Fake SSH Play: Playing *Bad Apple* to SSH Clients, and Session Hold as a Measurable Honeypot Property

Daozhendong Yang
School of Intelligent Science and Engineering,
Yunnan Technology and Business University, Kunming, China
yxpil|hi@yxpil.com
https://github.com/yxpil/Fake_SSH_Play

Abstract

FAKE SSH PLAY is a small SSH responder that accepts every authentication attempt and, instead of a shell, plays a 2,193-frame ASCII rendering of the *Bad Apple* shadow-art video over the SSH channel at 33 ms per frame, then closes the session. It is written in Rust on `russh` and ships a web dashboard. Accepting any credential is textbook honeypot behaviour and we do not present it as a contribution. The narrower question is what a responder with *server-determined* content and duration is good for. In a fake shell the session length is chosen by the client, so duration measures the client; in a fixed animation the server removes its own variance from the same quantity, which makes "how long did the client stay" comparable across responders. That is a modest methodological claim, and this paper treats it as one. We document the system from its source, report its measured envelope — 5,219,755 bytes of channel payload, 2,380 bytes per frame, a largest frame that is 7.2% of the RFC 4253 maximum packet, 72.4 s of nominal playback, 5.2 MB of resident memory per concurrent session, and 1.07 μs of CPU per frame against a 33 ms budget — and specify, without running, a controlled comparison against an interactive fake shell. The repository contains no captured sessions, so this paper reports no engagement results, no detection or deterrence claim, and no attribution of any kind.

Artifact. The responder, the frame asset, the converter, and the scripts that produce every measured number and figure in this paper are at `https://github.com/yxpil/Fake_SSH_Play`, with the measurement scripts included in this submission. The repository is nine commits by one author and has two release tags.

1 Introduction

An SSH honeypot is a server that expects to be attacked. It presents the SSH protocol, accepts or rejects authentication according to some policy, and records what happens next. The literature has a well-developed vocabulary for the middle of that pipeline — low, medium, and high interaction, emulated filesystems, recorded sessions Spitzner (2002); Provos and Holz (2007); Provos (2004) — and two mature artifacts, Kippo Tamminen and Cowrie Oosterhof, define what most operators mean by an SSH honeypot: a fake shell that accepts a login and then pretends to be a Unix system. The famous tarpits sit at the other end: Endlessh Wellons accepts a TCP connection and dribbles an SSH banner one byte at a time, never completing the handshake, for as long as the client waits.

FAKE SSH PLAY is neither. It completes the SSH handshake, accepts any credential, opens a session channel, and then plays a video. There is no shell, no filesystem, no prompt, and no command parser. The client receives a sequence of ANSI-escaped ASCII frames, a caption that counts up to the last frame, a closing message, and a disconnect. This is, on its face, a joke: a honeypot that rickrolls in ASCII.

The interesting question is whether the joke has a use. It does, but a narrow one, and the narrowness is the point of this paper. Most honeypots try to keep an intruder engaged with a plausibly interactive environment, because the engagement is the measurement: the longer the fake shell runs, the more commands, uploads, and credentials it records Ramsbrock et al. (2007); Owens and Matthews (2008). That design has a confound that is rarely stated. In a fake shell, the session duration is chosen by the client. It is a dependent variable whose value is set by the thing being studied, so a comparison of session durations across honeypots or across time is a comparison of attacker populations as much as of honeypot designs.

A responder whose content and duration are fixed by the server changes that. The server contributes a constant to the session duration; the client's behaviour is the only remaining source of variance. If clients close early, they do so at a measurable frame. If they sit through it, the duration is known in advance. The quantity "how long did the client stay" stops being a property of the attacker's patience and becomes a readout of the client's response to a non-shell protocol endpoint. We call this *session hold as a measurement instrument*. It is a hypothesis about experimental design, not a security benefit, and we have not yet run the experiment that would test it.

1.1 Contributions

C1. A description of what the system actually does (§3, §4). We read the source rather than the README and report the resulting behaviour: which authentication callbacks

accept, what is and is not written to disk, how the playback task is spawned, what happens when the client disconnects mid-animation, and where the implementation departs from its own documentation.

C2. A measured envelope for the responder (§5). Frame count, per-frame payload, total payload, protocol headroom, nominal timing, memory cost of the shared frame set, cost of the per-connection copy, and the CPU cost of the per-frame encoding expression. Each number is produced by a script in the artifact; none is typed by hand. Three figures and one table are generated from the same JSON.

C3. A pre-registered evaluation protocol for session hold (§6). We state the hypotheses, the A/B design, the instrumentation the current build lacks, the metrics, and the survival analysis that suits a responder with a fixed maximum duration. The protocol is executable in the narrow sense that the analysis script exists and runs; the results are absent because the data does not exist.

C4. An ethics and legal analysis specific to this artifact (§7). A credential-accepting responder records third parties by design. We state the authorisation assumptions, what the code stores, what the dashboard discloses, what the retention behaviour is, and which of these we consider liabilities rather than features.

1.2 Non-goals

We do not claim that this honeypot detects attackers better than a fake shell, that it deters anyone, that it traps or occupies an adversary to the adversary's detriment, or that it produces intelligence. We report no captured credentials, because we hold none: the repository contains no session logs, and no deployment data was collected for this paper. We do not claim that accepting every credential is novel — it is the documented behaviour of every low-interaction SSH honeypot since at least 2004 Provos (2004) — and we do not claim that an ASCII animation is a security mechanism. We do not run the system end to end in this paper, so no number here includes SSH framing, encryption, congestion control, or a real client's reaction time. Section 8 lists what remains unmeasured in more detail.

2 Background

2.1 Interaction levels and what each one buys

Honeypots are conventionally ranked by interaction. A low-interaction honeypot emulates enough of a protocol to elicit a payload; a high-interaction one exposes a real system and records everything Spitzner (2002); Provos and Holz (2007). HoneyD emulated network stacks and services Provos (2004); Potemkin built many virtual honeypots from one physical host to address the fidelity/scalability trade-off Vrable et al. (2005). For SSH specifically, the medium-interaction design won: Kippo Tamminen and its successor Cowrie Oosterhof implement a fake shell in user space, accept credentials, emulate a filesystem, and log the session as text. That design is popular because it captures the thing operators want — commands, uploaded files, attempted passwords — without the risk of a real shell.

The measurement literature built on that design. Owens and Matthews collected brute-force attempts and characterised the password lists and client behaviour Owens and Matthews (2008); Ramsbrock et al. profiled what attackers do after a successful SSH compromise Ramsbrock et al. (2007); Mirai later showed how commodity SSH/Telnet brute force MITRE scales into a botnet Antonakakis et al. (2017). In all of this work the honeypot is an instrument whose fidelity determines what can be observed, and the fake shell is the instrument of choice because it maximises observable behaviour per connection.

2.2 The confound in session duration

Session duration is a standard engagement metric, and it is almost always reported as a property of the attacker population. But duration is jointly produced: the server offers an environment, the client decides when to leave. When the environment is a shell, the client's decision depends on its own goals, its scripts, and its timeouts, and the server's contribution is essentially unbounded. When the environment is a fixed animation, the server's contribution is a constant that can be computed from the asset, and the client's contribution is visible as early termination. The variance decomposition changes, not the underlying population. This is the entire motivation for the design, and it is weaker than it sounds: a fixed responder also removes the very behaviour that makes a fake shell informative, which is why we present it as a complement and not a replacement.

2.3 Protocol quantities that bound a responder

Three SSH facts matter for the numbers in §5. The protocol separates a transport layer, an authentication layer, and a connection layer, and only the last of these carries channels Ylonen and Lonvick (2006a,b). First, the transport packet has a maximum size; RFC 4253 requires implementations to support at least 32,768 bytes and permits up to 35,000 Ylonen and Lonvick (2006c). Second, the connection protocol is window-based: a channel has an initial window and a maximum packet size chosen by the receiver, and the sender must stop when the window is exhausted until the receiver grants more Ylonen and Lonvick (2006d). Third, the transport is reliable and ordered Eddy (2022), so a server that writes faster than the client reads accumulates either window pressure or kernel buffer, depending on the implementation.

A responder that emits small frames slowly never approaches either limit. A responder that emits one large frame approaches both. Which of these a given deployment does is an empirical property of the asset and the pacing constant, and §5 shows that the committed asset is at one extreme and a Windows checkout of the same asset is at the other.

2.4 Session hold and concurrency

The operational meaning of holding a session is given by Little's law Little (1961): for a system in steady state, the mean number of sessions in progress L equals the arrival rate λ times the mean time in system W. A responder that holds every session for a fixed 75.4 s therefore fixes $L = 75.4\lambda$, whatever the clients do, whereas a fake shell's L depends on how long attackers choose to stay. This is a statement about capacity, not about security: a fixed hold makes the server's load predictable, and it also makes the server the party that decides how much of its own memory is committed per connection. Section 7 takes up the ethical reading of the same equation.

3 Design

3.1 Process and connection model

The binary is a single process with two listeners: an SSH server on `0.0.0.0:22` and an HTTP dashboard on `0.0.0.0:763`. Both run as Tokio tasks, and the process ends when either one does. An outer supervision loop retries up to ten times with a two-second delay, but it only catches a panicked task: a task that returns an error — a bind failure, for instance — is logged and ends the process without a restart. Frames are loaded once at startup and held in a shared `Arc<SharedState>`.

Each inbound TCP connection gets its own handler object, which is created before the SSH handshake begins and stores the peer address and a synthetic connection identifier of the form `CONN_<epoch-ms>_<9 random digits>`. Figure 1 shows the resulting exchange. The server presents the identification string `SSH-2.0-OpenSSH_8.9p1`, offers every authentication method, and accepts the first attempt of any kind. After authentication the client may open a session channel, optionally request a PTY, and request a shell or an exec. Either request spawns a playback task and returns immediately, so the SSH session stays open while the animation runs.

3.2 Authentication policy

Table 1 lists the four authentication callbacks. All four accept. The password callback discards the password (the parameter is named `_password` and is not referenced), and the attack record it writes carries an empty detail field. A public-key attempt is accepted at the offer stage, before any signature is verified, so the server never validates a key.

Two configuration values in the server setup are structurally dead. The authentication rejection time (2 s) applies only to rejections, of which there are none, and the maximum auth attempt count (20) never binds for the same reason. A study that wanted to manipulate exposure by tuning these knobs could not do so in this build.

The server also sets an inactivity timeout of 120 s. A full play-through takes 72.4 s of pacing plus a 3 s closing delay (Section 5), so it fits inside the timeout with margin; whether a slow client that stops reading survives the timeout depends on window-adjustment traffic, which we did not measure.

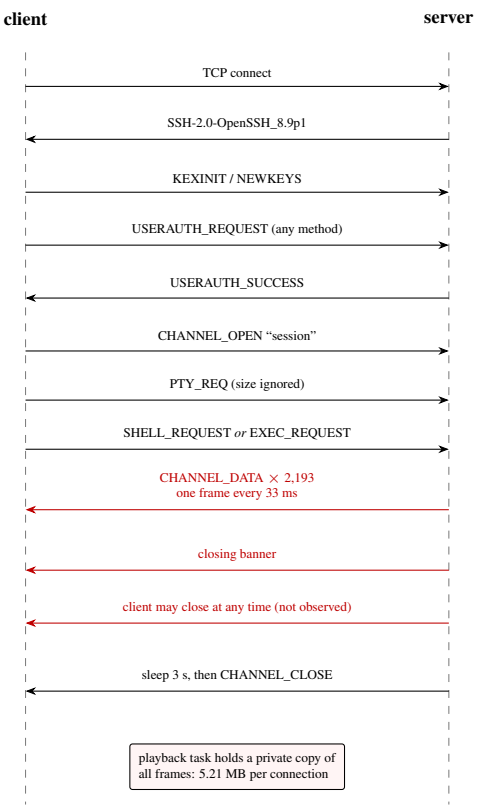


Figure 1: One session. Authentication always succeeds; the PTY request is accepted but the window size is discarded, so the frame geometry is fixed. The playback task writes 2,193 frames at 33 ms intervals and closes three seconds after the last one. The server never reads channel data, and it does not implement the channel-close callbacks, so a client that leaves early is indistinguishable from one that watched to the end until the write loop finishes.

3.3 The playback engine

When a shell or exec request arrives, the handler deep-copies the frame vector, spawns a task, and returns. An exec request first receives a single line of text ("Interactive shell only. Starting animation…"), which is the only sentence the server ever writes about itself during the session. The task then emits a clear-screen and hide-cursor sequence, then loops over the frames. For each frame it converts line endings to CRLF, prefixes a clear-screen, appends a colour-coded caption of the form `Frame  i/N`, writes the result to the channel, adds the byte count to a process-wide counter, logs progress every hundredth frame, and sleeps 33 ms. Figure 2 shows the layout. After the last frame it restores the cursor, writes three closing messages (one of which advertises the author's video channel), sleeps three seconds, closes the channel, and removes the connection from the active set.

Three properties of this loop matter downstream. The write

Table 1: Authentication callbacks. Every method is accepted on first use. The right column is what the attack log records.

Callback	Result	Logged
`auth_none`	accept	user, method “none”
`auth_password`	accept	user, method “password”
`auth_publickey_offered`	accept	user, method “publickey”
`auth_publickey`	accept	nothing (no log call)

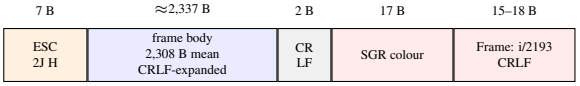


Figure 2: Per-frame payload. The fixed ANSI parts total 26 bytes, the caption 15–18 bytes, and CRLF expansion adds 28.55 bytes on average, so the mean payload is 2,380 B against a 2,308 B frame body.

is awaited before the sleep, so the per-frame period is 33 ms *plus* whatever the write costs; on a congested or window-limited connection the animation runs longer than nominal. Write errors are discarded, so a client that disconnects does not stop the loop: the task keeps sleeping and iterating over the remaining frames, and the connection is only removed from the active set when the loop finishes. And because the frame vector is copied per connection rather than shared behind an `Arc`, the cost of a session scales with the size of the animation.

3.4 Logging and the dashboard

Two JSONL files are appended to on disk, and every entry is also pushed to an in-memory vector that the dashboard serves. Table 2 lists the fields. The attack record has a fixed `success: true` and a fixed `attack_type: "ssh_brute_force"`, and its `user_agent` field is the constant string `ssh_client`, not an observed value. The country field comes from a third-party HTTP API queried with the peer address. The connection record carries an event name, one of `connection_established`, `ascii_animation_started`, `animation_progress` (every 100 frames), `ascii_animation_completed`, and `client_disconnected`.

That last event is misnamed: it is written by the server’s own playback task after it closes the channel, not by an observation of the client leaving. There is no handler for channel EOF or channel close, so a client-initiated disconnect is never recorded as such; it merely causes writes to fail, which the loop ignores.

The dashboard is an axum server that serves one HTML page and one JSON endpoint. The endpoint returns the last 100 in-memory entries, three pie aggregates, the attack counter, the active-connection count, the accumulated payload byte count, and the geolocation cache size. The vector itself is never trimmed, the JSONL files are never rotated, and the HTTP listener has no authentication.

Table 2: What is recorded, and what is not. “In memory” entries are lost on restart; “on disk” entries are appended without rotation.

Recorded	Not recorded
peer IP address	password (discarded in the callback)
username string	client identification string
auth method name	key material or key type
timestamps (RFC 3339, UTC)	commands (nothing is read)
event names around playback	client-initiated close time
payload bytes (global sum)	per-session byte count
country (third-party API)	PTY size, terminal type

3.5 What the responder can and cannot observe

The honest inventory is short. FAKE SSH PLAY observes a TCP connection, a source address, a username string, the name of the authentication method that was used, a wall-clock timestamp, and how far its own playback loop got before the process or the connection ended. It does not observe a password, a key, a command, a file, a client version, a keystroke, or a reaction. It cannot distinguish a scanner from a person, a botnet node from a proxy, or a researcher from an attacker. Nothing in the artifact supports attribution, and we make none.

4 Implementation

4.1 Codebase

The Rust crate is a binary with seven source files totalling 1,216 lines, plus a 44-line integration test and a 115-line HTML dashboard. It depends on fifteen crates directly, of which the load-bearing ones are `russh` 0.45 rus for SSH, Tokio tok for the runtime, axum 0.8 axu for the dashboard, and `reqwest` for the geolocation lookup. There is no library target, no configuration file, and no environment-variable override: the ports are compile-time constants (22 and 763), so a non-root test requires editing the source.

The test suite contains sixteen test functions: five for the frame parser, eight for the shared state and logging, two for the geolocation cache, and one integration test that starts the real binary in an empty directory and asserts that it exits nonzero with a “host key not found” message. The SSH path, the dashboard, the playback loop, and the configuration module have no tests. The repository’s own `TESTING.md` reports the same 15+1 split, and we recounted it rather than copying the number.

4.2 Frame assets and the converter

Two frame formats are supported, distinguished by their first marker. The loader first looks for `--FRAME_SEPARATOR--` and splits on that marker followed by a newline; failing that, it looks for `=== FRAME` headers and strips border lines. The committed asset, `old-js/badapple_ascii.txt`, is the first format.

That asset is a 5,399,021-byte text file with 70,150 lines, a five-line header declaring 2,192 frames at 80×30 columns and 10 FPS, and 2,193 separator lines. Its provenance is the JavaScript version of the project: the `old-js` directory contains the Node.js SSH server that consumed it and a separate ASCII banner file. The Rust-era converter, `mp4_to_txt.py`, is a different pipeline: it shells out to `ffprobe` and `ffmpeg`, decodes raw RGB at 30 FPS, downsamples to 100 columns, maps luminance onto a ten-character ramp, and writes the `=== FRAME NNNNN ===` format with 100-column borders. Its input and output paths are absolute paths on the author's machine, and neither the MP4 nor the 100-column output is committed.

This mismatch is the single most important scoping fact in the paper. The numbers in §5 describe the *committed* asset, which is the 80×30, 10 FPS, JavaScript-era file, not the 100-column file the converter would produce. A reader who regenerates the asset from a video will get different numbers, and we do not extrapolate to them.

4.3 Loader behaviour

Three behaviours of the loader are visible in the committed data and worth stating precisely, because they change what a client receives.

The header is a frame. The split produces a first chunk holding the five comment lines, and the filter only removes empty chunks. The header is therefore frame 1 of 2,193, and the client sees five comment lines before the first frame of the animation. The asset declares 2,192 frames; the loader yields 2,193.

Each frame carries its own caption. Every frame body begins with a line `FRAME_NNNN` emitted by the converter. The server then appends its own `Frame: i/2193` caption, so each frame displays both. Because the caption makes every frame textually unique, deduplication by content is impossible as committed: the file has 2,193 unique frames but only 2,143 unique art frames once the caption line is stripped. Sixteen frames at the start are blank (a ten-byte body), and thirty-nine blank frames occur in total.

Line endings decide the frame count. The loader splits on a literal LF-terminated marker. The committed blob uses LF, but the repository has no `.gitattributes` and Windows checkouts are commonly configured with `core.autocrlf=true`, which rewrites the file to CRLF. On such a checkout the marker never matches, the whole file becomes a single chunk, and the loader returns exactly one frame containing the entire animation, marker lines and all. Section 5 measures that case rather than speculating about it.

4.4 The legacy implementation

The JavaScript version that precedes the Rust one is a different system with the same name: it uses `ssh2` and `express`, accepts every credential, and streams a base64-encoded MP3 in 512-byte chunks with a progress prompt, alongside an ASCII banner. Its README claims a 312 KB file and about thirty seconds of transfer. We did not measure it; it is cited here only because it explains the provenance of the committed asset and because it shows that the project's first answer to "what should a fake SSH session contain" was audio, and its second was video. Neither is a shell.

4.5 Build and run

The binary refuses to start without a private host key and a frame file at fixed relative paths, and it binds port 22, so the documented run requires root and a key generated out of band. Neither prerequisite is committed. The integration test exercises only the missing-key path, which is the one path a fresh checkout can reach without preparation. In other words, the repository is runnable but not reproducible from a clean checkout without two manual steps that the README documents but does not automate.

5 Measured properties

5.1 What is measured, and how

All numbers in this section are produced by three artifacts. `scripts/measure_frames.py` parses the committed asset with the exact semantics of the Rust loader and computes frame sizes, payload sizes, pacing arithmetic, allocation accounting, and the channel-window comparisons; it writes `scripts/measured.json`. `scripts/frame_probe.rs` is a std-only Rust program, compiled with the same toolchain, that reimplements the loader and the per-frame encoding expression, and samples its own working set through the operating system; it writes `scripts/probe.json`. The figures are generated from those two files by `scripts/make_figures.py`.

We are explicit about what these measurements are not. They are not a measurement of an SSH session: no client was connected, no packet was encrypted, and no timer was started. The payload figures are the byte counts that the code would hand to the channel, before SSH framing, padding, and the MAC. The timing figures are nominal, derived from the 33 ms constant. The CPU figure is an isolated reimplementation of two expressions, not a profile of the server. The memory figure is the working set of a probe process, not of the honeypot under load. Each of these gaps is listed again in §8.

5.2 The asset and the loader

Table 3 gives the counted properties. The committed blob is 5,399,021 bytes; the Windows working tree is 70,150 bytes larger, exactly one byte per line, because of CRLF rewriting. The loader yields 2,193 frames from the LF blob, the first of which is the 102-byte header. Frame bodies total 5,061,764

Table 3: Counted properties of the committed asset and of the Rust loader. Every value is a field in `scripts/measured.json`, produced by `scripts/measure_frames.py` from the asset itself.

Property	Value
committed blob (LF)	5,399,021 B
Windows working tree (CRLF)	5,469,171 B
lines	70,150
separator occurrences	2,193
frames declared in header	2,192
frames returned by loader	2,193
first frame (header)	102 B, 5 lines
frame body total	5,061,764 B
frame body median / mean	2,432 / 2,308.15 B
frame body min / max	10 / 2,440 B
unique art frames (caption stripped)	2,143 of 2,193
blank art frames	39 (16 leading)
channel payload total	5,219,755 B
payload median / mean	2,505 / 2,380.19 B
payload min / max	51 / 2,514 B
mean per-frame overhead	72.04 B
nominal duration at 33 ms	72.369 s
authored duration at 10 FPS	219.2 s
speed relative to authored	3.029×
mean payload rate	72,127 B/s

bytes, with a median of 2,432 and a mean of 2,308; the distribution is spiky (Figure 3a), with forty near-empty frames and a dense cluster between roughly 1,200 and 2,514 bytes.

5.3 Payload envelope and protocol headroom

The largest payload is 2,514 bytes, which is 7.18% of the 35,000-byte maximum packet in RFC 4253 and 7.67% of the common 32,768-byte packet. No frame exceeds either limit, and no frame is large enough to be fragmented by a reasonably sized maximum packet size. At a 64 KiB channel window — the smallest value we consider — a window holds 27.5 frames, or 0.91 s of animation; at a 2 MiB window it holds 881 frames, or 29.1 s. Mean window fill is 3.63% at 64 KiB and 0.11% at 2 MiB. In other words, the pacer, not the protocol, is the bottleneck by two orders of magnitude: the server emits 72 kB/s while a single window can absorb half a minute of output.

5.4 Pacing and CPU

The pacing constant is 33 ms, which the source comments call "30 FPS". The asset was authored at 10 FPS and declares a duration of 219.2 s, so a full play-through at 33 ms is 72.369 s: the committed animation plays 3.03 times faster than its source timing (Figure 5b). Whether that is a bug or an intentional speed-up is not recorded anywhere in the repository; the constant and the asset header simply disagree.

The CPU cost of the per-frame encoding expression — the CRLF replacement, the `format!` that builds the payload, and the copy into an owned byte buffer — is small enough that the timer dominates. Over 2,193 frames the probe measured a median of 1.00 μs, a mean of 1.07 μs, a 95th percentile of 1.30 μs, and a maximum of 15 μs, for 2.68 ms of total encoding against 72.4 s of playback. The mean is 0.0032% of the 33 ms budget (Figure 5a). This says nothing about the cost of encryption, which the probe does not perform and which the repository does not benchmark.

5.5 Memory, and the cost of one session

Holding the animation costs, by allocation accounting, 5,061,764 bytes of frame text plus 24 bytes of string header and 24 bytes of vector slot per frame, or 5,167,028 bytes in steady state, with a peak of 10,566,049 bytes during loading while both the file contents and the parsed copies are alive. The probe measured a working-set delta of 4,456,448 bytes after loading, from a baseline of 10,596,352 bytes; the difference from the accounting is allocator slack and untouched pages.

The more consequential number is the per-connection copy. Because the handler clones the vector instead of sharing it, every session that reaches a shell request takes another copy. The probe measured 1.516 ms and 5,208,320 bytes of working set per clone, and 83,333,120 bytes for sixteen held clones, which is linear. Twenty concurrent sessions therefore commit about 104 MB to frame copies alone, and the server has no connection limit, no per-IP limit, and no limit on session channels per connection. This is not a vulnerability report — we did not test it as one — but it is the direct consequence of a line of code, and it means the operator pays for session hold in memory as well as in bandwidth.

5.6 The CRLF collapse

On a CRLF checkout the loader returns one frame of 5,539,356 bytes. The playback loop then writes that single payload, sleeps 33 ms, waits three seconds, and closes. The mean payload rate becomes 168 MB/s, which is meaningless as a rate because there is one frame, but the important consequence is structural: a 5.5 MB write exceeds the maximum packet by two orders of magnitude and exceeds a 64 KiB window by a factor of 84. The server must wait for the client to drain the channel before the write completes, so the session duration becomes a function of the client's receive rate rather than of the pacing constant. The encoding expression also becomes visible in the budget: 6.14 ms of work in a 33 ms frame, 18.6% of the timer.

We report this as a defect, not as a configuration option. It is silent (the loader logs "1 frames"), it is platform-dependent, and it inverts the property that the rest of the paper is about.

5.7 What these numbers do not establish

They do not establish that the animation holds anyone's attention, that clients stay for 72 s, or that a fake shell would produce different durations. They do not establish that the payload is delivered as computed: SSH framing adds packet headers, padding, and a MAC, and a real client's window adjustments determine whether the 33 ms timer or the network sets the pace.

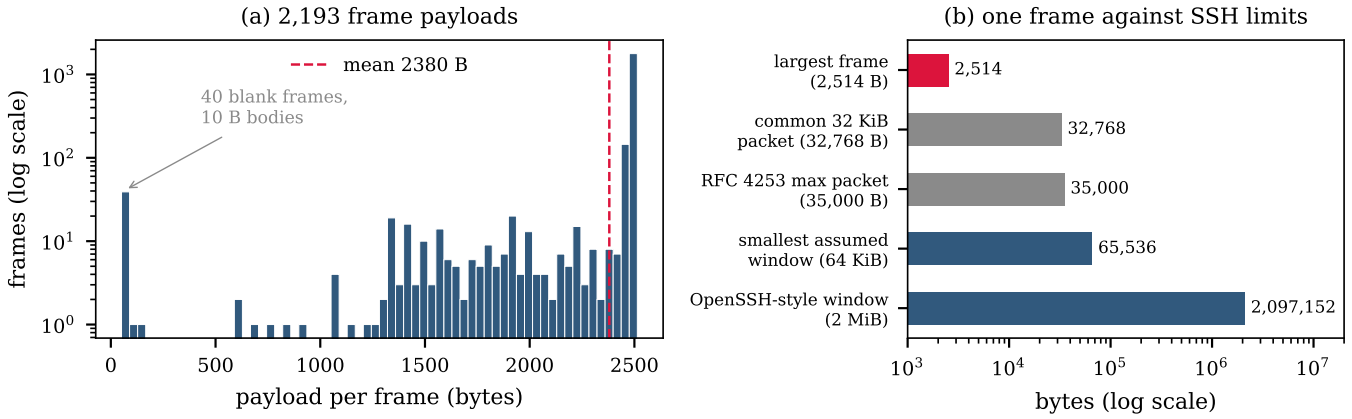


Figure 3: (a) Distribution of the 2,193 per-frame payloads on a logarithmic count axis: forty near-empty frames, then a dense band up to 2,514 bytes. (b) The largest single frame against the protocol limits and the channel windows considered in the text, on a logarithmic byte axis. The frame is two orders of magnitude below a 2 MiB window.

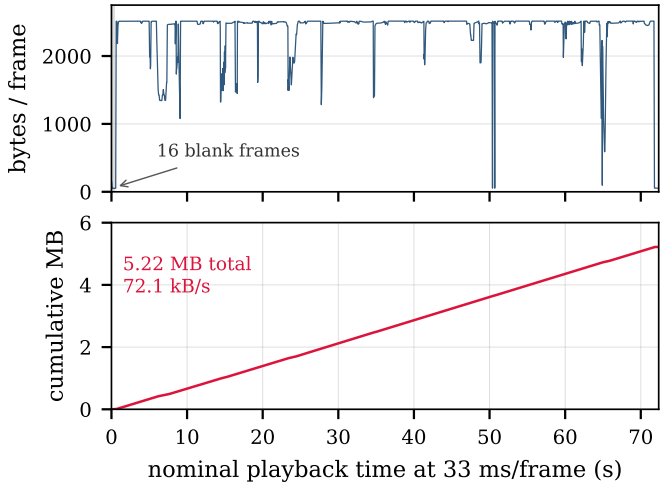


Figure 4: Top: per-frame payload against nominal playback time, showing the blank lead-in and the drops where the source frame is nearly empty. Bottom: cumulative payload, ending at 5.22 MB after 72.4 s, a mean rate of 72.1 kB/s.

They do not establish the CPU cost of a session, because encryption, task scheduling, and allocation are excluded. And they say nothing about the only quantity that matters for the hypothesis in §6: the distribution of client departures.

6 Evaluation protocol: session hold

The experiment we have not run is a controlled comparison between this responder and an interactive fake shell, on the same host, against the same client population, measuring the same quantities. This section is the protocol. It contains no results: the analysis script that would produce them exists and is exercised only on synthetic input.

6.1 Hypotheses

H1 (hold). A responder with a fixed maximum duration produces a different distribution of session duration than an interactive fake shell, after conditioning on the population of clients that authenticate.

H2 (input). A responder that never emits a prompt and never reads a command elicits fewer post-authentication bytes from clients than a fake shell, and the bytes it does elicit are dominated by clients that write before reading.

H3 (exit). The exit mode differs: in the animation arm, exits are either server-initiated at the fixed end or client-initiated at a frame that can be recorded; in the shell arm, exits are overwhelmingly client-initiated and unpredictable.

H1 is the interesting one and the easiest to get wrong. A difference in duration could be produced by the population (different clients arrive on different days), by the port or address (a second listener is discovered at a different time), by detection (a client that recognises the animation as fake leaves early), or by the responder itself. The design below tries to separate the last two from the first two; it cannot separate detection from boredom.

6.2 Design

Two arms run simultaneously on two ports of the same host: the animation arm (this system, unmodified except for the instrumentation in §6.3) and a shell arm (Cowrie Oosterhof or an equivalent medium-interaction SSH honeypot, in its default configuration). Assignment is by a salted hash of the source address, so that a given scanner always lands in the same arm and a returning client contributes to the same condition. The cost of that choice is that arms are not exchangeable across address blocks; the benefit is that a within-client comparison is possible for repeat visitors. A time-block randomisation (alternate arms hourly) is the alternative, and it trades the opposite way: it controls for drift in the population but gives repeat visitors different treatments.

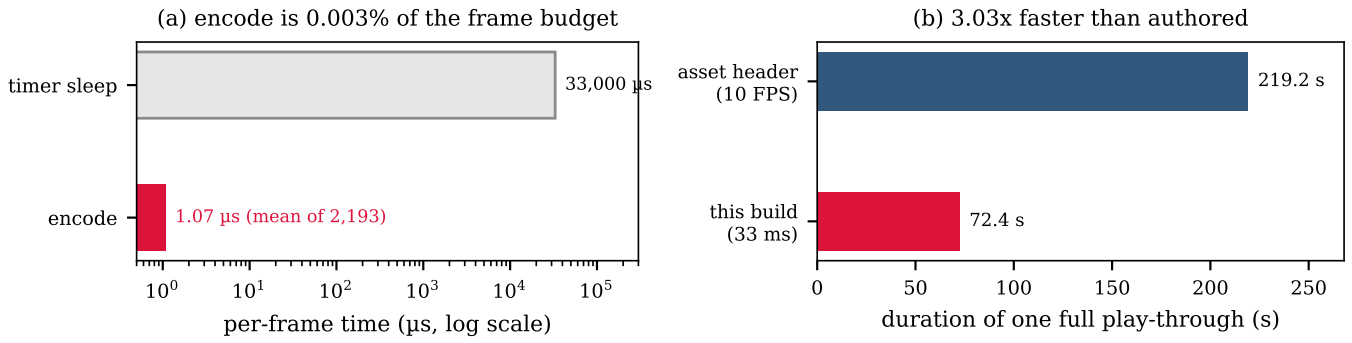


Figure 5: (a) Per-frame time on a logarithmic axis: the encoding expression averages 1.07 μs against a 33 ms timer budget, i.e. four and a half orders of magnitude below it. (b) A full play-through at 33 ms takes 72.4 s, while the asset header specifies 10 FPS and 219.2 s, a 3.03$\times$ speed-up.

The unit of analysis is the session, defined as one TCP connection that reaches authentication success. Sessions that never authenticate are counted separately and are not part of the comparison, because the arms differ in their authentication surface by construction.

Both arms must be deployed for long enough to see the diurnal cycle at least three times. Neither the deployment window nor the expected n per arm is stated here, because both require a pilot that has not been run.

6.3 Instrumentation the current build lacks

The committed build cannot run this experiment. It does not record when a client disconnects, it does not read channel data, and its active-connection counter is decremented only when its own loop finishes. The instrumented variant needs six changes, all inside `Src/Control/ssh.rs`:

I1. Implement `channel_eof` and `channel_close` and record a monotonic timestamp for the first of them. This is the client-observed end of the session and the censoring variable for H1.

I2. Implement a `data` handler that counts bytes and newlines received per channel. It should not store payload content; the counts are what H2 needs, and storing content would create the retention problem discussed in §7.

I3. Write one JSONL record per session, with the schema in `scripts/analyze_sessions.py`: authentication time, shell-request time, client-close time (or null), server-close time, frames written, bytes to and from the client, and the PTY size if one was requested.

I4. Record the frame index at client close. This is the animation arm's analogue of a command count: the point at which a client gave up.

I5. Move the geolocation lookup out of the authentication path, or cache it before deployment. Today the first connection from an uncached address waits up to three seconds for an HTTP call inside `auth_password`, which would contaminate any measurement of time-to-auth.

Table 4: Protocol metrics. The last column is where the value would come from once the instrumented build exists.

Metric	Summary	Source
session hold	KM median, quartiles, log-rank	I1 records
completion	fraction reaching frame 2,193	I4 records
frames before exit	median, distribution	I4 records
bytes from client	median, zero fraction	I2 records
exit mode	client vs server closed	I1 records
concurrency	$L = \lambda W$ Little (1961)	arrivals + hold
server cost	CPU-seconds and MB per session	probe + profiling

I6. Make the listen port configurable. It is a compile-time constant today, so two arms on one host require a rebuild or two addresses.

None of these changes alters the animation. They alter only what is recorded, which is exactly the property a honeypot study needs and the committed build does not have.

6.4 Metrics and analysis

Table 4 lists the quantities and how each is summarised. Because the animation arm has a hard maximum duration, durations are right-censored at that maximum: a client that stays to the end contributes a censored observation, not a completed one. The appropriate summary is the Kaplan–Meier estimator Kaplan and Meier (1958), and the appropriate comparison is the log-rank test, with the arm as the grouping variable and censoring at the server's close.

`scripts/analyze_sessions.py` implements the first five rows: it reads the per-session JSONL, prints per-arm counts, median hold, completion fraction, median bytes from the client and the fraction of client-initiated closes, and runs a two-sample log-rank test. It refuses to fabricate: with an empty input file it prints the empty table and the sentence "no result". It also has a self-test mode that runs the whole path on synthetic sessions and labels the output as synthetic.

6.5 Power and stopping rule

The between-arm hazard ratio, the deployment duration that would give 80% power at $\alpha = 0.05$, and the resulting sample size are left unspecified. Without a pilot we will not state a sample size: a number invented here would be indistinguishable from a result.

6.6 Threats to validity

The arms are not the same service. A shell honeypot advertises a shell; this one advertises an animation. Any difference in duration may be a difference in what the client wanted, which is the point, but it is also a difference in what the client could do. H2 is therefore descriptive, not causal.

Population drift. Scanning traffic follows diurnal and weekly cycles and changes after disclosures. Hash assignment controls for which clients are compared but not for when they arrive.

Detection. A client that recognises the responder as fake may leave early, which would look like a short hold. We have no way to measure detection and will not claim to.

Instrumentation effects. Counting received bytes and recording per-frame exits adds work to the session. It is small relative to 33 ms frames, but it is not zero, and it is present only in the instrumented build.

Ethics approval. The protocol requires collecting third-party session metadata over weeks. That requires review before deployment, not after (§7).

7 Ethics and legal framing

7.1 Authorisation and jurisdiction

A honeypot accepts connections from parties who did not consent to be studied. The operator's authorisation therefore has to come from somewhere other than the clients. In most jurisdictions the defensible bases are: the operator owns or is authorised to expose the address and port; the deployment is covered by an institutional review or a terms-of-service clause; and the collected data is limited to what is necessary for the stated purpose. This paper does not give legal advice and the repository documents none of these bases. A deployment of this code as committed would be an unreviewed collection of third-party data.

The data is personal data in the ordinary sense. An IP address and a username are identifiers under the GDPR gdp (2016) and under China's Personal Information Protection Law pip (2021), and the geolocation lookup sends the address to a third-party service, which makes that service a processor whose terms the operator has not necessarily accepted. Nothing in the code records a lawful basis, a purpose limitation, or a retention period.

7.2 Data minimisation and retention

The code does one thing right and several things wrong. It discards the password: the callback takes `_password` and never reads it, and the attack record's detail field is empty. Not storing credentials is a genuine minimisation choice, and it means this artifact cannot leak a password list.

against that: the attack records, connection records, and in-memory vector grow without bound and without rotation; the in-memory vector is never trimmed even though only the last hundred entries are served; the JSONL files have no retention policy, no access control, and no encryption at rest; and the geolocation cache is capped at five hundred addresses with a crude eviction but never expires an entry by time. A long-running deployment accumulates an unbounded, unencrypted record of third-party addresses and usernames. That is a liability, not a feature, and it is the opposite of what a study protocol would require.

7.3 Disclosure

The dashboard is the sharpest problem. It listens on all interfaces over plain HTTP with no authentication, and its timeline exposes source addresses, countries, event names, and session identifiers to anyone who can reach port 763. A public honeypot that publishes its own capture list is publishing other people's addresses. The dashboard also renders values with `innerHTML`, and one of those values — the country name — is a string returned by a third-party HTTP API, so a compromised or malicious geolocation response would execute script in the dashboard's origin. The repository's test documentation claims the front end renders text safely; the code path does not support that claim.

On the other side of the ledger, the closing message tells the client, in Chinese and English, that it has been watching an SSH ASCII honeypot and where to find the author. That is unusual and, in our view, good: the responder does not pretend to be a production system to the person on the other end. It does not undo the recording, but it removes the pretense.

7.4 Engagement ethics: whose time, whose risk

Prolonging an attacker's session is sometimes described as a defensive win. We do not make that claim, for two reasons.

First, the time is mostly not a person's. Automated SSH clients do not experience 72 seconds; they experience a socket that stays open, and their timeouts are configured in advance. A human operator who watches the animation is choosing to, and we have no evidence about how often that happens — the instrument to find out does not exist in this build (H1/H3).

Second, holding a connection has a cost on the defender's side. Each session commits 5.21 MB of frame copies and a live socket for at least 75 s, and the server imposes no limit on either. A responder that advertises itself on port 22 and accepts every credential is a machine that will spend memory and bandwidth on request. If the goal were to consume an adversary's resources, this design would first consume the operator's; that is the trade-off, stated plainly, and it is why we

frame the fixed duration as a measurement convenience rather than as a defence.

7.5 Art, attribution, and licence

The animation is a derived work. The song and the 2009 shadow-art fan video bad are third-party works, and the committed asset is an ASCII rendering of that video produced by an uncommitted pipeline. The repository declares MIT in its README and in `Cargo.toml` but ships an Apache-2.0 licence file, and neither licence resolves the status of the artwork. We flag this rather than resolve it: a reader who wants to reuse the frame asset should treat its provenance as unresolved.

The closing message also advertises the author's video channel. That is the honest motive for the project — it is partly a broadcast channel dressed as a honeypot — and we prefer to say so. It does not affect the measurements, but it does affect how the artifact should be described.

8 Limitations

- **No engagement data.** The repository contains no captured sessions, and none were collected for this paper. Every claim about attacker behaviour is a hypothesis.
- **No end-to-end run.** No SSH client was connected. Wire sizes exclude framing, padding, MAC, and retransmission; durations are nominal.
- **No encryption cost.** The 1.07 μs per-frame figure covers encoding only.
- **Two defects in the loader.** The header is emitted as frame 1, and a CRLF checkout collapses the animation to one 5.5 MB frame. Both are measured, neither is fixed.
- **The PTY is ignored.** `pty_request` accepts and discards the window size, so an 80-column frame is sent to a 40-column terminal and the client's terminal wraps it. The visual result on small terminals is not measured.
- **Nothing is read from the client.** No command, keystroke, or byte sent after authentication is observed, so the build cannot measure input volume (H2) at all.
- **Client close is not observed.** There is no EOF or close handler; `client_disconnected` is written by the server after its own loop ends.
- **Active connections are overcounted.** A client that leaves early remains in the active set until the loop finishes, up to the full 72.4 s.
- **Unbounded collections.** The in-memory log and both JSONL files grow without limit; the dashboard has no authentication; the geolocation cache is capped but the log is not.
- **The restart loop does not cover errors.** It restarts a panicked task, not a failed bind; the README's "auto-restarts on crash" overstates it.
- **Resource amplification.** Each session copies 5.21 MB; channel count per connection is unlimited; there is no connection or rate limit.
- **Over-broad private-range test.** The geolocation short-circuit treats every address beginning `172.` as private, which mislabels public addresses in that range.
- **Asset/converter mismatch.** The committed asset is 80×30 at 10 FPS; the converter targets 100 columns at 30 FPS; the produced asset is not committed. Our numbers describe the former.
- **Test coverage.** Sixteen test functions cover parsing, state, and geolocation. The SSH path, playback loop, dashboard, and configuration have none.
- **Single author, nine commits.** There is no CI, no benchmark, and no external review of the code.
- **Licence inconsistency.** README and manifest say MIT; the licence file is Apache-2.0; the artwork's status is unresolved.

9 Related work

SSH honeypots. Kippo Tamminen and Cowrie Oosterhof are the reference medium-interaction designs: accept credentials, emulate a shell, log everything. FAKE SSH PLAY keeps the acceptance and drops the emulation, which places it below Kippo in interaction and makes it useless for the questions Kippo was built to answer (what commands, what uploads, what passwords). Its value, if any, is as a control condition for those questions.

Low-interaction frameworks. HoneyD Provos (2004) and Potemkin Vrable et al. (2005) address scale and fidelity at the network and VM level. FAKE SSH PLAY is a single service on a single host and does not attempt either.

Tarpits. Endlessh Wellons holds a connection open by dribbling a banner forever, with the explicit goal of wasting a scanner's resources. FAKE SSH PLAY holds a connection for a bounded time with content, and does not claim to waste anyone's resources; the difference is the bound, and the bound is what makes the hold measurable.

Attacker measurement. Owens and Matthews characterised brute-force password lists Owens and Matthews (2008); Ramsbrock et al. profiled post-compromise behaviour Ramsbrock et al. (2007); Mirai showed the scale of commodity SSH brute force Antonakakis et al. (2017). All of these depend on a shell or on credential capture. A responder that offers neither can only contribute a duration and an exit frame, which is a much thinner signal and should be described as such.

Survival analysis of sessions. Right-censored durations are standard in reliability and clinical work Kaplan and Meier (1958). Applying the estimator to honeypot sessions is not new in principle, but it is the right tool for a responder with a hard maximum, and we are not aware of it being used to compare a fixed-duration responder against a shell.

Ethics of honeypot research. The Menlo Report Dittrich and Kenneally (2012a,b) sets out respect for persons, beneficence, and justice for ICT research, and its guidance on third-party data is directly applicable here: the parties being observed have not consented, the data is personal, and the burden of justification falls on the operator. Our §7 is an application of that framework to a very small system, and it concludes that the committed build would need changes before it could be deployed for research.

10 Conclusion

FAKE SSH PLAY is a joke with a measurable core. It accepts every SSH credential, plays a 2,193-frame ASCII animation at 33 ms per frame, and closes. Its measured envelope is small and fully accounted for: 5.22 MB of payload per play-through, 2.4 kB per frame, a largest frame at 7.2% of the protocol's maximum packet, 4.5 orders of magnitude of CPU headroom under the frame timer, and 5.21 MB of resident memory per concurrent session because the frame set is copied rather than shared.

The defensible idea is not the honeypot and not the animation. It is that a responder with server-determined content turns session duration into a comparable quantity: the server contributes a constant, the client's behaviour is the variance, and censoring at a known bound makes the comparison statistically tractable. That idea is a hypothesis. Testing it requires an instrumented build, two arms, weeks of traffic, and ethics review, and this paper provides the protocol, the analysis script, and an explicit list of the numbers it does not have. What it does not provide — and does not claim — is evidence that anyone stays, that anything is detected, or that the operator gains any security at all.

References

axum: an ergonomic and modular web framework for Rust. https://github.com/tokio-rs/axum. Software, version 0.8; retrieved 2026.

Bad apple!! https://en.wikipedia.org/wiki/Bad_Apple!! Reference for the 2009 shadow-art fan video and its source song; retrieved 2026.

russh: a Rust implementation of the SSH protocol. https://github.com/Eugeny/russh. Software library, version 0.45; retrieved 2026.

Tokio: an asynchronous runtime for Rust. https://tokio.rs. Software; retrieved 2026.

Regulation (EU) 2016/679 of the european parliament and of the council (general data protection regulation). https://eur-lex.europa.eu/eli/reg/2016/679/oj, 2016.

Personal information protection law of the people's republic of china. http://www.npc.gov.cn/, 2021. Adopted 20 August 2021, effective 1 November 2021.

M. Antonakakis, T. April, M. Bailey, et al. Understanding the Mirai botnet. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

D. Dittrich and E. Kenneally. The Menlo report: Ethical principles guiding information and communication technology research. Technical report, U.S. Department of Homeland Security, 2012a.

D. Dittrich and E. Kenneally. The Menlo report: Ethical principles guiding information and communication technology research. *IEEE Security & Privacy*, 10(2), 2012b.

W. M. Eddy. Transmission control protocol (TCP). RFC 9293, IETF, 2022.

E. L. Kaplan and P. Meier. Nonparametric estimation from incomplete observations. *Journal of the American Statistical Association*, 53(282):457–481, 1958.

J. D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.

MITRE. ATT&CK technique T1110: Brute force. https://attack.mitre.org/techniques/T1110/. Retrieved 2026.

M. Oosterhof. Cowrie: a medium-interaction SSH and telnet honeypot. https://github.com/cowrie/cowrie. Software; retrieved 2026.

J. Owens and J. Matthews. A study of passwords and methods used in brute-force SSH attacks. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

N. Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

N. Provos and T. Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley, Upper Saddle River, NJ, 2007.

D. Ramsbrock, R. Berthier, and M. Cukier. Profiling attacker behavior following SSH compromises. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007.

L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, Boston, MA, 2002.

U. Tamminen. Kippo: an SSH honeypot. https://github.com/desaster/kippo. Software; retrieved 2026.

M. Vrable, J. Ma, J. Chen, et al. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

C. Wellons. Endlessh: an SSH tarpit. https://github.com/skeeto/endlessh. Software; retrieved 2026.

T. Ylonen and C. Lonvick. The secure shell (SSH) protocol architecture. RFC 4251, IETF, 2006a.

T. Ylonen and C. Lonvick. The secure shell (SSH) authentication protocol. RFC 4252, IETF, 2006b.

T. Ylonen and C. Lonvick. The secure shell (SSH) transport layer protocol. RFC 4253, IETF, 2006c.

T. Ylonen and C. Lonvick. The secure shell (SSH) connection protocol. RFC 4254, IETF, 2006d.