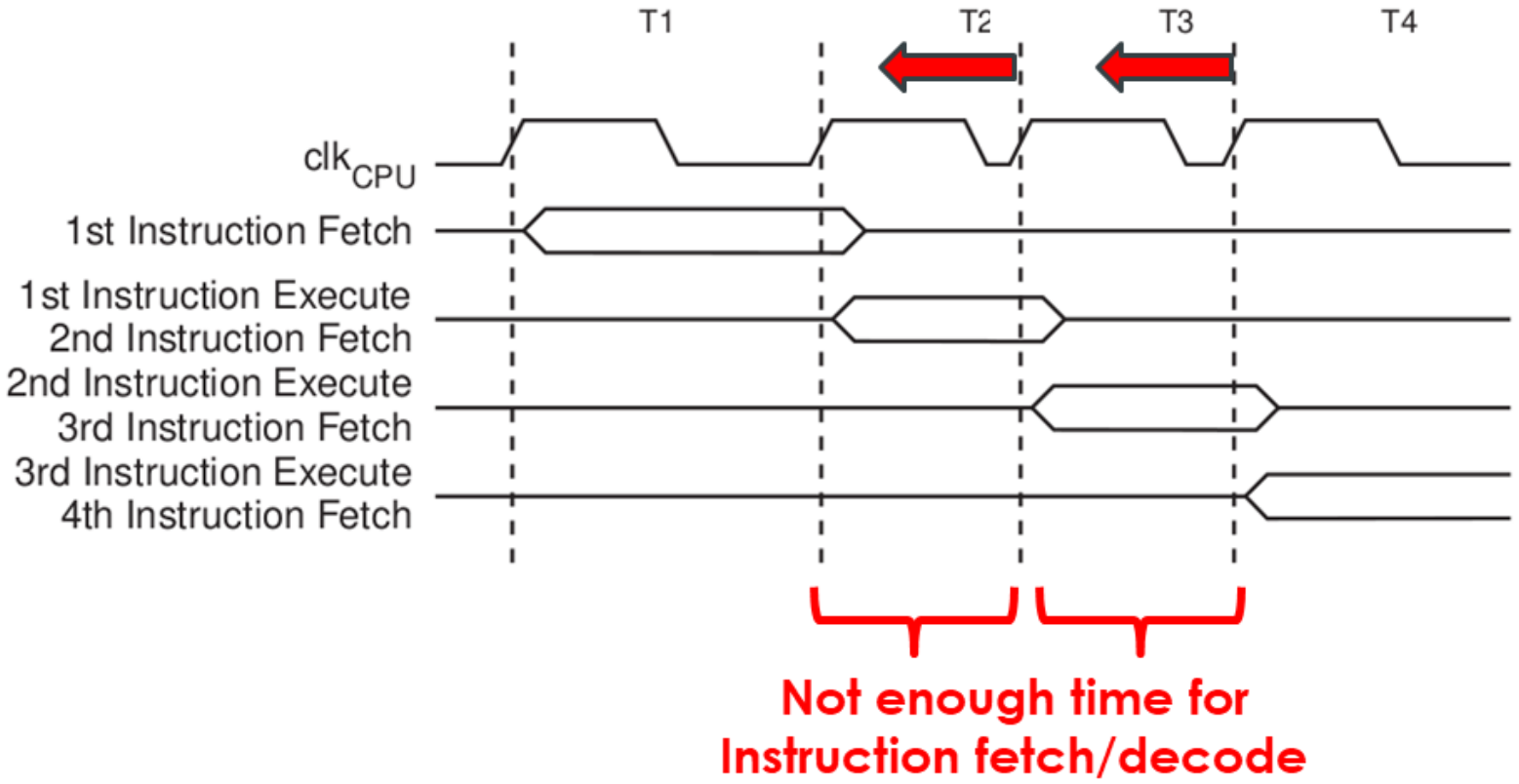
```
login: root
Password:

Login incorrect.

login: root
Password:

Login incorrect.

Last login: Wed Dec 31 12:34:56 2025 from 127.0.0.1
root@iotvictim:~#
```

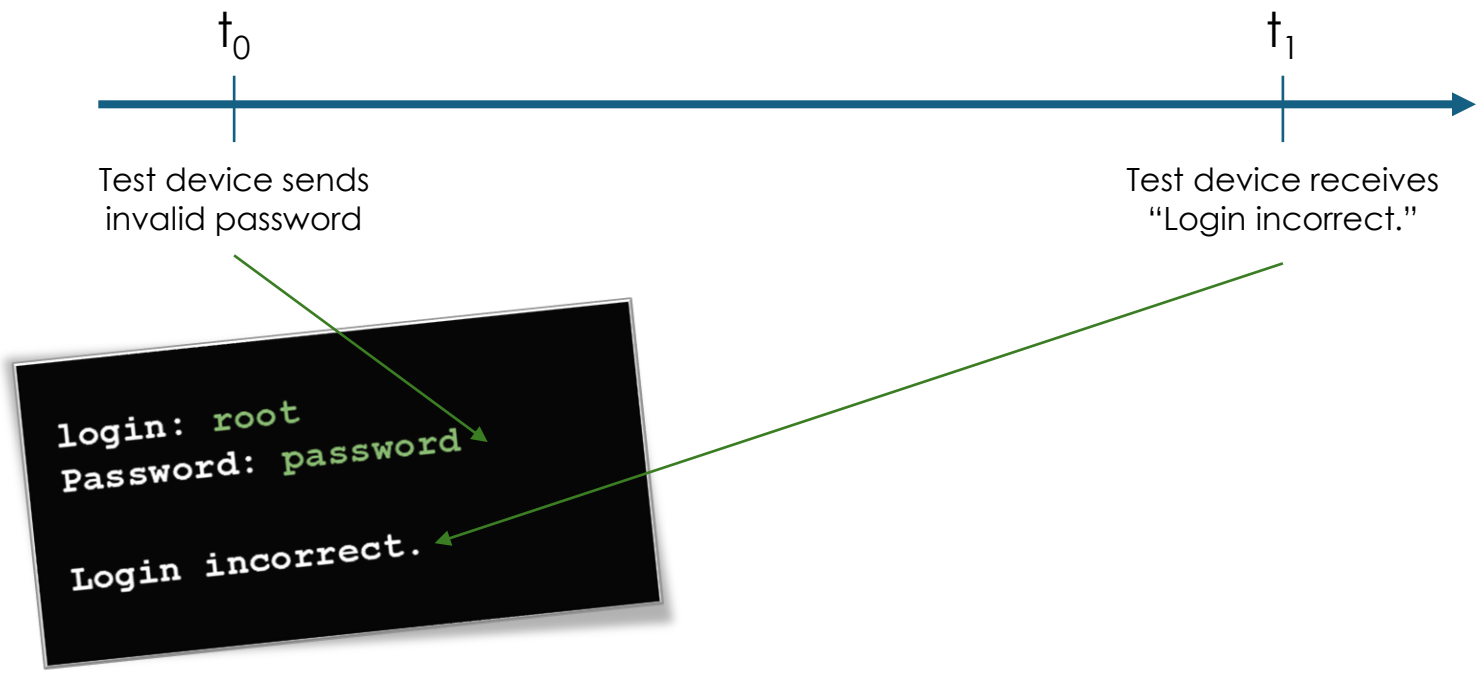

ATMEL 35473D
ATMEGA328P U
1933BV4

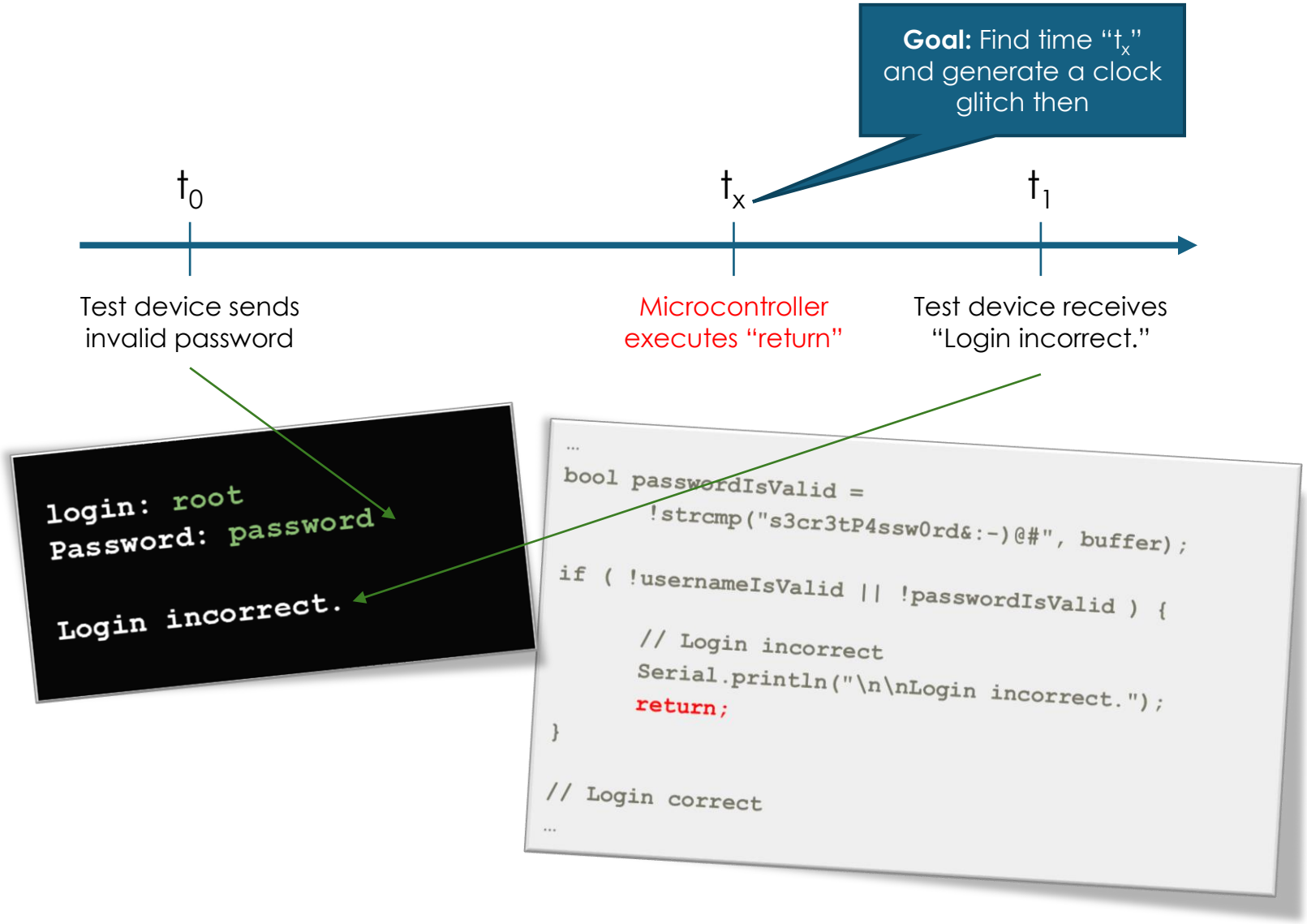

USB Output
Operating Mode
Custom Security Test Tool (ESP32)
• Serial I/O
• Clock Signal
• Power
Device Under Test (ATMega238P)
Status Display
IDLE
3.3V/8MHz/28800-N-8-1

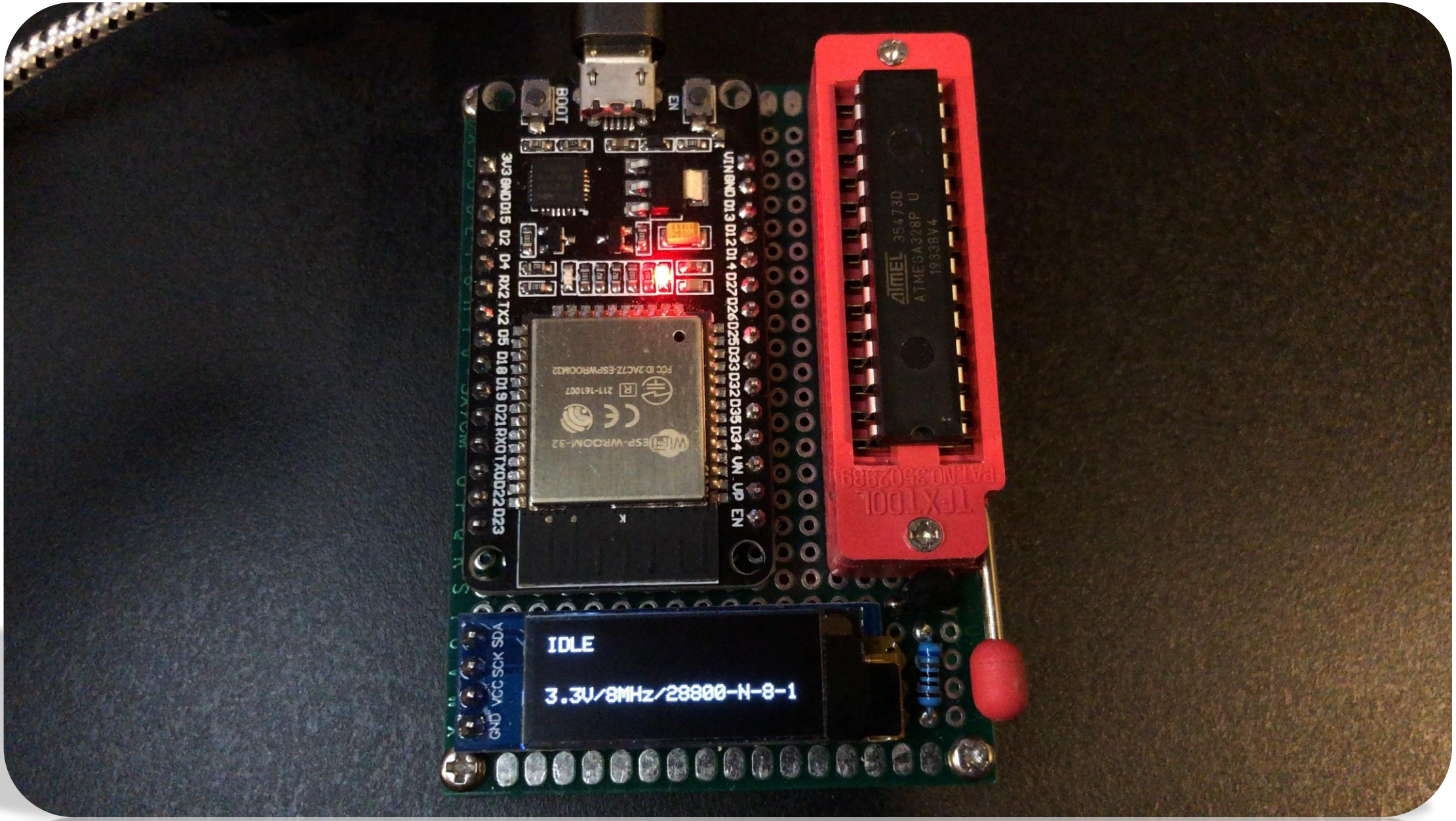
Clock Glitching

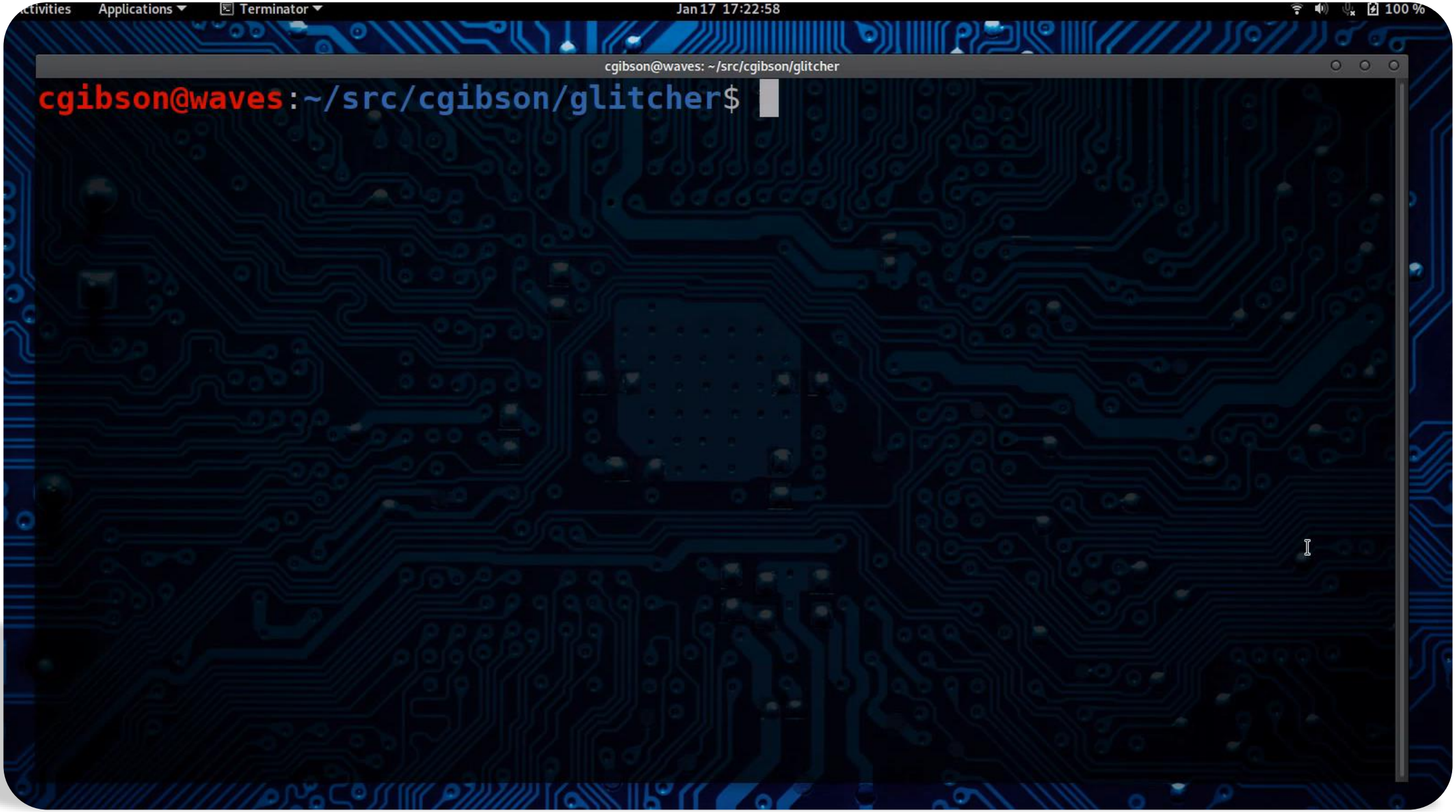
What happens if we introduce a very brief, rapid series of clock pulses? (*e.g.*, 30-40x the normal clock speed?)

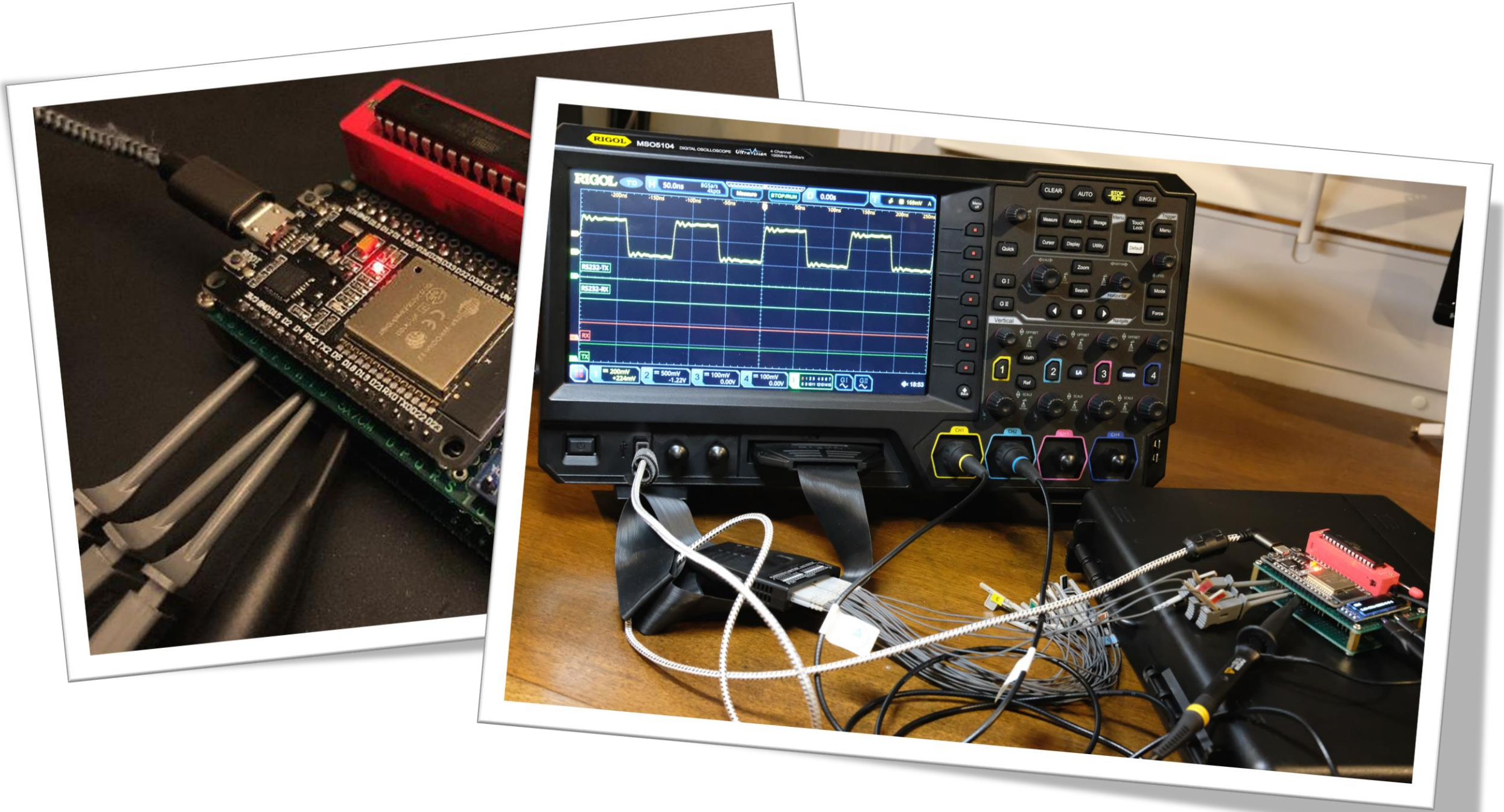



t_0
t_1
Test device sends invalid password
Test device receives "Login incorrect."
login: root
Password: password
Login incorrect.

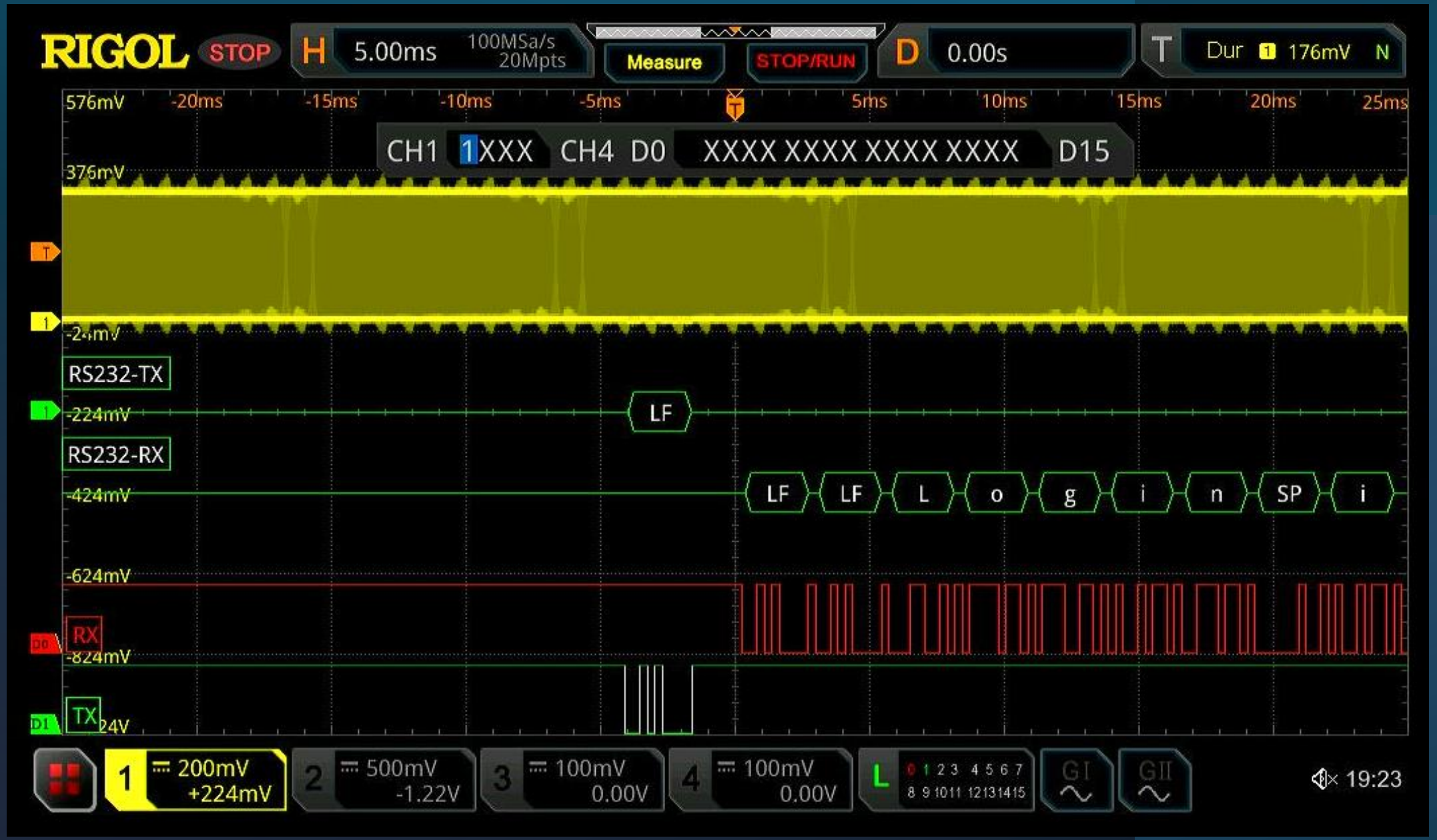

Goal: Find time "t_x" and generate a clock glitch then
t_0
t_x
t_1
Test device sends invalid password
Microcontroller executes "return"
Test device receives "Login incorrect."
login: root
Password: password
Login incorrect.
…
bool passwordIsValid =
!strcmp("s3cr3tP4ssw0rd&:-)@#", buffer);
if (!usernameIsValid || !passwordIsValid) {
// Login incorrect
Serial.println("\n\nLogin incorrect.");
return;
}
// Login correct
…

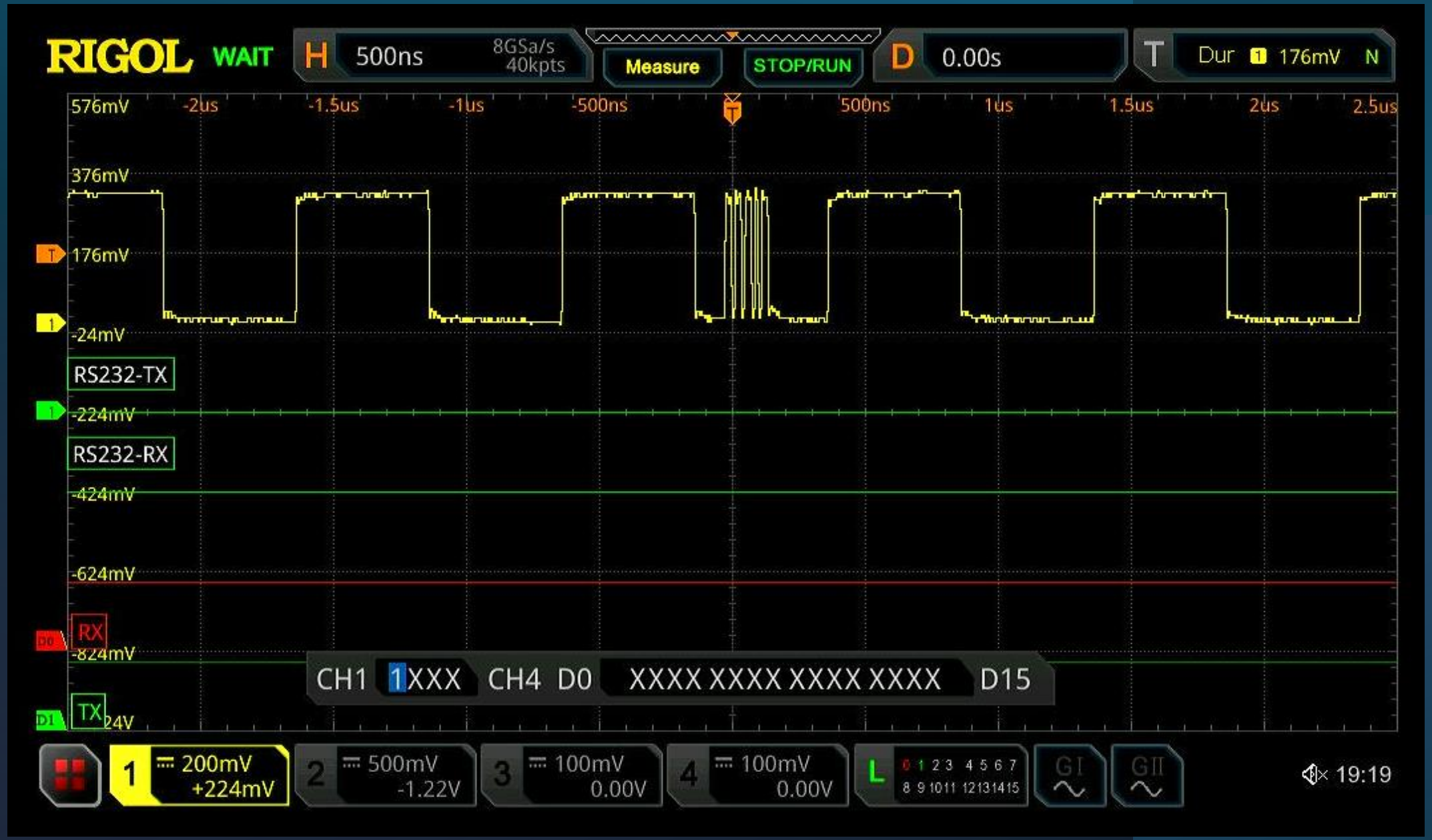

IDLE
3.3V/8MHz/28800-N-8-1
GND VCC SCK SDA
BOOT
EN
ESP-WROOM-32
ATMEGA328P

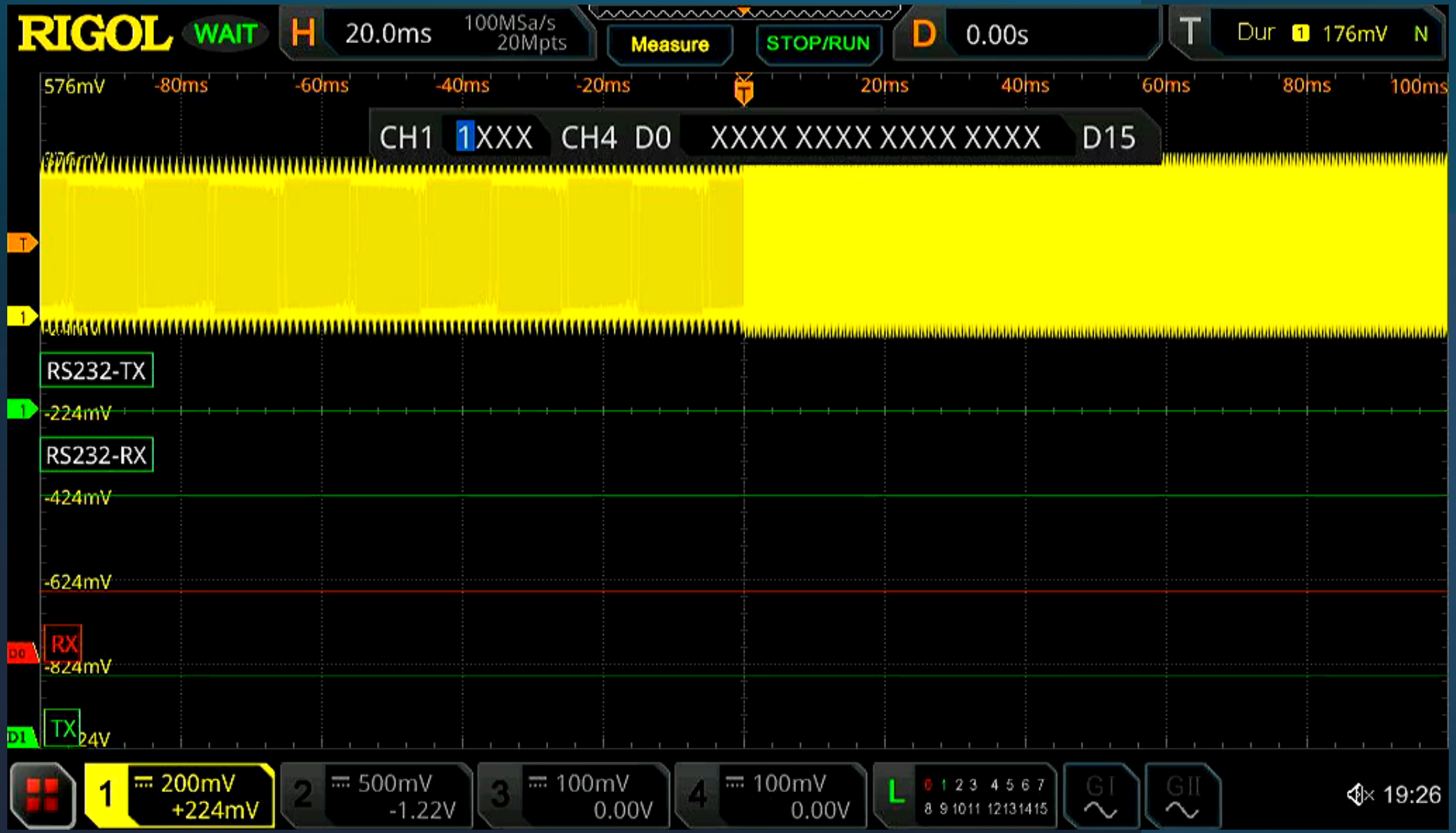

ctivities
Applications
Terminator
Jan 17 17:22:58
100 %
cgibson@waves: ~/src/cgibson/glitcher
cgibson@waves:~/src/cgibson/glitcher$

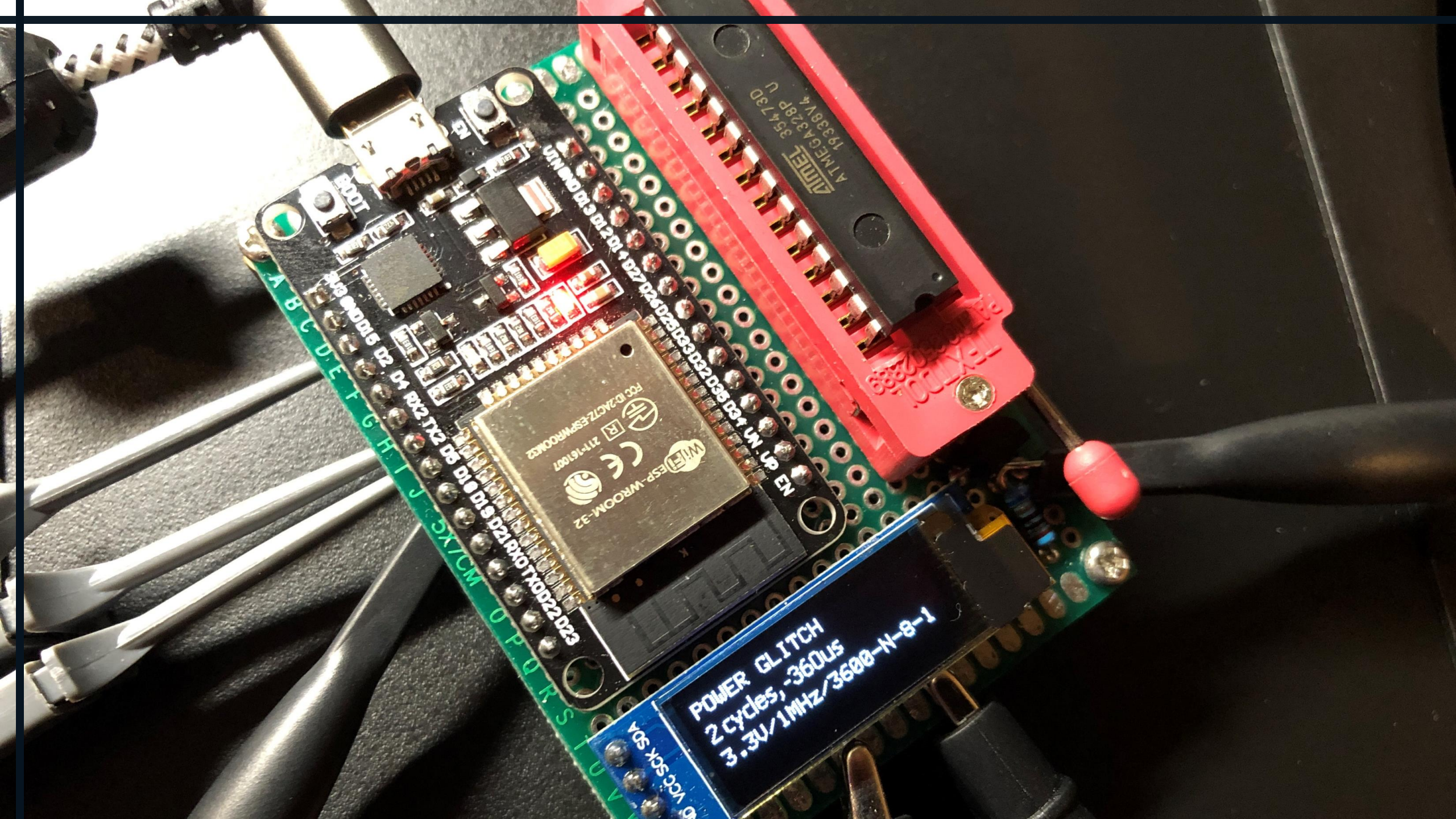

RIGOL
MSO5104
DIGITAL OSCILLOSCOPE
RS232-TX
RS232-RX
RX
TX
CLEAR
AUTO
SINGLE
Measure
Acquire
Storage
Cursor
Display
Utility
Zoom
Search
Math
Ref
Vertical
Horizontal
Trigger
Mode
Force
Quick
Touch Lock
Default

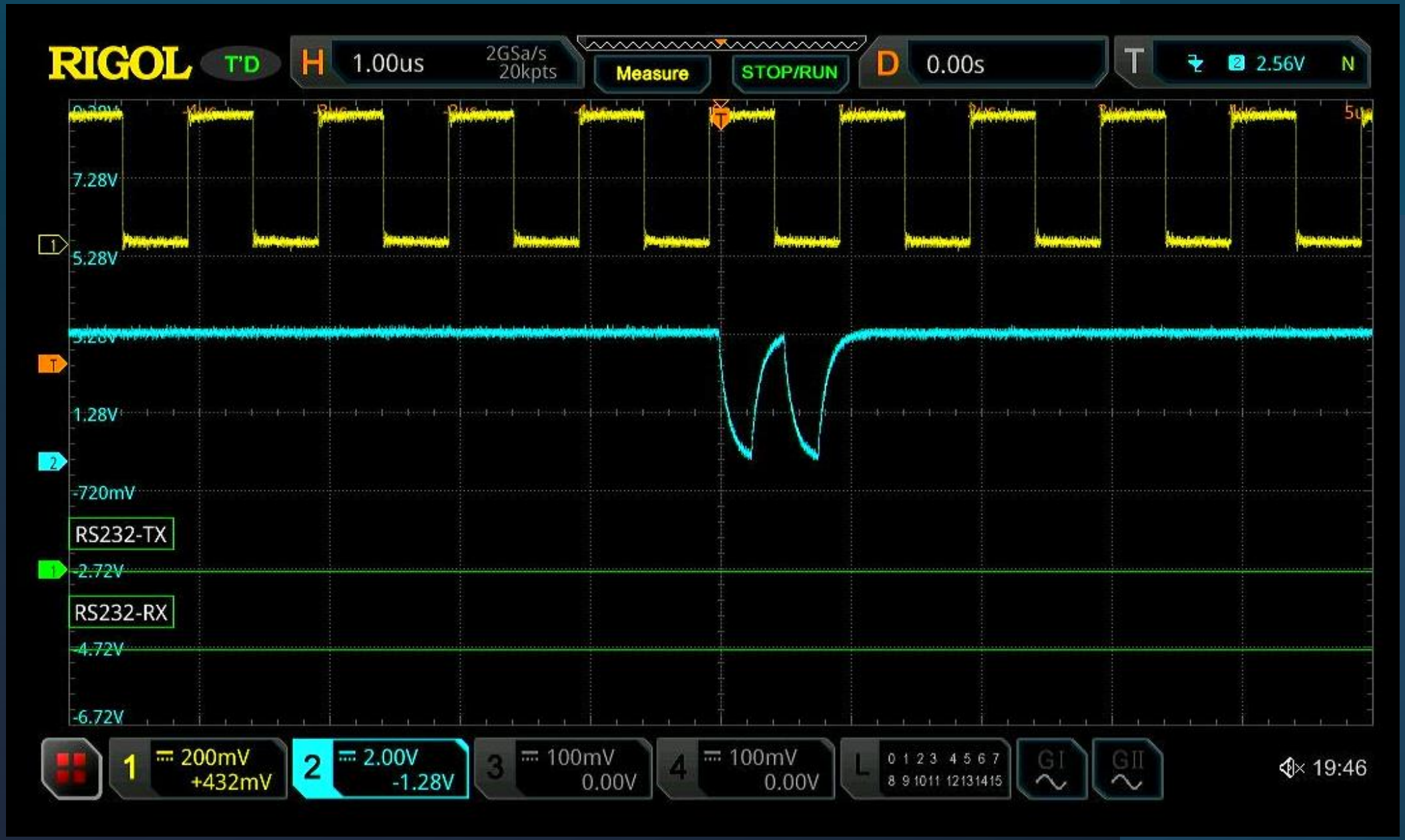

RIGOL
T'D
H
2.00ms
1GSa/s
20Mpts
Measure
STOP/RUN
D
0.00s
T
1
176mV
A
-8ms
-6ms
-4ms
-2ms
2ms
4ms
6ms
8ms
10ms
T
RX
D0
TX
D1
1
200mV
+224mV
2
500mV
-1.22V
3
100mV
0.00V
4
100mV
0.00V
L
0 1 2 3 4 5 6 7
8 9 1011 12131415
GI
GII
19:06

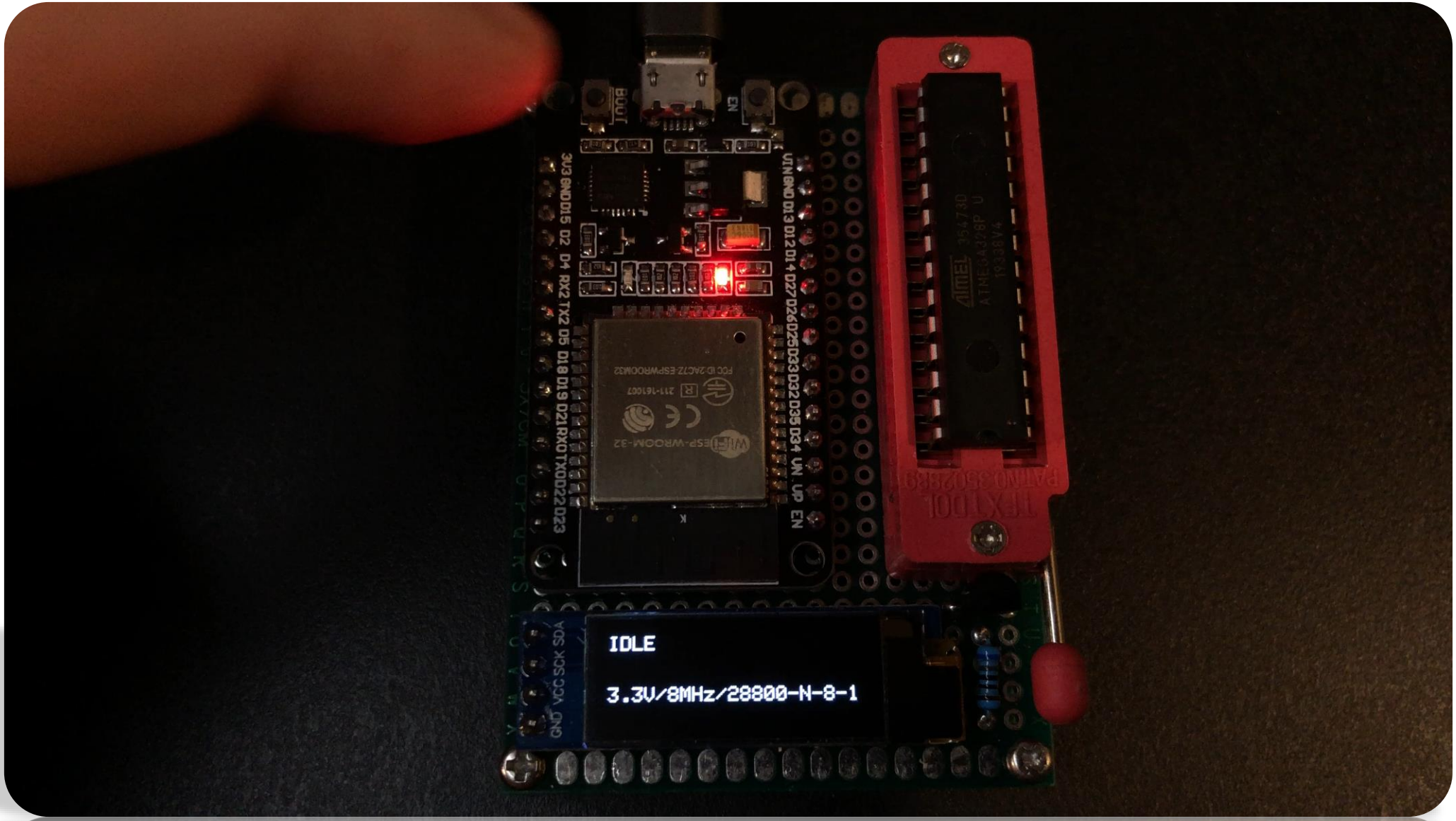

RIGOL
STOP
H 5.00ms 100MSa/s 20Mpts
Measure
STOP/RUN
D 0.00s
T Dur 1 176mV N
576mV -20ms -15ms -10ms -5ms 5ms 10ms 15ms 20ms 25ms
CH1 1XXX CH4 D0 XXXX XXXX XXXX XXXX D15
376mV
-24mV
RS232-TX
-224mV
LF
RS232-RX
-424mV
LF LF L o g i n SP i
-624mV
RX
-824mV
TX
1 200mV +224mV
2 500mV -1.22V
3 100mV 0.00V
4 100mV 0.00V
L
GI
GII
19:23

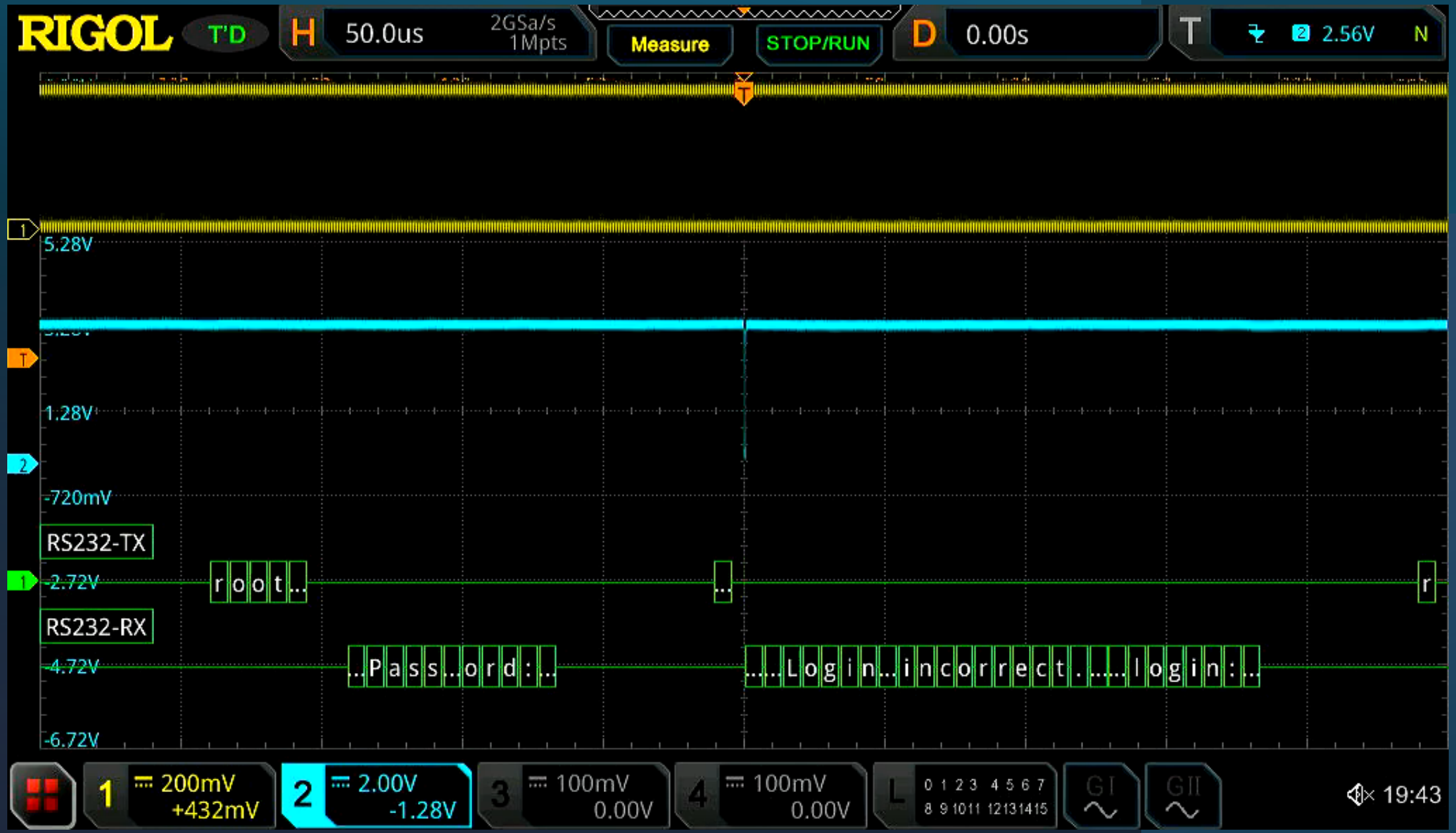

RIGOL
WAIT
H 500ns 8GSa/s 40kpts
Measure
STOP/RUN
D 0.00s
T Dur 1 176mV N
576mV
-2us
-1.5us
-1us
-500ns
500ns
1us
1.5us
2us
2.5us
376mV
176mV
-24mV
RS232-TX
-224mV
RS232-RX
-424mV
-624mV
RX
-824mV
TX
CH1 1XXX CH4 D0 XXXX XXXX XXXX XXXX D15
1 200mV +224mV
2 500mV -1.22V
3 100mV 0.00V
4 100mV 0.00V
19:19

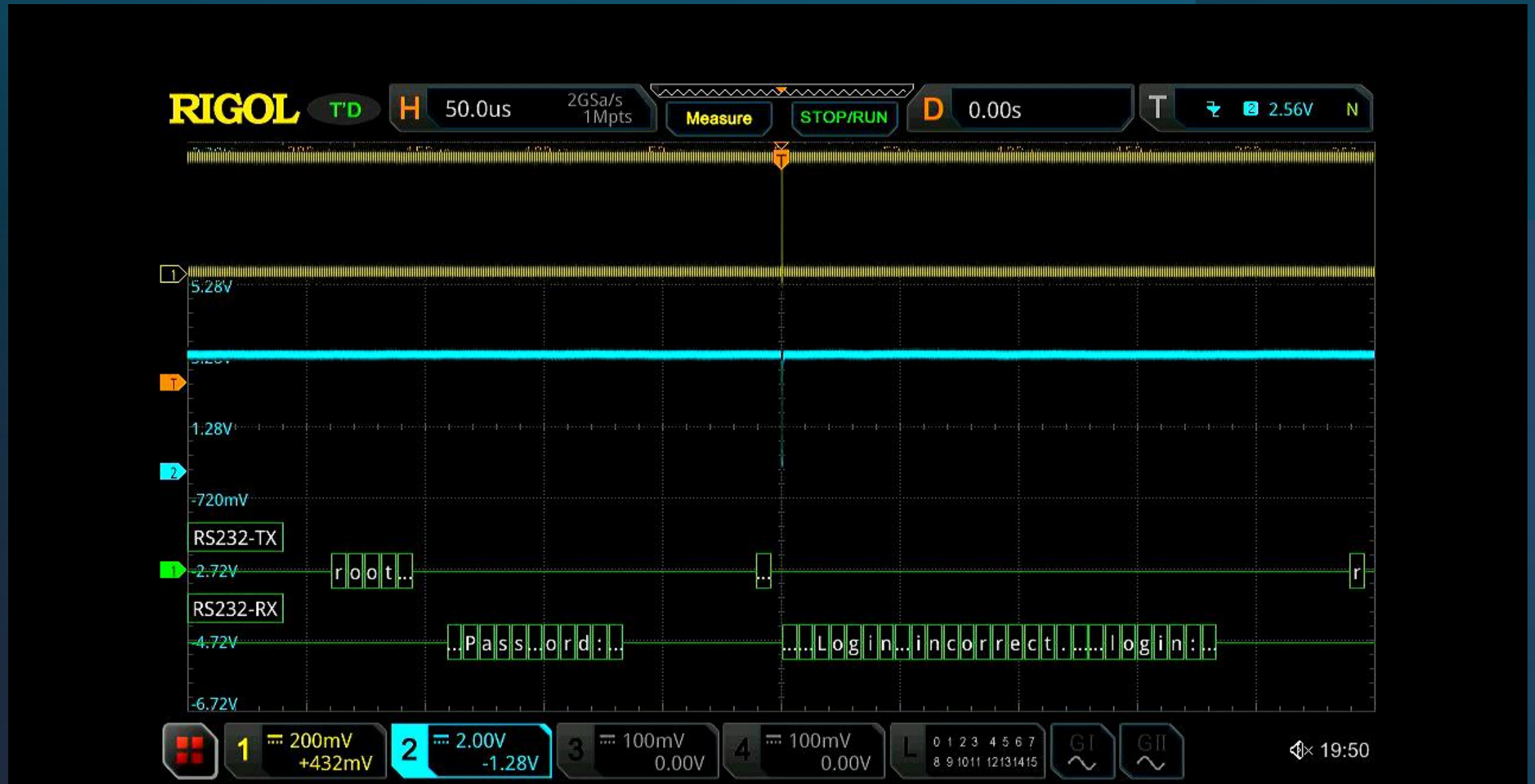

RIGOL
WAIT
H 20.0ms 100MSa/s 20Mpts
Measure
STOP/RUN
D 0.00s
T Dur 1 176mV N
576mV -80ms -60ms -40ms -20ms 20ms 40ms 60ms 80ms 100ms
CH1 1XXX CH4 D0 XXXX XXXX XXXX XXXX D15
RS232-TX
-224mV
RS232-RX
-424mV
-624mV
RX
TX
1 200mV +224mV
2 500mV -1.22V
3 100mV 0.00V
4 100mV 0.00V
L
GI
GII
19:26


POWER GLITCH
2 cycles, -360us
3.3V/1MHz/3600-N-8-1
ESP-WROOM-32
ATMEGA328P
BOOT
EN


RIGOL
T'D
H 1.00us 2GSa/s 20kpts
Measure
STOP/RUN
D 0.00s
T 2.56V N
7.28V
5.28V
1.28V
-720mV
RS232-TX
-2.72V
RS232-RX
-4.72V
-6.72V
1 200mV +432mV
2 2.00V -1.28V
3 100mV 0.00V
4 100mV 0.00V
L 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
19:46


IDLE
3.3V/8MHz/28800-N-8-1

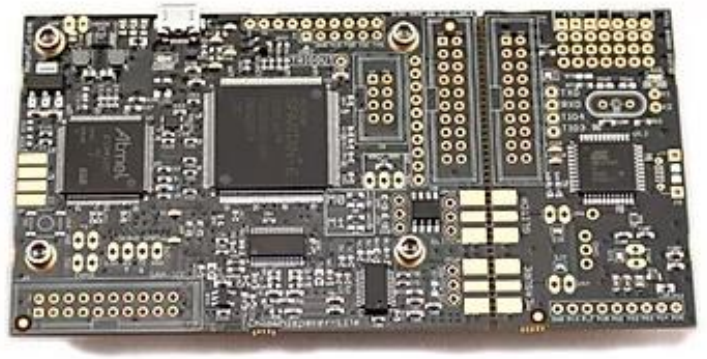

RIGOL
T'D
H 50.0us 2GSa/s 1Mpts
Measure
STOP/RUN
D 0.00s
T 2.56V N
5.28V
1.28V
-720mV
RS232-TX
-2.72V
root..
..
r
RS232-RX
-4.72V
..Password:..
.....Login..incorrect.....login:..
-6.72V
1 200mV +432mV
2 2.00V -1.28V
3 100mV 0.00V
4 100mV 0.00V
L 0 1 2 3 4 5 6 7 8 9 1011 12131415
GI
GII
19:43


RIGOL
T'D
H 50.0us 2GSa/s 1Mpts
Measure
STOP/RUN
D 0.00s
T 2.56V N
5.28V
1.28V
-720mV
RS232-TX
-2.72V
root..
..
r
RS232-RX
-4.72V
..Pass..ord:..
....Login..incorrect.....login:..
-6.72V
1 200mV +432mV
2 2.00V -1.28V
3 100mV 0.00V
4 100mV 0.00V
19:50

```
login: root
Password:

Login incorrect.

login: root
Password:

Login incorrect.

Last login: Wed Dec 31 12:34:56 2025 from 127.0.0.1
root@iotvictim:~#
```

Fault Injection Attacks

Unlooper
($100USD)



Chip Whisperer
($250USD)

Reference: *Secure Application Programming in the Presence of Side Channel Attacks*, Whitteman et al., RSA Conference 2008
http://www.riscure.com/benzine/documents/Paper_Side_Channel_Patterns.pdf

Fault Injection Attacks

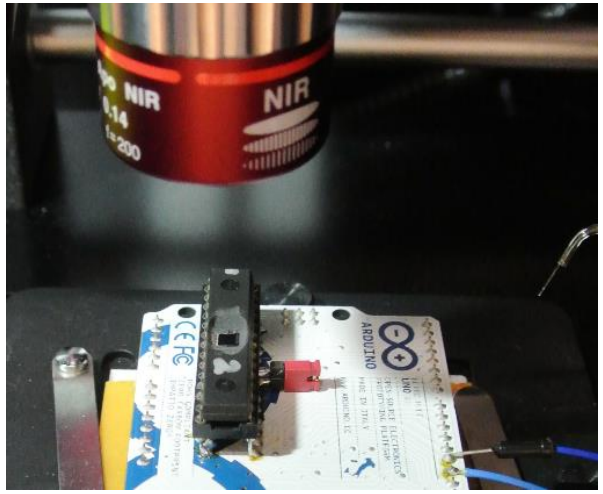
Various researchers are successfully using ML techniques to find the correct duration/timing.

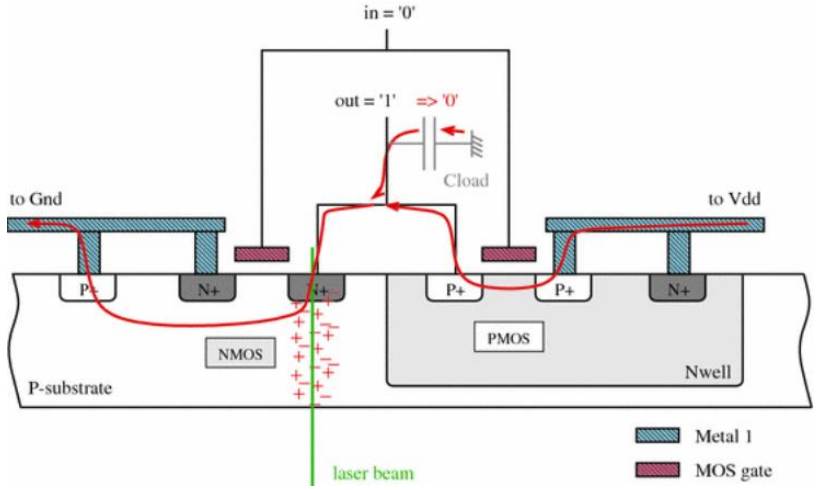
Many other ways to cause controlled circuit malfunctions:

- Laser
- Strobes
- Focussed EM pulses

Fault Injection Attacks

Properly-focussed laser or electromagnetic pulses can flip the output bits of logic gates.





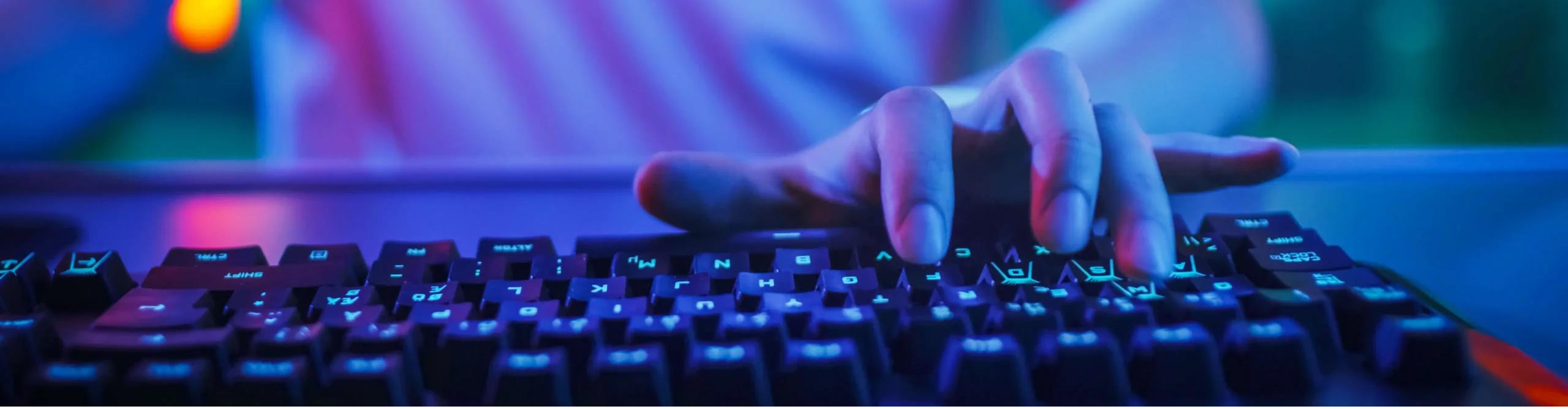
Automated test tools exist that can automatically scan a chip surface (x-/y-/z-axis).

Source: J. Brier, D. Jap. *Testing Feasibility of Back-Side Laser Fault Injection on a Microcontroller.* (2015)

Fault Injection Attacks

Chip Shouter
(EM Pulse Generation: $3300USD)

Laser Station 2
(Laser Glitch Generation: $$$)



Defenses

UNIVERSITY OF TORONTO

Buffer-Overflow Defenses

Many of the attacks discussed have depended on overflowing buffers. The most obvious way to defend against buffer overflow vulnerabilities is not to make them:

- Audit code rigourously
- Use a type-safe language with bounds checking
  - *e.g.*, Java, C#
  - Code will be **memory safe**: compiler will enforce the memory access rules of the language

However, this is not always possible:

- Too much legacy code
- Source code is not available
- Performance may be a concern
- Easy to write C code without correct checks

Other Options for Defense?

Buffer overflow attack requires an input string to be copied into a buffer without bounds checking

- Typical attack requires three steps
  - Control over a location such as return address
  - Overwrite location with guessed address
  - Inject and execute shell code

What is needed for these steps to succeed?

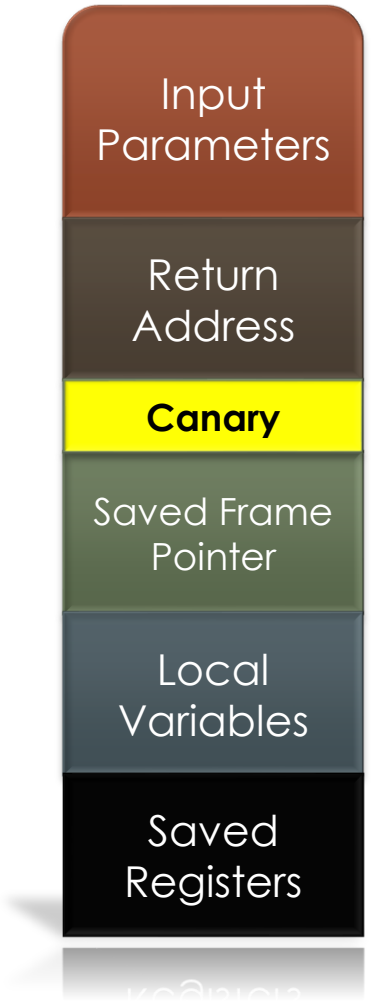
- Return address overwrite
- Target address has to be guessed
- Injected code has to be executable

Let's look at how to detect or prevent each of these steps…

Defending Against Stack Smashing

A number of different protection techniques will prevent the return address from being overwritten

- **Stackshield**
  - Put return addresses on a separate stack with no other data buffers there
- **Stackguard**
  - On a function call, a random **canary** value is placed just before the return address
  - Just before the function returns, the code checks the canary value and, if the value has changed, the program is halted
  - MS VC++ compiler supports it with the *GS flag*
  - Recent GCC compilers support it
  - Does the canary stop format string attacks?



Defending Against Stack Smashing

Run Time Checking: Libsafe (Avaya Labs)

Dynamically loaded library

- Overrids **libc.so** (usually via /etc/ld.so.preload)
- Done at runtime, so doesn't need program recompilation or source-code changes

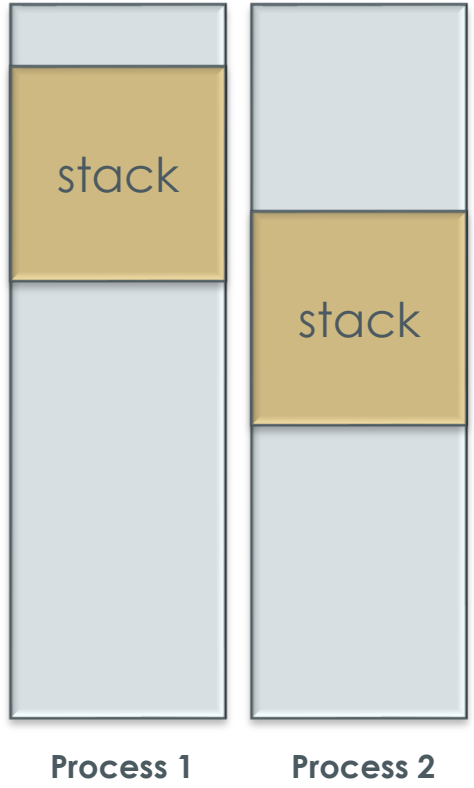
Intercepts calls to dangerous functions such as **`strcpy`**

- Validates sufficient space in current stack frame

ASLR: Address Space Layout Randomization

Recall that the target address (*e.g.*, the buffer's location on the stack) has to be guessed:

- With ASLR, the OS maps the stack of each process at a **randomly** selected location with each invocation
  - An attacker will not be able to easily guess the target address
  - Application will crash rather than executing the attacker's code
  - ASLR also randomizes location of dynamically loaded libraries, making it harder to perform return-into-libc attacks or GOT overwrites
- Linux 2.6 and Windows Vista use ASLR



NX: Non-Executable Pages

If stack is made non-executable, then shellcode on the stack will not execute

- All current Intel and AMD processors allow non-executable pages
  - Page tables have NX protection bit
- Requires support from OS
  - NX implemented in Windows XP SP2 patch

However, non-injection attacks are still possible

- *e.g.*, return-into-libc attacks, argument overwrite attacks

Static Analysis

Analyzes code for common errors or issues that could lead to vulnerabilities.

- **LGTM:** Automated scans of GitHub
- **Flawfinder:** C/C++ source code analysis

https://lgtm.com

https://dwheeler.com/flawfinder

Vulnerability Databases

To aid computer administrators, there are several large databases of vulnerabilities on the Internet:

- **National Vulnerability Database:** http://nvd.nist.gov
- **CERT:** http://www.cert.org
- **SecurityFocus:** http://www.securityfocus.com/vulnerabilities
- **Bugtraq:** http://www.securityfocus.com/archive/
- **OSVDB:** http://www.osvdb.org

For any program and version, one can query these databases and get a description of the vulnerability

Conclusion

- Easy to make a mistake, end up with a vulnerability
  - Exploiting them takes a bit of work, but is not beyond someone who knows what they are doing
- Certain vulnerabilities can be removed by moving to safer languages
  - A lot of vulnerabilities result from uses of pointers and running off the end of arrays

The only real defense is to be aware of what vulnerabilities exist, to be extra careful when creating code and let others audit your code.

Questions?

UNIVERSITY OF TORONTO