TO C, OR NOT TO C

HTSC, 8 Nov 2022

MIDNIGHT BLUE

Systems & Vulnerability Analysis

- Threat Modeling
- Vulnerability Research
- Reverse Engineering

Capability Development

- Attack Scenario Development
- RTO / BAS Development
- Training

Defensive Design

- Architecture Reviews
- SDLC Consultancy

MIFARE CLASSIC+

Broke MIFARE CLASSIC+

Attack integrated in industry-standard Proxmark tool

BLACKBERRY QNX

Multiple kernel 0days

Used in ICS, automotive, avionics, defense

SSD ENCRYPTION

Broke multiple popular SEDs

Impacted Bitlocker defaults

CAR IMMOBILIZERS

Broke Peugeot, Opel, Fiat

Co-developed world's fastest attack against Hitag2 cipher

PROJECT MEMORIA

100+
vulnerabilities

14
TCP/IP stacks analyzed

FORESCOUT
VEDERE LABS

250K+
impacted products

500+
impacted vendors

3 billion
vulnerable devices

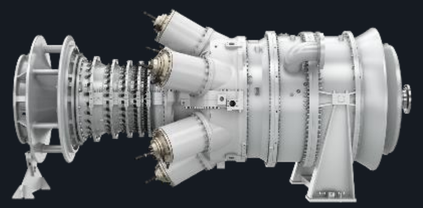
https://www.forescout.com/research-labs/project-memoria/

AFFECTED STACKS

Stack	Vendor	Example OS Integrations
NicheStack	HCC Embedded	NicheTask / ChronOS SEGGER embOS (emNET)
Nucleus NET	Siemens	Nucleus
Treck	Treck / Xilinx	Many
IPnet	Wind River	VxWorks
NetX	Microsoft	ThreadX
FreeBSD	FreeBSD	FreeBSD
uIP	SICS	FreeRTOS, Contiki
PicoTCP	Altran	seL4, TRENTOS
uC/TCP-IP	Micrium	uC/OS-II, uC/OS-III, Cesium
MPLAB NET	Microchip	FreeRTOS, Bare metal
NDKTCPIP	Texas Instruments	TI-RTOS
CycloneTCP	Oryx	FreeRTOS, CMSIS-RTOS
Nut/Net	Ethernut	NutOS
FNET	Freescale	FreeRTOS, CMSIS-RTOS

EXAMPLE AFFECTED PRODUCTS

Gas Turbines

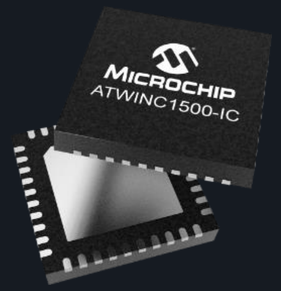


RTUs

PLCs

Infusion Pumps

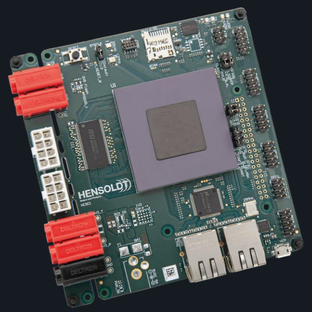
Wifi Modules



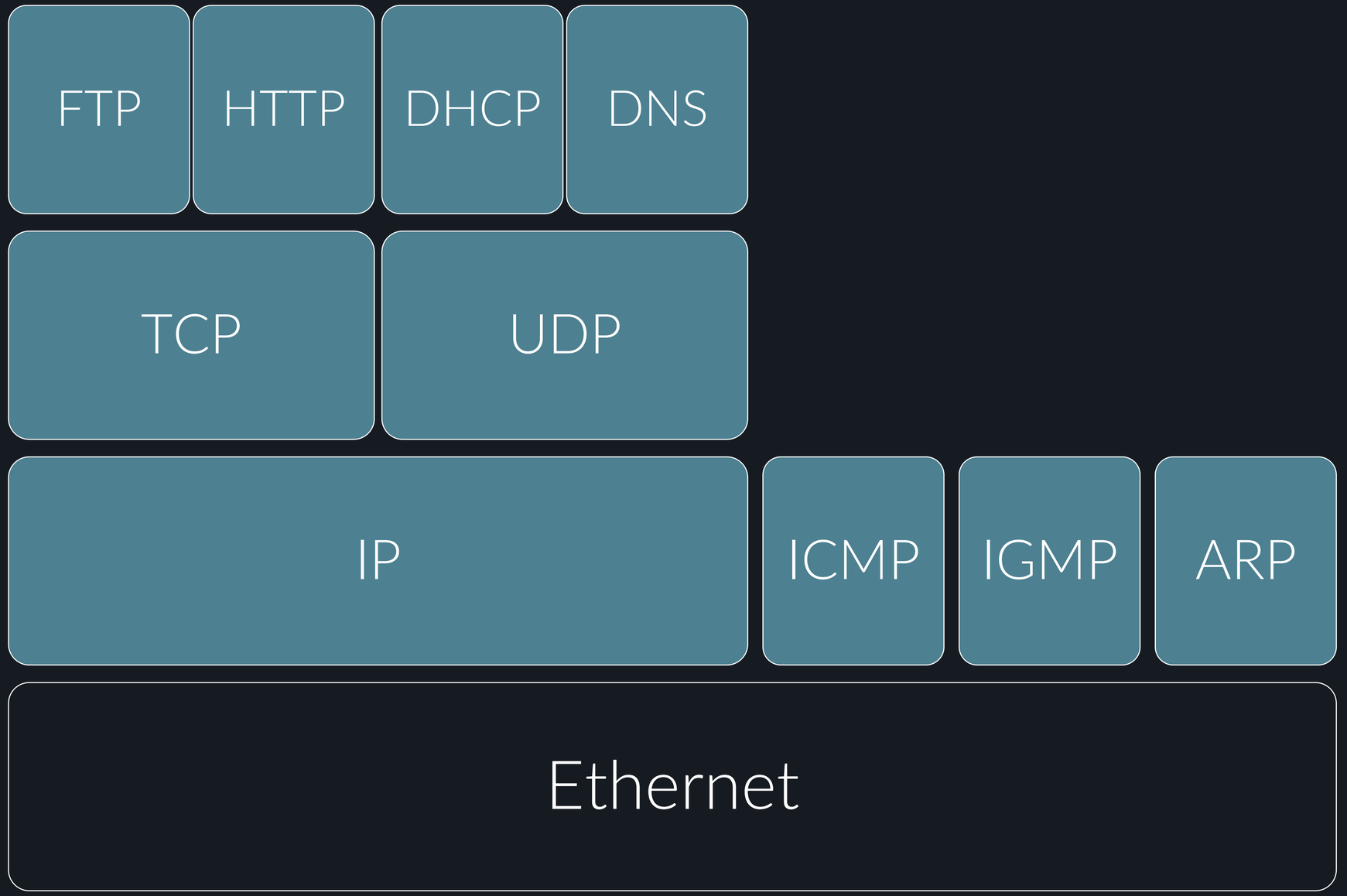
Switches

Printers

Aerospace

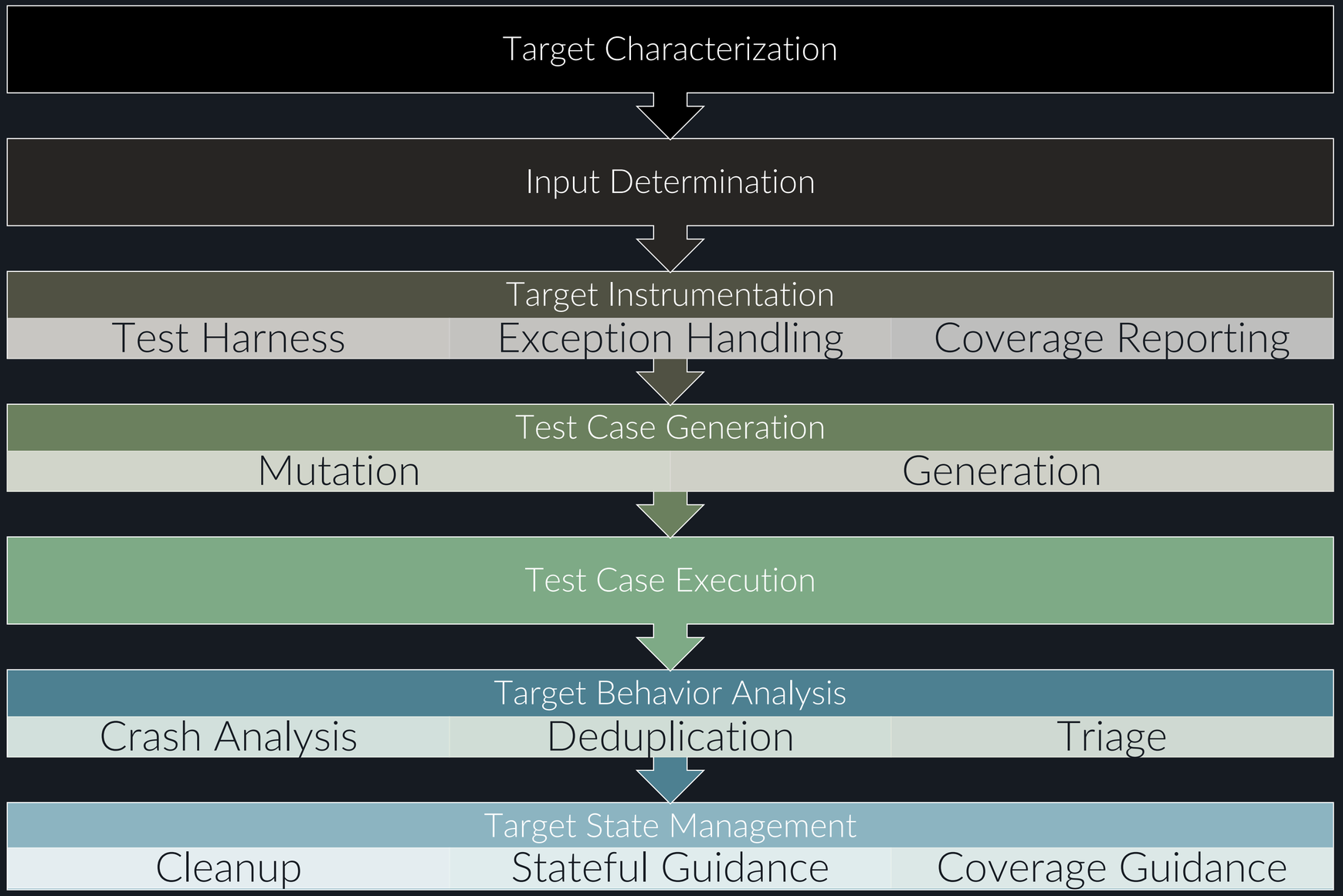


ATTACK SURFACE



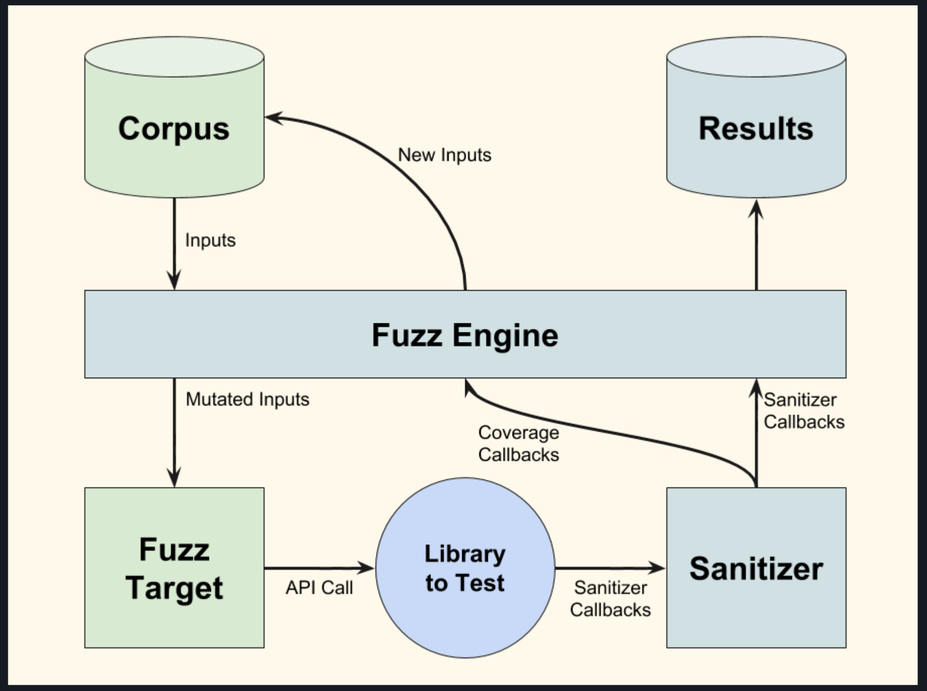
RESEARCH METHODOLOGY

FUZZING


Target Characterization
Input Determination
Target Instrumentation
Test Harness
Exception Handling
Coverage Reporting
Test Case Generation
Mutation
Generation
Test Case Execution
Target Behavior Analysis
Crash Analysis
Deduplication
Triage
Target State Management
Cleanup
Stateful Guidance
Coverage Guidance

LIBFUZZER

- Part of LLVM compiler infrastructure
- White-box
- Coverage-guided
- Mutational
- Original corpus from legit IP traffic
- Write test harness around functions
- Coverage challenges when used out-of-the-box
    - Stateful fuzzing
    - Checksums




```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingWithData(Data, Size);
  return 0;
}
```

VARIANT HUNTING

- Wheel gets reinvented every minute
  - Dozens of TCP/IP stacks, JSON parsers, DNS clients, etc. out there

- If you invent the wheel, you will run into wheel-shaped problems

- People solving similar problems under similar conditions tend to produce similar bugs

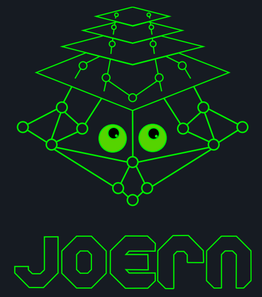
- IDEA: If we see a bunch of vulnerabilities in one implementation, why not (automatically) hunt for the same mistakes in others

ANTI-PATTERNS

- Generalize bundles of vulnerabilities with similar root-causes into classes of anti-patterns

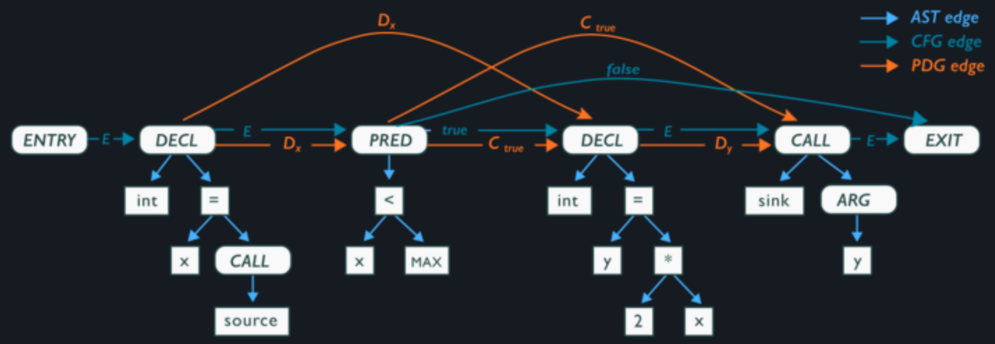
#	Anti-Pattern	Study
1	Absence of bounds checks	AMNESIA:33
2	Misinterpretation of RFCs	AMNESIA:33
3	Shotgun parsing	AMNESIA:33
4	IPv6 extension headers/options	AMNESIA:33
5	Predictable ISN generation	NUMBER:JACK
6	Lack of TXID validation, insufficiently random TXID and source UDP port	NAME:WRECK
7	Lack of domain name character validation	NAME:WRECK
8	Lack of label and name lengths validation	NAME:WRECK
9	Lack of NULL-termination validation	NAME:WRECK
10	Lack of record count fields validation	NAME:WRECK
11	Lack of domain name compression pointer and offset validation	NAME:WRECK

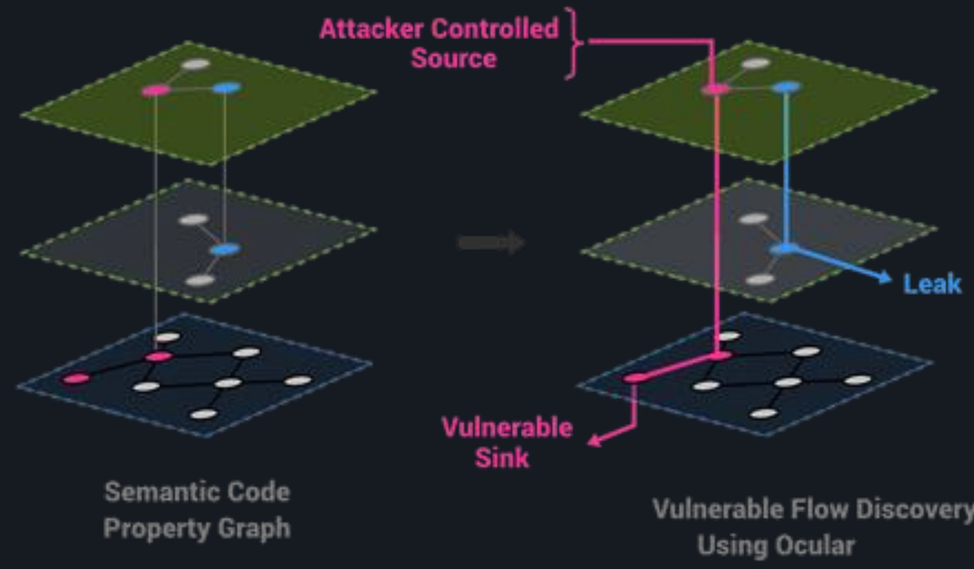
JOERN



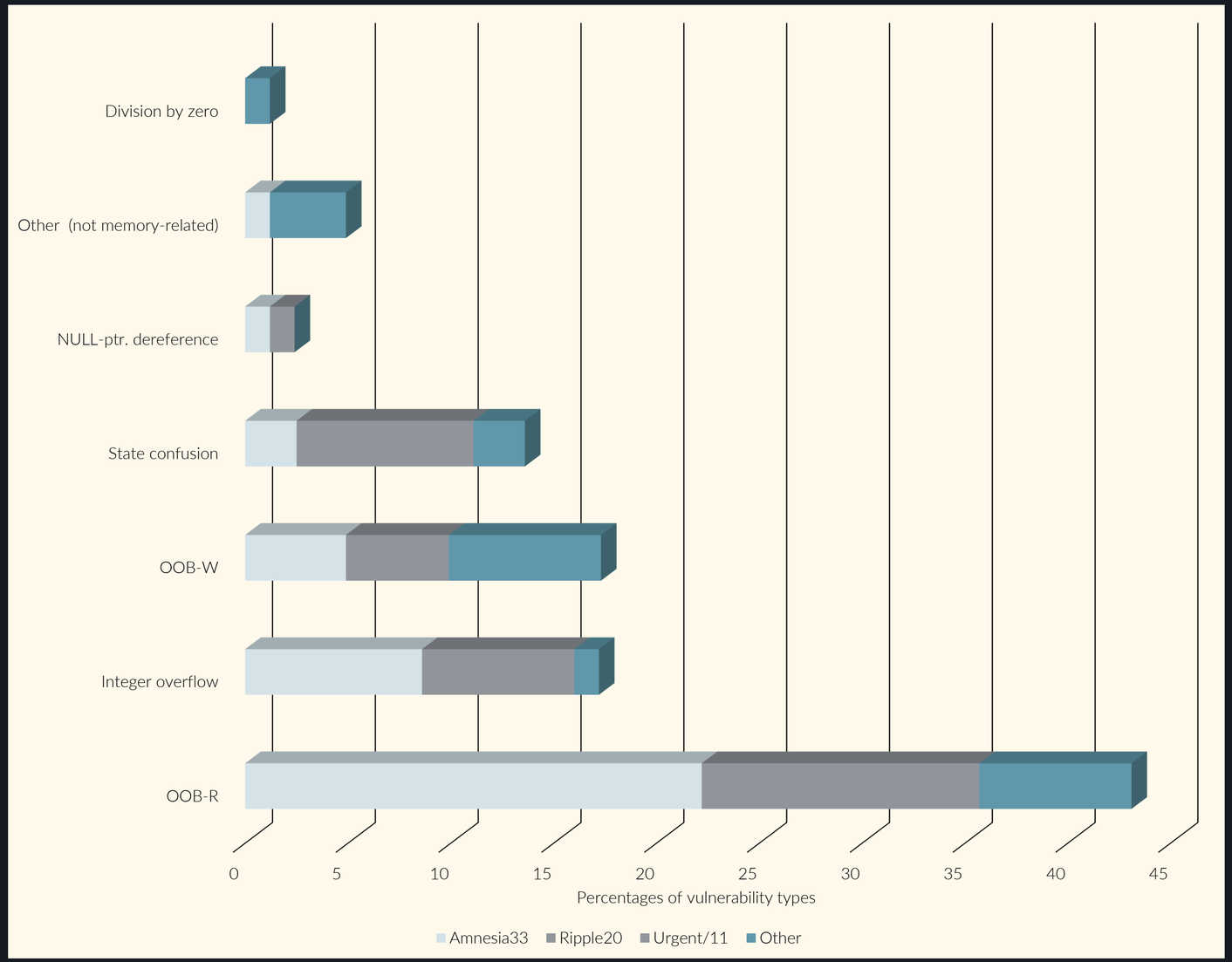
- Derive Code Property Graph (CPG) from source-code
  - Merge Abstract Syntax Tree (AST), Code Flow Graph (CFG), Program Dependence Graph (DPG)
  - Captures syntactic structure, code flow, and data dependencies in one Graph DB

- Formalize anti-patterns as CPG queries in Scala
  - Note: coverage & target specific finetuning make this more suitable for CI/CD integration than one-off vulnerability research





VULNERABILITY TYPES



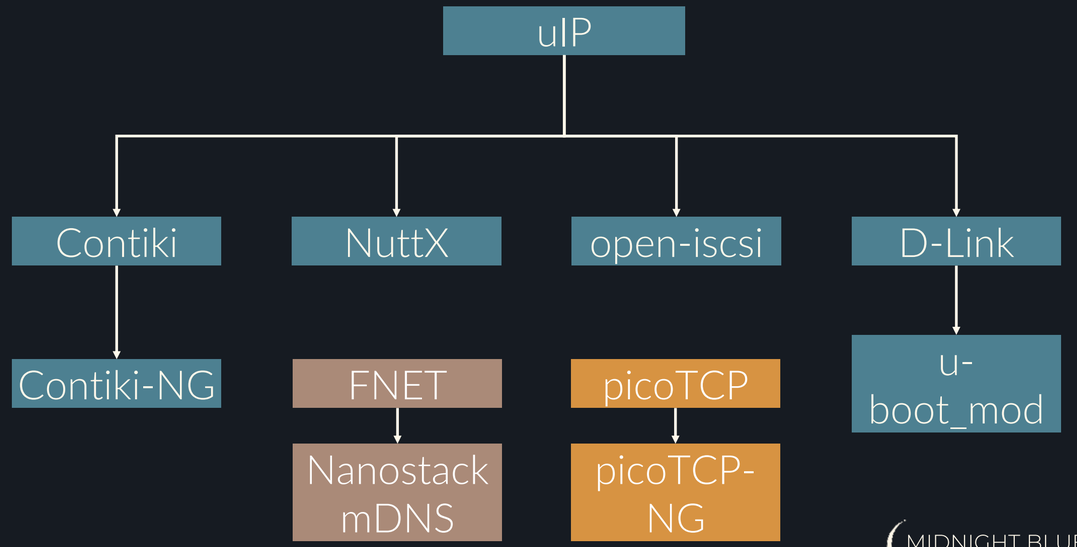
- Mostly violations of memory safety
- Results in DoS / RCE / infoleak
  - Depending on platform & config
- Absence of exploit mitigations
  - No DEP/ASLR/Canaries (lack of MMU/RTOS)
- Some vulns in certain IP stack layers exploitable without any open ports ...

PATCHING

- Who is responsible for patches?
  - Open-source maintainer? White label vendor? OEM?
  - Disclosed to CERT/CC+ICS-CERT, help from Github → no official patches uIP, Contiki, PicoTCP

- Many complications
  - Not all systems have OTA fw updates (e.g. manual, serial ports, etc.)
  - Rare maintenance windows in critical infra
  - Highly complicated supply chains (lack of SBOMs, forks, component copy-paste, etc.)

- Many devices remain vulnerable for loooong time



THE LIMITS OF CERTIFICATION

- Many security certs require fuzzing
  - IEC 62443-4-1
  - GE Achilles ACC
- Yet certified products turn out to still suffer from shallow bugs in common stacks
- Need QA guarantees on fuzz results
  - In-depth methodology description
  - Code coverage figures

9.4 SVV-3: Vulnerability testing

9.4.1 Requirement

A process shall be employed for performing tests that focus on identifying and characterizing potential security vulnerabilities in the product. Known vulnerability testing shall be based upon, at a minimum, recent contents of an established, industry-recognized, public source for known vulnerabilities. Testing shall include:

a) abuse case or malformed or unexpected input testing focused on uncovering security issues. This shall include manual or automated abuse case testing and specialized types of abuse case testing on all external interfaces and protocols for which tools exist. Examples include fuzz testing and network traffic load testing and capacity testing;

Achilles Grammars

Achilles Grammars test for protocol boundary conditions in the device communications. They systematically iterate over each field and combinations of fields to produce repeatable, quantifiable tests of the common types of implementation errors.

Achilles Grammars send invalid, malformed or unexpected packets to the Device Under Test (DUT) to test for vulnerabilities in specific layers of the protocol stack.

Product (ACC SL2)	TCP/IP stack vulns
Nucleus RTOS	NUCLEUS:13 NAME:WRECK
VxWorks 7 RTOS	Urgent/11 NAME:WRECK
Siemens SENTRON PAC4200	Amnesia:33
Schneider ATV6000	Ripple20
Rockwell ControlLogix	Urgent/11 NAME:WRECK

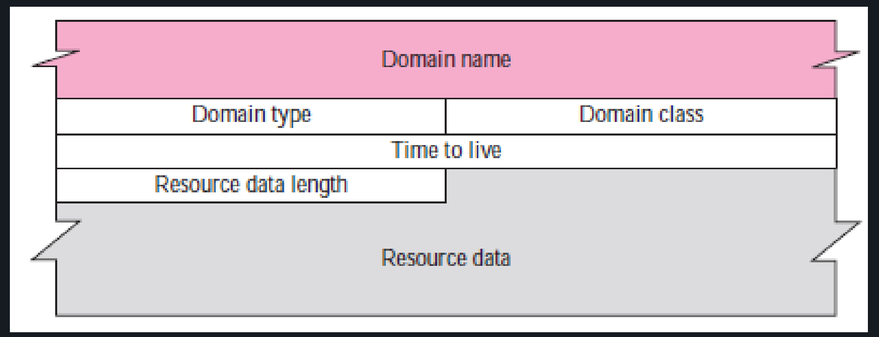
TOWARDS HIGH-ASSURANCE ENGINEERING

“Testing can be used to show the presence of bugs, but never their absence!”

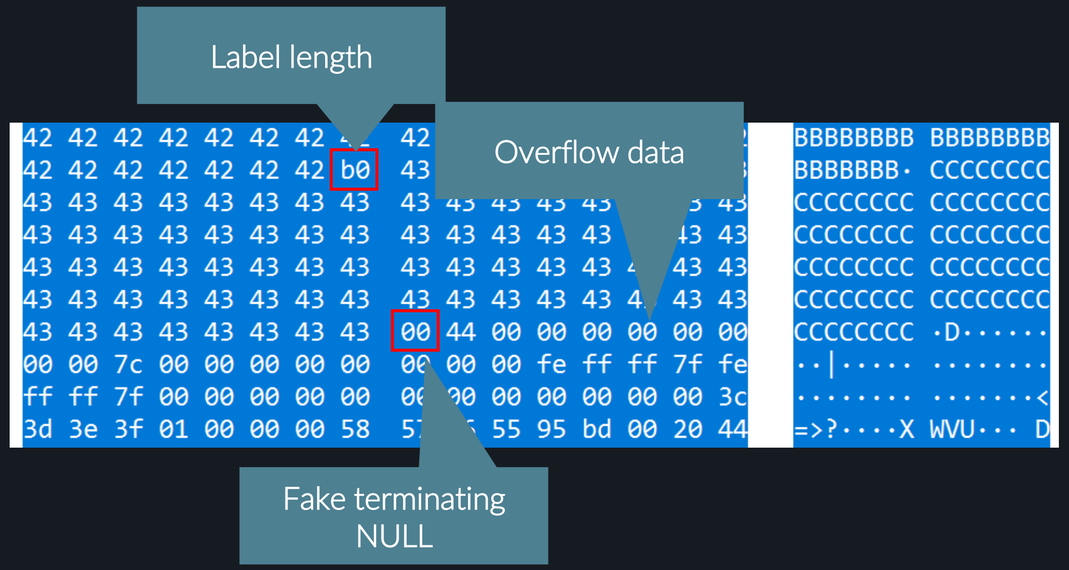
— Edsger W. Dijkstra

ADDRESSING THE ROOT CAUSE

- C(++) is unsafe language → easy to shoot yourself in foot with memory corruption!
  - **Memory Safety**: Only access memory locations they are permitted to access by scope
  - **Type Safety**: Well-typed programs can never result in type errors



CVE-2020-25111

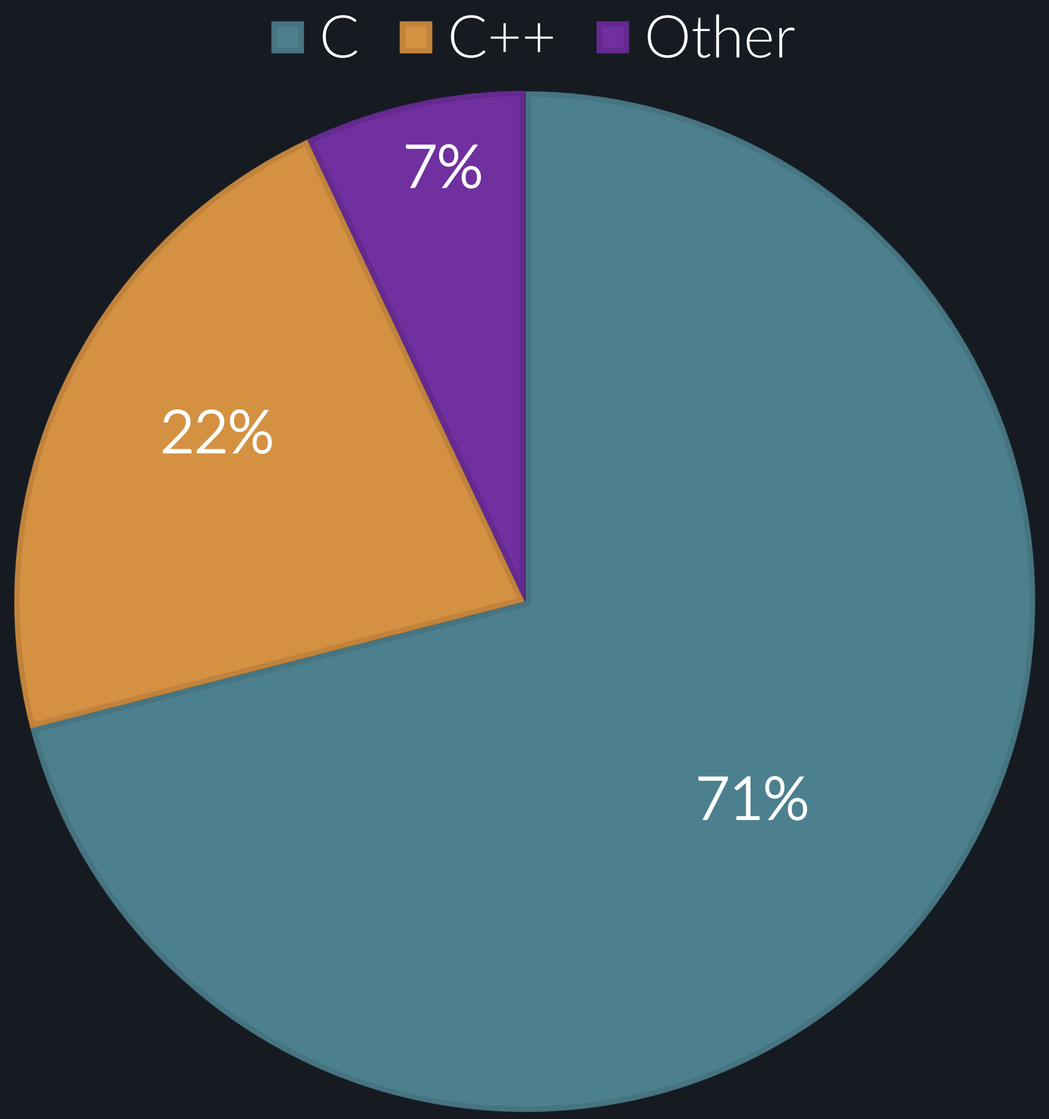



```
168 static uint16_t ScanName(uint8_t * cp, uint8_t ** npp)
169 {
170     uint8_t len;
171     uint16_t rc;
172     uint8_t *np;
173
174     if (*npp) {
175         free(*npp);
176         *npp = 0;
177     }
178
179     if ((*cp & 0xC0) == 0xC0)
180         return 2;
181
182     rc = strlen((char *) cp) + 1;
183     np = *npp = malloc(rc);
184     len = *cp++;
185     while (len) {
186         while (len--)
187             *np++ = *cp++;
188         if ((len = *cp++) != 0)
189             *np++ = '.';
190     }
191     *np = 0;
192
193     return rc;
194 }
```

Attackers explicitly control the allocation of *npp

Attackers explicitly control the number of bytes to be written into *npp (len)

C(++) CONTINUES TO DOMINATE EMBEDDED



** 2017 Barr Group Embedded Systems Safety & Security Survey*

BUT RUST IS GAINING MOMENTUM

Microsoft Azure CTO Wants to Replace C and C++ With Rust

Google shows off KataOS, a secure operating system written in Rust

Protocol Libraries

Our protocol libraries are written in safe Rust and compile to native code. They offer the raw performance of C/C++ combined with state-of-the-art memory and thread safety guarantees. Model-generated bindings are available for C/C++, Java, and .NET.

RUST & MEMORY SAFETY

- Rust is a high-performance, system programming oriented, safe language

- Guarantees safety through data ownership semantics
  - All resources (e.g. variables) have clear owner
  - Others can borrow from owner
  - Owner cannot free/mutate resource while being borrowed

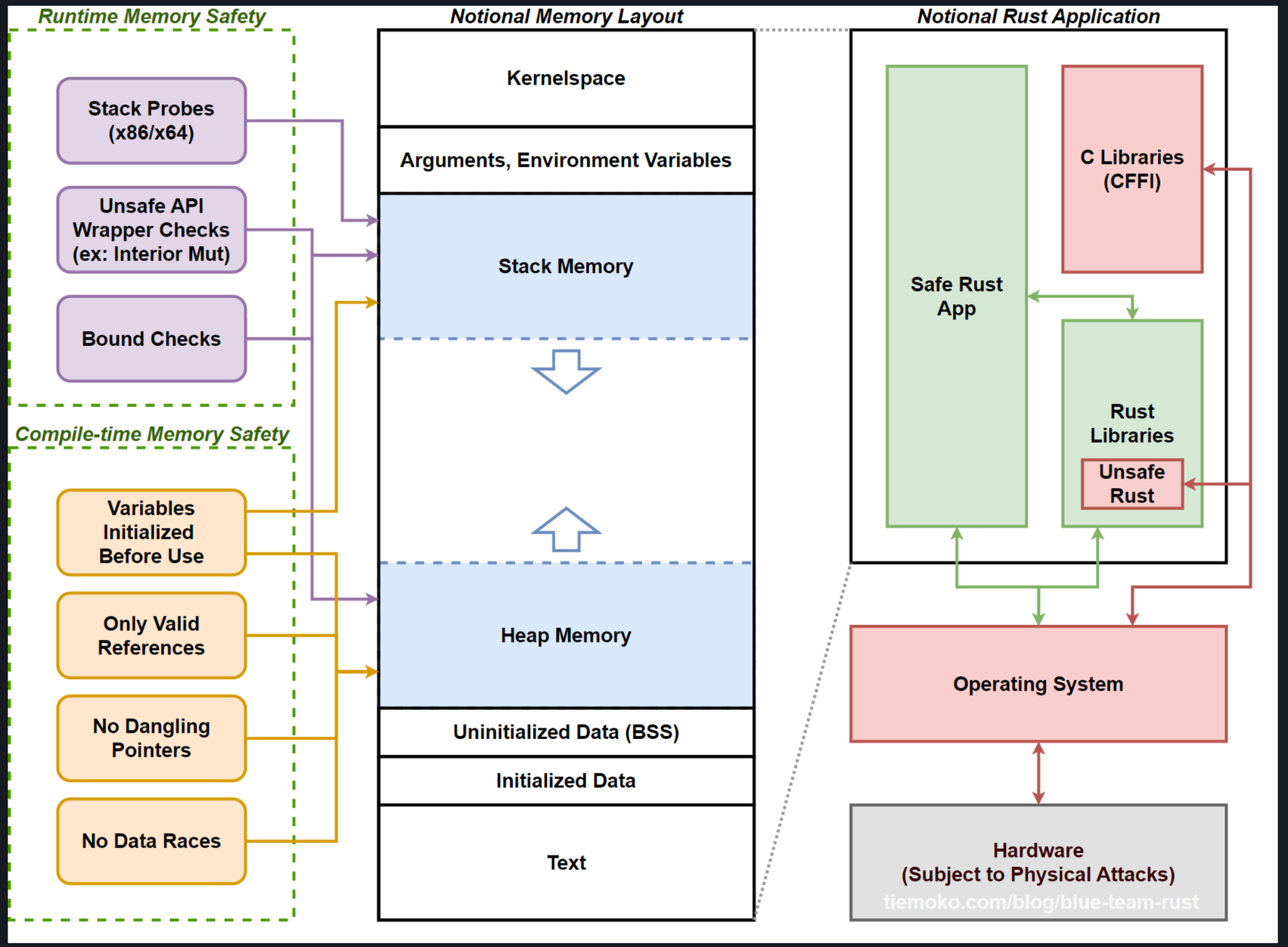
- Guarantees memory safety at compile-time, no costly runtime or garbage collector

> Our analysis result shows that Rust can keep its promise that all memory-safety bugs require unsafe code, and many memory-safety bugs in our dataset are mild soundness issues that only leave a possibility to write memory-safety bugs without unsafe code.

- NOTE: Rust helps you against memory corruption
  - Not with weak crypto, SCA/FI, logic bugs, cmd injection, etc.

Xu et al. - Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs

RUST MEMORY SAFETY MODEL



UNSAFE RUST

- Guarantees are very broad but not (yet) universal

- unsafe keyword drops some checks for limited scope, allows Undefined Behavior (UB)

- Why unsafe?
  - Some code hard to implement under safe rules (e.g. MMIO, certain low-level HW interaction, doubly-linked lists, etc.)

  - Foreign Function Interface (FFI) to interact with
    1. C libraries (especially for large proprietary codebases, legacy protocol stacks, binblobs with lost source)
    2. Underlying OS (since there's no fat runtime to do that for us)

- Parts of Rust standard library are marked unsafe

FURTHER GOTCHAS

- Rare compiler bugs affecting soundness could violate safety
- Memory leaks & panic handler invocation can still happen
  - Typically results in DoS (worrying in cyber-physical systems)
  - Explicit error handling can cover this
- Integer overflow checks depend on compiler settings
  - Debug → panic, Release → wraparound
- Heavily constrained #![no_std] target systems tend to lack MPUs, NX bit, etc.
  - Only compile-time memory safety, no run-time guarantees
- Regardless, attack surface reduced by orders of magnitude compared to C(++)

HARDENING RUST CODEBASES

- If full codebase can’t be in Rust → high-risk targets as hardened Rust components (e.g. protocol stacks, TEE, etc.)

- Iteratively reduce C(++) footprint in codebase + focus security attention on those parts
    - Hardening, fuzzing, audits, etc.

- Cover unsafe Rust & gotchas by integrating fuzzing into CI/CD pipeline
    - Cargo-fuzz (libFuzzer wrapper)

- CONCLUSION
    - Rust massively reduces memory corruption attack surface by tackling root cause
    - Thus reducing cost of vulnerability mgmt. programs
    - While remaining performant

GET IN TOUCH WITH US

WEB

www.midnightblue.nl

CONTACT

sales@midnightblue.nl