

bsu-tool: Behavioral Sleuth for USB

Portland State University CS Capstone Project Proposal

Sponsoring Faculty: Bart Massey, Associate Professor of Computer Science, PSU

Team Size: 5–8 senior undergraduates + student lead

Duration: 20 weeks (four five-week milestones)

Language: Python or Rust (see Section 4)

AI Tooling: Claude / Claude Code — accounts funded by sponsor

Target Platform: Linux (primary); Windows live capture (stretch goal)

License: The work must be published under an MIT / Apache 2.0 dual license.

Abstract

`bsu-tool` (Behavioral Sleuth for USB) is a command-line tool and AI agent interface for analyzing USB device protocols on Linux. It reads USB traffic captures produced by the standard Linux `usbmon` subsystem, decodes the low-level USB exchanges recorded in them, and exposes an analysis interface to an AI coding assistant via the Model Context Protocol (MCP). The intended workflow is a collaboration between a human analyst and an AI assistant: the analyst captures traffic while operating the device under AI instruction, and the AI assistant uses `bsu-tool` to explore the capture, infer the protocol structure, and produce a human-readable protocol description. The Linux implementation is immediately useful to open-source driver developers and security researchers, who already work primarily on Linux and currently rely on manual packet inspection. The longer-term goal is to extend this to Windows-only USB devices — devices whose drivers exist only on Windows and whose protocols are undocumented — enabling analysis of hardware that is currently very difficult to reverse-engineer. Windows live-capture support is a stretch goal the sponsor expects to complete after the Capstone if the team does not get there first.

Why You Should Pick This Project

This project combines systems programming, protocol analysis, and AI-assisted tooling — three of the most in-demand skill areas in industry right now. By the end of 20 weeks, you will have:

- Built and shipped a real open-source tool that does something no existing tool does
- Written low-level systems code (binary parsing, byte-level protocol decoding) against real hardware — physical devices the sponsor will provide for the team to work with
- Designed and implemented an MCP (Model Context Protocol) server — a technology that is actively reshaping how AI systems interact with domain-specific tools
- Used Claude and Claude Code as genuine development accelerators throughout, not as toys
- Produced a portfolio artifact that is technically substantial and immediately recognizable to systems engineers and security researchers

USB reverse engineering is a real skill used in open-source driver development, hardware security research, and embedded systems work. You will understand how USB actually works at the wire level — something most CS graduates never learn. The domain is specialized enough to be distinctive and concrete enough to explain in an interview. Writing code that figures out how a physical device works from first principles is genuinely satisfying.

Is this project within reach for our team? Yes. This project is specifically designed around AI-assisted development. The sponsor expects students to use Claude, Claude Code, and other AI tools extensively: for learning unfamiliar technologies, for exploring design options, for writing and debugging code, and for understanding USB protocol details. This is not a project where AI is a shortcut or a cheat — it is a project where AI is a necessary tool, and learning to use it effectively *is part of the work*. A team of motivated students of average skill, using these tools seriously, can complete this project. A team of strong students ignoring the AI tooling probably cannot.

The sponsor has 26 years of experience mentoring student projects and has scoped this one deliberately to be achievable. Weekly check-ins provide a safety net: if the team is falling behind, scope gets cut — not students.

Language choice is discussed in detail in Section 4, including the sponsor’s perspective and the tradeoffs between Python and Rust for this project.

The sponsor will fund Claude Max accounts for the student lead and one other team member, and standard Claude accounts for the rest.

1. Introduction and Motivation

Universal Serial Bus (USB) is the dominant interface for connecting peripheral devices to computers. Despite its ubiquity, understanding the communication protocol between a USB device and its host driver remains a highly specialized skill. Analysts who need to write open-source drivers, validate device behavior, or study protocol correctness currently depend on a fragmented toolkit: packet capture tools such as Wireshark, decade-old analysis scripts, and manual inspection of raw byte streams.

`bsu-tool` addresses this gap by providing a structured, AI-assisted pipeline for capturing, decoding, and interpreting USB traffic. The tool ingests standard `pcap-ng` capture files produced by the Linux `usbmon` subsystem, decodes USB Request Blocks (URBs) into structured records, and exposes an analysis API through the Model Context Protocol (MCP). This allows an AI coding assistant such as Claude Code to drive a semi-automated analysis session, with a human operator acting only when physical interaction with the device is required.

The immediate beneficiaries are developers writing or validating open-source Linux USB drivers, security researchers studying device behavior, and educators teaching operating systems or protocol analysis.

1.1 Longer-Term Vision

Linux is the primary platform for open-source USB driver development and hardware security research — the people most likely to need a tool like this already work there. A `bsu-tool` that runs on Linux against devices with existing Linux drivers is immediately useful: it lets developers understand undocumented vendor protocols, validate driver behavior against wire-level ground truth, and bring AI-assisted analysis to a workflow that currently relies on manual packet inspection. This is the core deliverable, and it stands on its own.

The longer-term goal is to extend this to USB devices that have drivers only on Windows — industrial instruments, specialty peripherals, legacy hardware — whose protocols are undocumented

and currently require a Windows machine and significant expertise to reverse-engineer. The vision is a workflow where a user on Windows plugs in a mystery device, runs `bsu-tool` with live capture via USBPcap, and has an AI assistant walk them through the protocol within minutes, producing documentation and working code. The Windows live-capture backend is an explicit stretch goal described in Milestone 4; if the team does not reach it, the sponsor expects to complete it afterward. The Linux work is not a stepping stone — it is the foundation of the analysis engine — but adding Windows capture support is what unlocks the tool’s broadest audience.

1.2 The Software Engineering Context

This project is scoped as a software engineering exercise, not just a programming task. Students will work through a realistic industry development cycle: requirements analysis, architectural design, iterative implementation with automated testing, ground-truth validation against known-correct device captures, and delivery of documented installable open-source software. The domain provides genuine technical depth while remaining tractable for a well-supported team. Weekly check-ins with the sponsor provide mentorship and course correction without micromanagement.

AI tools are central to this project, not peripheral. Students are expected to use Claude and Claude Code extensively throughout — to learn new technologies, to explore design alternatives, to write and debug code, and to understand protocol details. Students should not feel that this “doesn’t count” or reflects poorly on their abilities. Using AI tools effectively to deliver working software *is* the skill being practiced. The project is designed with this in mind: it would be very difficult to complete without AI assistance, and quite manageable with it.

1.3 Why This Project Is Timely

Two converging developments make this project particularly relevant. First, the Model Context Protocol has emerged as a standard interface for connecting AI assistants to domain-specific tools, and the ecosystem of MCP servers for systems-level analysis is essentially empty — there is real opportunity to build something that matters. Second, AI-assisted coding environments dramatically accelerate the implementation of structured, well-specified systems. A well-supported undergraduate team with Claude Code access is now capable of producing work that would have required more senior engineers only a few years ago.

2. High-Level System Description

`bsu-tool` is a command-line tool and AI agent interface for capturing, decoding, and analyzing USB traffic on Linux. At its core, it reads standard packet capture files produced by the Linux `usbmon` subsystem, makes sense of the USB protocol exchanges recorded in them, and exposes that understanding to an AI coding assistant through the Model Context Protocol.

The intended usage pattern is a conversation between an analyst and an AI assistant, mediated by `bsu-tool`: the analyst captures traffic from a device, the AI assistant uses `bsu-tool`’s interface to explore the capture, and the two collaborate to produce a human-readable description of the device’s protocol. The analyst’s only required physical intervention is operating the device during capture.

The team is expected to work out the detailed architecture, component breakdown, and API design

as part of the project — requirements elicitation and specification are explicit learning goals. The sponsor will provide guidance and feedback at weekly check-ins. The following pointers are offered to get the team oriented, not to pre-empt their design work.

Relevant standards, formats, and frameworks:

- The Linux `usbmon` subsystem exposes USB traffic as pcap-ng files, recordable with Wireshark or `tshark`. The pcap-ng format is documented in the Wireshark wiki and is straightforward to parse.
- USB Request Blocks (URBs) are the fundamental unit of USB communication. The USB 2.0 specification (freely available) defines their structure. *USB in a NutShell* (free online) is a more accessible introduction.
- The Model Context Protocol (MCP) is an open standard for connecting AI assistants to external tools and data sources. It is documented at <https://modelcontextprotocol.io>. Mature SDKs exist for both Python (`mcp`) and Rust (`rmcp`). MCP servers expose typed *tools* that an AI assistant can invoke — roughly analogous to a REST API designed for LLM consumption. The protocol is not complicated; a basic MCP server can be written in an afternoon using an SDK.
- `usbrevue` (Python, partially maintained) is a prior art project for USB pcap analysis. The team is not expected to use or fork it, but its source is useful reference for URB field layouts and pcap parsing approaches.
- Wireshark's USB dissector source is the authoritative reference for pcap-ng link-layer format details, and Wireshark itself is invaluable for visually inspecting captures during development.

Devices in scope: The tool targets USB devices that use vendor-specific protocols — devices that communicate in ways defined by the manufacturer rather than a USB standard device class. Devices covered by standard USB Device Class drivers (HID, Audio, Video, CDC, Mass Storage, etc.) are explicitly out of scope: they are well-documented by the relevant class specifications and `bsu-tool` adds little value there.

3. Software Requirements

The team is responsible for producing a complete software requirements specification (SRS) as part of Milestone 1. The following are high-level requirements the sponsor considers non-negotiable; everything else is for the team to elicit, specify, and prioritize.

Functional requirements (high level):

- Accept USB traffic captures from Linux `usbmon` as input
- Decode and present USB exchanges in a structured, human-readable form
- Expose analysis capabilities to an AI assistant via MCP
- Enable an AI assistant to drive a protocol analysis session with minimal human intervention

Stretch goals (functional):

- Generate skeleton code (Rust or Python) that communicates with an analyzed device, based on an inferred protocol hypothesis
- Windows live-capture support via USBPcap, enabling analysis of devices without an existing Linux driver

Non-functional requirements:

- Automated tests must be present and must pass in CI
 - The tool must be installable from source on a stock Debian or Ubuntu system
 - Code must be linted and formatted (clippy for Rust; ruff for Python) with no warnings on every PR
 - Python code must be fully type-annotated and pass `pyright` with no errors on every PR (not applicable if Rust is chosen)
 - All public interfaces must be documented
 - The repository must carry an MIT / Apache 2.0 dual license
-

4. Required Skills and Background

The following table summarizes the skill areas relevant to this project. No single team member needs all of these; the team should self-organize around individual strengths early in the project.

| Skill Area | Relevance | Required of Team? |
|---|---|---|
| Python systems programming | Core implementation language if Python chosen | Yes (most members) |
| Rust systems programming | Core implementation language if Rust chosen | Yes, if Rust chosen (2–3 members minimum) |
| Binary data parsing / struct layout | pcap-ng and URB decoding | Helpful |
| USB fundamentals (transfers, endpoints) | Protocol understanding | Helpful; sponsor can teach |
| Network/protocol packet analysis | Wireshark familiarity | Helpful |
| Automated testing (unit + integration) | Quality gates throughout | Yes |
| Git / GitHub collaborative workflow | Team coordination | Yes |
| Technical writing | SRS and documentation deliverables | Yes (at least 1 member) |
| JSON / structured data APIs | MCP tool design | Yes |
| AI-assisted development (Claude Code) | Core workflow tool | Will be learned on project |

On language choice: The sponsor has a mild preference for Rust. Rust’s strong static type system and ownership model make it less error-prone for the kind of low-level binary parsing this project requires, and Rust binaries are more portable across Linux distributions than Python environments. Claude Code is also quite capable in Rust — AI assistance reduces the traditional learning-curve cost considerably. Still, Python is the right choice for most teams. It is faster to get started with, has excellent libraries for every part of this project, and is completely appropriate for the work. If the

team has two or more members with solid working Rust knowledge, Rust is worth considering; otherwise, Python is the sensible default and the sponsor is entirely happy with it.

If Python is chosen, **static type annotations are required throughout**. All code must be fully annotated and pass `pyright` with no errors. This is non-negotiable: type annotations catch a significant class of bugs in binary parsing code and make the codebase much easier to maintain. `ruff` handles linting and formatting; `pyright` handles type checking — both must pass clean on every PR.

The team should pick one language and stick with it. Mixed-language teams without prior experience designing cross-language interfaces tend to produce unmaintainable code.

5. Milestones

The project uses four five-week milestones. Each milestone builds directly on the previous one. The Week 5 milestone is deliberately scoped to be achievable and demonstrable — it represents the foundation that everything else depends on, and completing it solidly is more valuable than rushing toward later features.

Milestone 1 — Weeks 1–5: Foundation

Theme: *Parse real USB traffic and prove the architecture works*

Deliverables:

- Language and toolchain decision documented and justified
- Development environment documented and reproducible (README with setup instructions)
- Repository established with CI running on every PR (GitHub Actions or equivalent)
- Draft software requirements specification (SRS) covering the full project scope, reviewed and approved by sponsor at Week 5 check-in
- pcap-ng reader able to parse captures made by the team from the provided devices
- URB decoder handling Control and Bulk transfers, validated against reference captures from the provided devices
- Basic CLI that prints a human-readable summary of a capture (devices, endpoints, packet counts)

Success Criteria:

- CI passes on all PRs
 - Capture files made by the team decode without errors and produce correct device summaries
 - CLI output is legible and useful to a human analyst
 - SRS is complete enough that the sponsor can assess whether scope is appropriate
 - Most work has been completed using AI assistance; a description of how AI tools were used must be provided at the Week 5 check-in
-

Milestone 2 — Weeks 6–10: Session Model and MCP Interface

Theme: *Understand what the device is doing, and expose that understanding to an AI assistant*

Deliverables:

- URB decoder extended to Interrupt transfers (Isochronous is out of scope — it is used primarily for audio/video streaming, which falls under Device Class drivers excluded from this project)
- MCP server implemented: server starts, accepts connections from Claude Code, and exposes tools for loading captures, enumerating devices and endpoints, and retrieving decoded packets
- Session model implemented: persistent state across a capture including device registry, per-endpoint history, and a named marker system for correlating physical device actions with captured traffic
- Ground-truth validation report: a Claude Code session used to analyze at least two of the provided reference devices, with results documented
- Updated user guide covering all implemented MCP tools with examples

Success Criteria:

- All implemented MCP tools return correct structured results for reference captures
 - Marker system correctly isolates packet sets in at least two documented test scenarios
 - A Claude Code session can enumerate devices and retrieve packets with no human intervention beyond loading the capture
 - Ground-truth validation report is complete and reviewed by sponsor
-

Milestone 3 — Weeks 11–15: Analysis and Protocol Understanding

Theme: *From packet stream to protocol description*

Deliverables:

- Repeated URB sequence detection implemented and validated against reference devices
- Command/response pairing implemented and validated
- MCP tool for protocol hypothesis implemented: given a capture, the AI assistant can produce a structured human-readable description of the device's protocol
- End-to-end demonstration: load a capture of a provided device, run a Claude Code analysis session, and produce a protocol description accurate enough that a developer could use it to write a driver
- Extended ground-truth validation covering at least two of the provided devices

Stretch goal (if time permits): MCP tool for skeleton code generation — given a protocol hypothesis, emit compilable Rust or Python code that communicates with the device.

Success Criteria:

- Protocol hypothesis for at least one provided device correctly identifies its primary communication pattern, as assessed by sponsor
- End-to-end demo is recorded and can be reproduced by the sponsor independently

- Claude Code session drives the analysis with no human intervention beyond loading the capture and operating the device
-

Milestone 4 — Weeks 16–20: Polish, Documentation, and Delivery

Theme: *Ship something other people can actually use*

Deliverables:

- All known defects from Milestones 1–3 resolved or documented as known limitations
- Complete user guide: installation, basic usage, and at least two worked examples with provided reference devices
- API reference documentation generated from doc comments
- Automated test suite passing in CI with coverage of all core decoding paths
- Installation verified on a stock Debian 12 or Ubuntu 24.04 system by a team member other than the one who wrote the installer
- Final presentation to sponsor and Capstone coordinator
- GitHub repository in public release state with LICENSE file

Stretch goals (if time permits): - Skeleton code generation tool (if not completed in Milestone 3) - Windows live-capture backend via USBPcap named pipe — the first step toward the longer-term goal of analyzing Windows-only devices (see Section 1.1)

Success Criteria:

- Installation succeeds from scratch on the reference Linux distribution
 - All Milestone 1–3 success criteria remain passing
 - Documentation is complete enough that a person unfamiliar with the project can complete a basic analysis session on Linux against one of the provided reference devices using only the user guide
 - Final presentation clearly communicates the architecture, the SE process, and lessons learned about AI-assisted development
-

6. Resources and Support

Reference devices: The sponsor will provide a small set of inexpensive USB devices with existing Linux drivers for the team to work with. Devices will be chosen to have vendor-specific protocols (not standard USB Device Class protocols) of varying complexity. Candidates include USB relay boards, USB-connected sensors, and similar embedded-style peripherals. The team should make their own captures using Wireshark or `tshark` against the `usbmon` interface — this is a straightforward process well within reach from the first week, and making captures is itself a useful way to build intuition about the devices.

Existing reference material:

- `usbrevue` (Python, partially maintained) provides useful reference for URB field layouts and pcap-ng parsing approaches. The team is not expected to use or fork it, but the source is worth reading.

- The Wireshark USB dissector source is the authoritative reference for pcap-ng link-layer format details. Wireshark itself is invaluable for visually inspecting captures during development.
- The MCP Python SDK (`mcp`, available on PyPI) and Rust SDK (`rmcp`, available on crates.io) both include example servers and are well-documented. The MCP specification at modelcontextprotocol.io is the normative reference. Several open-source MCP server projects (filesystem, Git, database servers) provide useful patterns for tool design.
- Claude itself is a resource. Students should ask Claude about USB, MCP, pcap-ng parsing, and any other unfamiliar topic before reaching for a textbook.

AI tooling: The sponsor will fund Claude Max accounts for the student lead and one other team member, and standard Claude accounts for remaining team members. Students are expected to use Claude and Claude Code heavily throughout the project — for learning unfamiliar technologies, for design discussions, for writing and reviewing code, and for understanding USB and MCP details. This is not optional or supplementary; it is the intended development model. Students who are new to a technology should start by asking Claude to explain it before reaching for documentation. Usage patterns and reflections on what worked should be documented at each milestone check-in.

Weekly check-ins and sponsor involvement: The sponsor will hold a weekly 30–60 minute check-in with the full team. The agenda is team-driven: blockers, design questions, scope concerns. The sponsor does not attend standups or review day-to-day PRs unless requested. If the team is falling behind, scope will be adjusted at check-ins — the milestone definitions are a plan, not a contract.

The sponsor expects to be asked for a lot of help on this project and is genuinely excited to provide it. USB protocols, binary parsing, systems programming, MCP design, AI-assisted development workflows — these are topics the sponsor knows well and enjoys discussing. Beyond the technical, the sponsor is a former Capstone coordinator and is happy to advise on software engineering process, team dynamics, and project management as well. No question is too basic. Students should not struggle in silence when a quick check-in question could unblock them in five minutes. Asking for help early and often is a sign of good engineering judgment, not weakness.

USB background: Team members without prior USB experience should ask Claude for an introduction before the Week 2 check-in, and follow up with Chapters 1–5 of *USB in a NutShell* (available free online) for a more thorough grounding. The USB 2.0 specification (freely available; Chapter 8, Wire Protocol and Chapter 9, Device Framework are the most relevant sections) is the normative reference for anything requiring precision. The sponsor can answer USB questions at check-ins.

7. Risks and Mitigations

| Risk | Likelihood | Impact | Mitigation |
|---|------------|--------|---|
| Team chooses Rust without sufficient prior experience | Medium | High | Python is the preferred default; team should assess their own skills honestly |

| Risk | Likelihood | Impact | Mitigation |
|--|------------|--------|---|
| MCP specification or tooling changes mid-project | Low | Low | Pin MCP SDK version; monitor changelog |
| Scope creep into Windows live capture | Medium | Medium | Windows port is explicitly a stretch goal; sponsor enforces boundary at check-ins |
| Scope creep into USB Device Class devices | Low | Medium | Explicitly out of scope; sponsor flags at check-ins |
| Team velocity lower than estimated | Medium | High | Milestone 1 is deliberately conservative; scope is cut from Milestone 4 first |

8. Definition of Done

The project is complete when:

1. The `bsu-tool` repository is public on GitHub under an MIT / Apache 2.0 dual license
2. A new user on Linux can install the tool and complete a basic analysis session against one of the provided reference devices using only the written documentation
3. All Milestone success criteria are passing
4. The team has delivered a final presentation
5. The sponsor has accepted the codebase

Proposal prepared by Bart Massey, Portland State University Department of Computer Science, with assistance from Claude (Anthropic).

Questions: bart@cs.pdx.edu