

sol pbc

a colorado public benefit corporation

confidential processing in solstone: how your journal checks the service before it sends

a security whitepaper on measurement, verification, pinning and transparency, and on their limits

version 2.0 · state as of 18 november 2026

solpbc.org

document control

document	confidential processing in solstone: a security whitepaper
version	2.0
state as of	18 november 2026
what it describes	the service as it runs on the state-as-of date. appendix b names the code and records this version was checked against; appendix d is the changelog
contact	security@solpbc.org
published by	sol pbc, a colorado public benefit corporation · solpbc.org

this paper describes a system that changes, and we update it when the system does. a statement here is true of the state above; the code and the signed records are the authority after the state-as-of date, and where this paper and either of them disagree, they win.

contents

summary 3

1. scope and how to read this 4

2. what confidential processing is 4

3. threat model 5

4. custody: who holds which key and which data 8

5. architecture 11

6. what is measured, register by register 15

7. what the journal checks before it sends 17

8. pins, rotation and release 20

9. the transparency surface 22

10. failure behaviour 24

11. limits and open gaps 24

12. where we would most value critique 29

appendix a. glossary 30

appendix b. current values and evidence index 31

appendix c. how the statements in this paper were checked 34

appendix d. changelog 35

appendix e. related work and references 39

summary

solstone is a personal journal platform: an owner's journal lives on a device they control. **confidential processing** is an optional, paid service in which the journal sends a piece of work to an AI model that sol pbc runs on confidential hardware in Microsoft Azure: an AMD SEV-SNP virtual machine with an NVIDIA H100 GPU in confidential-computing mode. the model reads that work in the clear, inside the machine, while it answers. the journal as a whole is never sent to this service. what one piece of work needs is sent, and that can itself be sensitive.

the protection for this service is not a key we never hold. it is **the build that booted the machine, and the journal's check of it before anything is sent.** the serving image is assembled by a public recipe from pinned inputs, some of them upstream binaries we did not compile (§ 5.2), and is designed to be reproducible (our own two builds matched; no outside rebuild yet); its boot file, kernel command line and verified read-only root filesystem are bound together and measured into the machine's virtual TPM; and its expected measurements are published in a signed record and compiled into each current journal release. before sending, the journal verifies AMD's signature over the processor's report, requires the processor's firmware level, read from the chip's certificate, to be at or above a compiled-in minimum, checks that the cloud firmware that measured the boot was launched under the signer it pins, checks a fresh TPM quote over fourteen registers against those published values, verifies the GPU's own evidence against NVIDIA's roots, and confirms that the encrypted session it holds ends inside that machine. if any check fails, no credential and no content is sent, and nothing is sent anywhere else.

in one line: **the owner's journal verifies the published build before it sends.** the machine's virtual TPM signs what it booted, and the AMD processor's signed report vouches for that TPM. the journal checks those signatures and sends only to a published build it pins, which is set up to write nothing it is sent to storage and to send it nowhere else. what it holds in memory while it works, including a prompt cache kept per journal, can stay there until the engine restarts (§ 5.3).

the check covers what was loaded at boot, not what the software does afterwards. non-retention is a property of the build, not something the attestation proves. the journal can verify that the engine booted the build we published, not that we published the right one: no one outside sol pbc has yet rebuilt or audited the image. it relies on AMD, NVIDIA and Microsoft's confidential firmware, and checks some of their evidence more narrowly than it could. an owner gets the checks in the journal release they are running. the image before the serving one stays in service while owners update, and a journal accepts every image its release pins (§ 8.4). and our own transparency records for the journal have gaps. § 11 sets out the limits. § 12 asks where you think the design is weakest.

1. scope and how to read this

this paper describes one service: **confidential processing**, the optional service in which an owner's solstone journal sends work (text, images and audio for transcription) to an AI model that sol pbc runs on confidential GPU hardware in Microsoft Azure. it explains how the service is built, what the owner's journal checks before it sends anything, what those checks establish, and what they do not.

it is written for a security-literate reader who wants to judge the design for themselves. every load-bearing technical statement points at something you can check without us: a line of code at a fixed commit in one of our public repositories, or an object on our public transparency site. where a statement rests only on our own internal testing or records, the paper says so, and appendix c lists those statements.

what the paper describes, and as of when. the body describes the service as it runs on the date in the document control table: the serving image, the earlier image still in service beside it while owners update, and what a current journal release checks. it names no release numbers. appendix b gives the commit of the serving image, the journal release this version was checked against, and the values that release pins. appendix c says how each load-bearing statement was checked, and when. appendix d is the changelog: what each serving image improved, which journal releases changed what the journal checks, and which releases admit which image. when the system changes, the body is brought up to date and the change is added to appendix d. appendix e places the design beside others' published work and lists the references.

conventions.

- "we" and "sol pbc" mean the company. "the owner" is the person whose journal it is. "the journal" is the owner's solstone journal, the software and the memory it keeps, on a device the owner controls.
- "the portal" is our services portal, the site where an owner signs in and turns the service on.
- "the engine" is the confidential virtual machine that runs the model. "the serving image" is the newest disk image in service, built from a named public commit. an engine boots one published image.
- "attestation" names a mechanism: signed hardware evidence that the journal checks. it is never used here as a property of the software build. the paper says what is measured and what the journal compares, register by register, and states the conclusion that follows and the one that does not.
- PCR means a TPM platform configuration register. a "pin" is an expected value compiled into a journal release.
- a number, hash or date appears with how it was obtained, or it is not given as a number.

what is out of scope. the owner's own device and the journal at rest on it; our other services (private network, encrypted backup, solstone.me), whose content is closed to us by keys we never hold and which are not described here; and the general security of the companies whose hardware this depends on, beyond naming what we rely on them for.

2. what confidential processing is

a solstone journal can use a model in three ways. it can run a model locally on the owner's own hardware, which is the default and needs nothing from us. it can use a provider or an endpoint the owner brings. or, if the owner turns it on and pays for it, it can use **confidential processing**: an AI model sol pbc runs itself, on

confidential GPU hardware, for work that needs more capacity than the owner's machine has.

what leaves the owner's device on that third path is what one piece of work needs: the text, images and audio a model must read or transcribe. the journal as a whole, the owner's accumulated memory, never leaves. the model reads what it is sent, in the clear, inside the confidential machine, while it answers. the protection for this service is the **build that booted the machine**, and the journal's check of that build before it sends. our encrypted backup and relay are protected differently, by keys we never hold.

the machine is a confidential virtual machine in Microsoft Azure: an AMD processor with SEV-SNP memory encryption and isolation, paired with an NVIDIA H100 GPU running in confidential-computing mode. Microsoft hosts the machine. the hardware is designed to keep the host, its hypervisor and its operators out of the virtual machine's private memory; § 11 and appendix e say where published research has broken that design. Microsoft's firmware layer inside the machine, which implements its virtual TPM, is trusted (§ 11). no third-party AI provider is in the path: we run the model ourselves, an open-weights model, on our own serving stack, and both are baked into the published image.

the service is reached at one public name, `processing.solstone.app`, port 9443. each address behind that name is one engine, and the journal judges each engine by its evidence, not by its address. on the state-as-of date two engines answer: one runs the serving image and one runs the image before it (§ 8.4).

---

3. threat model

the claim this design makes. it is not that no one could ever reach what an owner sends. it is that the service is built to write none of it to storage and to send none of it elsewhere (§ 3.1). and it is that the owner's journal verifies the published build before it sends, within the limits § 11 sets out: the recipe and its pinned inputs are public (§ 5.2), anyone can recompute the measurements the image determines from the published boot file, the hardware vouches for what booted, and the journal checks that. the rest of this section reads each party against that claim: what the check lets an owner or a reviewer confirm, and where it stops.

3.1 what is being protected

1. **the content of each request and its answer** while it is in transit and while it is being processed.
2. **no lasting copy**: no prompt, image, audio, transcript or answer written to storage or sent anywhere else by the service. what the engine holds while it works stays in volatile memory and is lost when the engine restarts. that includes a prompt cache kept per journal (§ 5.3), and we state no shorter lifetime for it.
3. **the link between an owner and their use of the service**, which we keep as small as the service allows.

3.2 what the check covers, party by party

every "defended" in this table holds under four assumptions: no signing key has been extracted from the processor, the GPU or Microsoft's firmware layer; that firmware layer is honest; the engine's software has not been compromised since it booted; and the journal release the owner runs is authentic. the later rows and § 11 say what happens when one of them fails.

adversary	what they can do	how the design answers	result
a network attacker between journal and engine	read, alter, replay or redirect traffic	TLS 1.3 to a key generated inside the verified machine, bound to a fresh TPM quote and the processor's signed report; the journal refuses any connection whose evidence it cannot verify	defended
the cloud host's software and management plane: the hypervisor, the host operating system, and staff acting through them	read guest memory, inspect disks, attach devices, inject commands through the cloud's management plane	SEV-SNP keeps guest memory encrypted and integrity-protected from the host; the GPU runs in confidential-computing mode; the image carries no remote-command agent and no SSH server, its root filesystem is read-only and verified block by block, and it refuses to boot from a disk whose partition layout differs from the published one	defended within the limits of the hardware and of Microsoft's firmware layer (§ 11, items 5 to 7)
someone who can redirect the service name (a DNS compromise, or anyone controlling where journals connect) and runs a confidential VM of their own in Azure	point journals at a machine of their choosing	engine identity is the evidence, not the address: the journal admits only a machine whose signed evidence matches a build it pins, on firmware launched under the signer it pins. a copy of the serving image on a confidential VM of their own in Azure would be admitted, and would show them connection metadata but no content (§ 4.2)	content defended, subject to the firmware-signing assumption in § 11. a journal admits every image its release pins, so it is exposed to the weaknesses of the oldest of them (§ 11, item 14)
a routine sol pbc infrastructure operator: someone holding our cloud, DNS or deployment credentials. sol pbc is one person and an AI agent (§ 9.4), so this is either of them	start, stop or replace machines; change DNS; boot a different image; change the disk the machine boots from	a different boot file or root filesystem measures differently, and the journal refuses it; a changed partition layout stops the boot before anything serves (§ 5.2); there is no shell or agent to log into, so an unhealthy machine is replaced	content defended: a modified boot file, root filesystem or disk layout is refused before any content is sent. the separation of one journal's prompt cache from another's rests on our portal's answer, which an operator could change (§ 11, item 10)
sol pbc itself, acting deliberately or under legal compulsion, publishing a different build and shipping a journal release that trusts it	everything above, plus choosing what "the published build" is	the build's recipe, its pinned inputs and its measurements are public, and a new trusted build reaches owners only through a new journal release, which is normally public and recorded (§ 9.4 and appendix d list the gaps)	**not prevented. normally visible**, though nothing technically forces publication. see § 11
someone with a machine of an accepted processor family under their physical control, or holding a signing key extracted from one, who can also redirect the service name	present evidence that verifies from a machine that is not Azure's and does not run our build	none. the journal accepts a valid report from any processor of three AMD families whose firmware is at or above the minimum (§ 7), and every check after AMD's signature is over data that the processor or the firmware layer signs. the GPU's evidence names no particular machine (§ 11, item 7). nothing in the evidence says where a machine is or whose it is	**not defended.** published attacks have read and altered the memory of confidential virtual machines of this class with physical access to a server's memory bus (appendix e)

adversary	what they can do	how the design answers	result
an attacker exploiting the running software (the kernel, the model server, the transcription service, the gateway), including an owner with a valid credential who sends hostile text, images or audio	run code inside an admitted engine	the measured image limits what can be loaded; nothing is written to disk; on the serving image, the model server refuses media sent as an address (§ 5.3); the model and transcription services are confined to loopback, and the gateway's outbound access is limited (§ 11). none of that stops a compromised model server from putting one journal's content into another journal's answer: every journal shares one model server	**outside what the journal's check can see** (§ 6, § 11)
the hardware and firmware vendors: AMD, NVIDIA, and Microsoft's firmware layer that implements the virtual TPM	sign evidence for a state that is not what it claims	trusted. we pin what we can (§ 7); § 11 lists what we cannot	trusted, not defended
physical and side-channel attacks on the processor or GPU	extract secrets or data through hardware effects	out of scope. what stands between such an attacker and an engine is Microsoft's physical control of its data centres, and the journal cannot tell one of those machines from any other (see the physical-control row above)	not defended

each result is for a current journal release, and is bounded by what the check measures. § 11, item 14 and appendix d say what older releases check.

3.3 what the check establishes

before any content is sent, the owner's journal establishes that the machine on the other end is AMD SEV-SNP hardware whose report carries a valid AMD signature chain and whose firmware is at or above a minimum level (§ 11, item 6 says what is not checked), with an NVIDIA H100 whose evidence chains to NVIDIA's root and whose firmware measurements match NVIDIA's signed manifests, that its boot evidence matches a boot file we published and pinned in that journal release, and through it the root filesystem that boot file commits to, and that the encrypted connection terminates inside that machine. each of these holds under the assumptions at the top of § 3.2.

3.4 what it does not establish

- **that nothing was kept.** measurement covers what was loaded at boot, never what the running program later does. the build is set up to write nothing to storage; the attestation does not prove that it kept nothing, and this design gives no per-request evidence of erasure.
- **that we published the right build.** the journal can check that the machine booted the build we published; it cannot check that the build we published deserves to be trusted. that judgment rests on the public source, on anyone's review of it, and on our record.
- **anything about the hardware vendors' honesty or correctness.**

- **where the machine is, or whose it is.** nothing in the evidence says the machine is in an Azure data centre or is one of ours (§ 3.2).
- **anonymity.** the service does not hide who is using it from us: our portal learns that an owner's credential was presented on each channel, and the engine sees the owner's network address (§ 11, item 10).
- **that every input the boot consumes is measured into a register.** the quote covers fourteen of the virtual TPM's twenty-four registers. the disk's partition entries are in none of them; the serving image pins them inside its signed boot file instead and refuses to boot if the disk differs (§ 5.2).

---

4. custody: who holds which key and which data

4.1 data

data	where it exists	who can read it	kept?
the owner's journal	the owner's device	the owner	never sent to the service
a request (text, images, audio) and its answer	in transit inside TLS; while processing, in the virtual machine's encrypted memory and in the GPU's on-package memory, which is not encrypted (appendix e, NVIDIA)	the owner's journal; inside the admitted engine, the gateway, the model and the transcription service	never written to storage: held in volatile memory only; no swap is configured, crash dumps are off, and logs stay in memory (§ 5.2). the model server's prompt cache holds state derived from prompts, per journal, until it is evicted or the engine restarts. a second cache, of values computed from recent images, is not kept per journal (§ 5.3)
the owner's service credential	the owner's journal; presented to the engine inside the verified channel	the owner's journal; the engine; our services portal, which issued it and receives it on every check	the portal keeps access records (below), not the credential
a per-journal processing count	inside the engine's memory	the engine	in memory only, for capacity management; the per-journal figures are never sent out (§ 5.3)
the health report: whether the engine's parts are running, error codes, the engine's token totals across all journals, counted from its start and frozen at each midnight UTC, and a monthly spread of how much journals used	sent from the engine every minute to our monitoring, which runs on Cloudflare	sol pbc, and Cloudflare, which runs the monitoring	yes, in our monitoring. it names no journal or sign-in and holds nothing an owner sent; on a day when one journal was the only one to use an engine, that engine's totals for the day are that journal's

data	where it exists	who can read it	kept?
access records: when the service was turned on and last confirmed, a fingerprint of the current credential, the disclosure the owner acknowledged; a log of each credential issued or refused, and when; a short-lived hand-off record while the journal collects its credential	our services portal (on Cloudflare)	sol pbc, and Cloudflare, which runs the portal	yes, until the owner closes their sign-in, when sol pbc deletes them; the hand-off record within a day or two
payment	Stripe	sol pbc and Stripe	per Stripe's and our terms
connection metadata (addresses, timing, volumes)	the networks in the path; the services portal keeps a request trace for credential checks that carries no journal or sign-in identifier	Microsoft Azure (the engine's network); Cloudflare (the services portal, including the owner's network address on the journal's own access check); our providers keep short-lived request logs	the engine writes none to storage; sol pbc keeps no connection metadata for this service beyond that trace

our published privacy policy is the line we hold, and it states each of these (solpbc.org/privacy, § confidential processing). all of the records above are Customer Data under our articles of incorporation (§ 4.3).

4.2 keys and signing material

key	held by	what it vouches for	if it were compromised
the engine's TLS key	generated inside the engine at connection time; never leaves it	that this TLS session terminates in the verified machine	by itself it is not enough to be admitted: the second quote binds the live session, not this key (§ 7, step 5). the key exists only in the engine's memory, for one connection
AMD's chip-endorsement key (VCEK) and AMD's root	AMD, in the processor and AMD's key infrastructure	that the report came from a genuine SEV-SNP processor at the stated firmware level	evidence from that processor could be forged; out of our control
the virtual TPM's attestation key	the confidential firmware layer (Microsoft's paravisor) inside the virtual machine	the boot measurements (PCRs)	quotes could be forged. the key is protected by the firmware layer's isolation and bound to the AMD-signed report; nothing in the journal would detect its theft
the firmware layer's ID signing key	Microsoft, by inference (§ 11)	that the launch was authorised by this signer: it signs the ID block, which names the expected launch measurement and policy	firmware we have not accepted could pass; see § 11

key	held by	what it vouches for	if it were compromised
the GPU's device identity key	NVIDIA, fused in the GPU	the GPU's confidential-computing evidence	GPU evidence could be forged until NVIDIA revokes it and the journal's accepted status for it (up to 24 hours) runs out
our Secure Boot signing key	sol pbc	that the boot image was signed by us; it is enrolled in the image's UEFI database, which makes the Secure Boot register predictable	a key-holder could sign a different boot image, but the journal compares the boot image's own measurement, so a different image is still refused
our transparency signing keys (a root and role keys)	sol pbc	which release and evidence records are current and authentic	an attacker could publish false records that verify under our root. the root mirrors (§ 9) would show a replaced root, not records signed with the existing keys; there is no append-only log or outside witness (§ 11)
our journal release signing keys	sol pbc, or a code-signing service acting on sol pbc's verified identity	the journal releases owners install, and therefore the pins in them	an attacker could ship a journal that trusts a different build; this is the same residual as § 3.2's row on sol pbc itself

the engine holds **no secret of ours**. it carries no credential for our services portal: it passes the owner's own credential through, and the portal answers whether that credential is good and, if it is, returns an identifier for the journal it belongs to (§ 5.3). someone who ran a copy of the published image on a confidential VM of their own in Azure, and redirected journals to it, would be admitted. the image gives them no way in, so they would not read content. they would see the connection metadata of the journals that reached it: which addresses connected, when, and how much they sent.

4.3 the covenant

sol pbc is a colorado public benefit corporation. Article 8 of our articles of incorporation forbids us to sell, license, sublicense or lease Customer Data under any circumstances, and forbids providing it to anyone outside the company except to a service provider strictly to deliver the service the customer asked for, as required by law, or at the customer's specific, informed direction. it defines Customer Data to include records and inferences about a customer, whether or not aggregated, de-identified or encrypted, so the access records and the in-engine count above are inside it. it cannot be amended without the founder's personal signature, and after the founder the language can only get stronger, never weaker.

two boundaries. the covenant governs what **we** do; the technical design governs what there is to take. and the covenant's own exception for what is required by law is why § 11 treats legal compulsion as a real adversary: the design's answer to compulsion is to keep no content to compel (the records in § 4.1 remain) and to make a changed build normally visible, not to promise that compulsion cannot happen. the engines run in a Microsoft Azure region in the united states and sol pbc is a united states company, so legal process would arrive under united states law.

5. architecture

5.1 the pieces

figure 1 shows the parts and what moves between them.

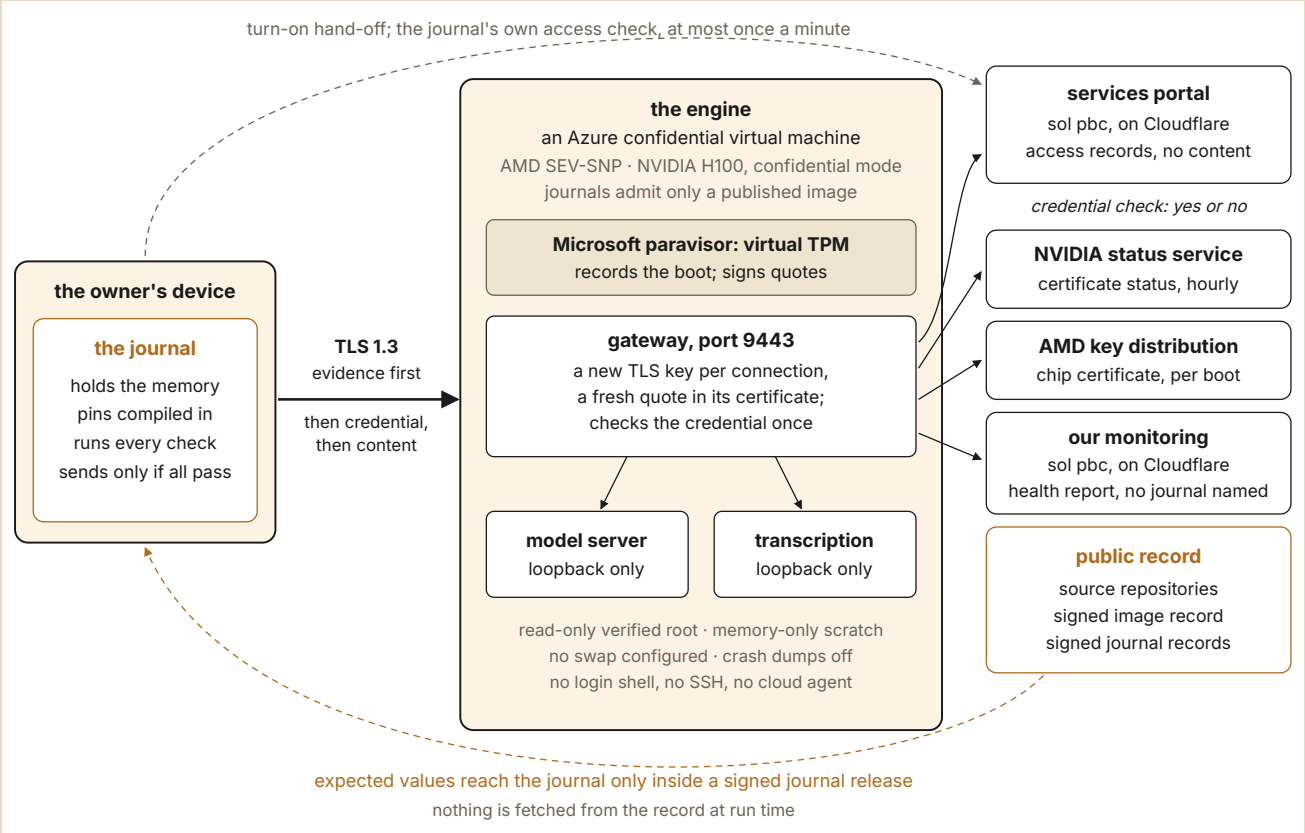


figure 1. the parts of the service and what moves between them. the journal sends content only to an engine it has verified; the engine's outbound calls carry no content.

- **the owner's journal**, on the owner's device. it holds the owner's memory, decides what work to send, runs every check in § 7, and is the only party that sends content.
- **the engine**: a confidential virtual machine in Microsoft Azure, size `Standard_NCC40ads_H100_v5`, with AMD SEV-SNP and an NVIDIA H100 in confidential-computing mode, Secure Boot and a virtual TPM. each engine boots one published image and nothing else. the virtual TPM is implemented by Microsoft's confidential firmware layer (the paravisor), which runs inside the virtual machine at the most privileged guest level.
- **our services portal**, which runs on Cloudflare. it sells and grants access, issues each journal its service credential, and answers one yes-or-no question about a credential when a channel opens. no content is ever sent to it.
- **our monitoring**, which runs on Cloudflare and receives the engine's health report every minute (§ 5.3). it accepts reports only from our engines' network addresses. the engine reads back nothing but a status code, and nothing in the engine acts on the report.

- **NVIDIA's certificate status service**, which the engine asks every hour for signed status responses about the GPU's certificates. it sees certificate serial numbers, which identify the GPU, and nothing about any owner.
- **AMD's key distribution service**, from which the engine fetches its processor's certificate once per boot. it sees the chip's identity and firmware level.
- **the transparency site and the public repositories**, where the image's source, its measurements and its signed release record are published (§ 9).

5.2 the serving image

the serving image is assembled by a public recipe: repository `solpbc/conf-proc`, at the commit appendix b names. figure 2 shows how it boots and what each stage is measured into.

inputs, all pinned. the root filesystem starts from the public SGLang serving container, pinned by its registry digest and by a digest over its unpacked file tree; the build applies five patches to its files, each refused unless the file it patches matches a pinned digest (§ 5.3). the language model (Qwen3.5-4B) is pinned per file and by tree digest; the speech model (NVIDIA Parakeet TDT 0.6B v3) by its SHA-256, which the transcription service checks again at start. the kernel is Canonical's Azure kernel, taken byte for byte from Canonical's signed package; the NVIDIA open kernel modules and user-space driver come from Canonical's and NVIDIA's packages. every input is pinned by SHA-256, and the build tools themselves are a pinned set of Ubuntu packages. the build runs without network access and refuses a source tree with uncommitted changes. the image records its commit inside the root filesystem, so a rebuild must be made at exactly that commit to match. "public" here means the recipe and the digest of every input are public. the model server, the kernel, the GPU driver and the models are other parties' published binaries and weights, not compiled by us, and changed only by the five patches above; our own programs in the image (the boot program, the gateway, the evidence collector, the health reporter) are the source in the repository.

one signed boot file. the kernel, a small initial ramdisk and a fixed kernel command line are packed into one unified kernel image (UKI), and that file is what our Secure Boot key signs. Canonical's own signature on the kernel is not what the firmware checks; ours, over the whole file, is. there is no shim and no machine-owner key mechanism: the firmware checks the UKI directly against the image's UEFI signature database. the command line names the root filesystem's dm-verity root hash, so the hash is inside the signed, measured file. the command line also turns off systemd's automatic discovery of partitions and swap (`systemd.gpt_auto=0 systemd.swap=0`), and the ramdisk carries a copy of the disk's expected partition entries.

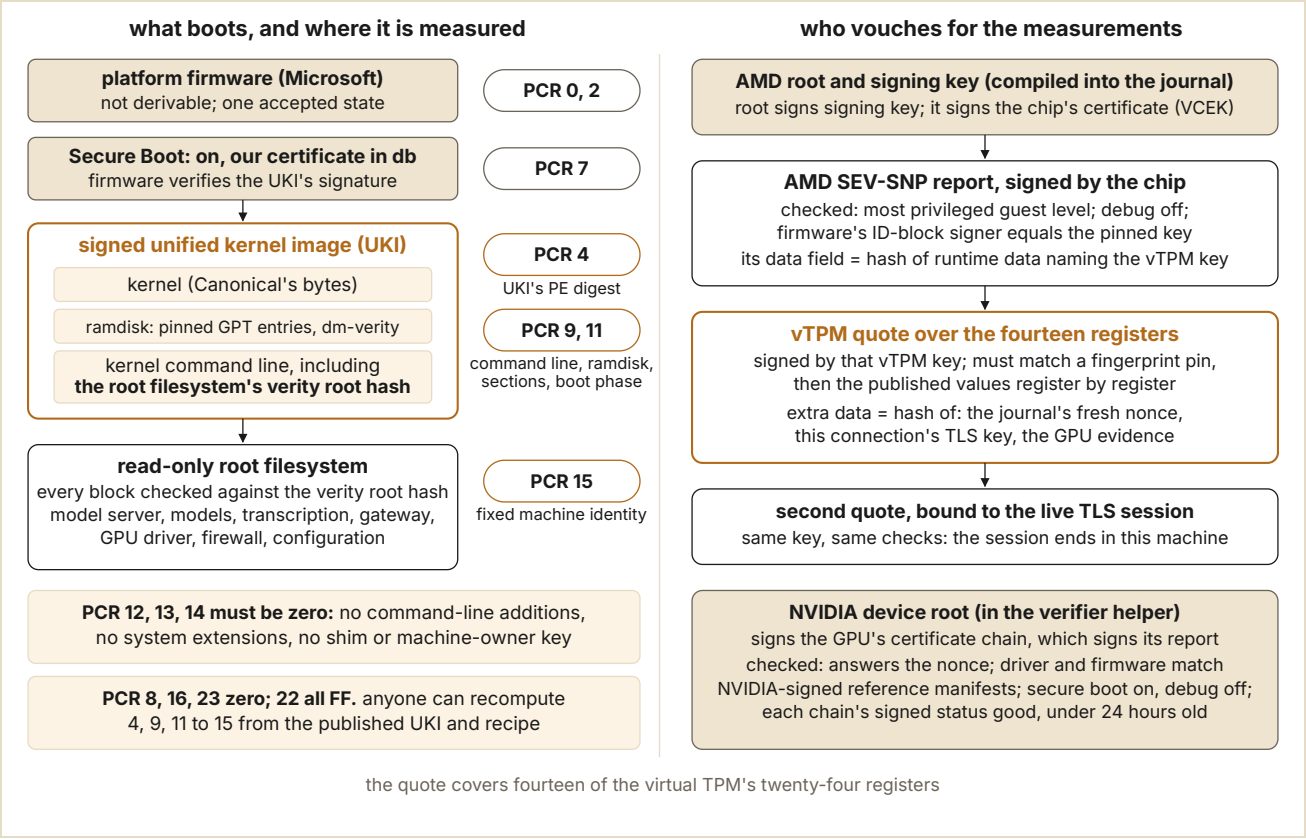


figure 2. left: what the serving image boots and which register records each stage. right: the chain of signatures the journal follows to trust those registers.

a verified, read-only root. the ramdisk's only program first compares the boot disk's partition entries with the copy it carries, and refuses to go on unless the two are byte for byte the same (with a valid protective MBR, both GPT headers and both entry arrays checked, and the backup equal to the primary; the headers are checked for consistency, not against the copy). it then loads the dm-verity driver, opens the root filesystem's data and hash partitions, sets up verification against the root hash from the command line, mounts the result read-only (and refuses if it is not), then hands over to the system's init. every block of the root filesystem is checked against that hash as it is read. if the verification cannot be set up, the machine halts before anything serves; a block that fails verification later cannot be read. in qualification, a disk with one bit flipped in the root filesystem never opened its serving port.

nothing written down. the image is set up so that nothing a request carries is written to persistent storage:

- the root filesystem is read-only. the only writable filesystems are memory-backed (`/run` and `/dev/shm`), and every service's scratch space, caches and home directory point there. the model is copied into memory before serving starts.
- no swap is configured, and the build refuses any swap file or swap entry. swap discovery is also off on the measured command line, and a disk with an added partition does not boot. crash dumps are disabled.
- the system journal is held in memory only (16 MB cap) and forwards nowhere, and the image has no log shipper. the gateway logs short event names, never request text or error text; the transcription service logs reason classes and byte counts. we have not separately shown that the model server's own log

lines carry no content; whatever they carry stays in memory and is lost with it.

no door to log in through. the image removes the SSH server and client, masks every login terminal, and carries no Azure guest agent (so the cloud's "run command" and extension features have nothing to talk to), no cloud-init and no container runtime. services run as separate unprivileged users with no capabilities. Canonical's kernel is built to enter its lockdown mode when Secure Boot is on, which refuses unsigned kernel modules; Secure Boot being on is part of register 7, which the journal checks. the image itself does not set or test this, so it is a property of the packaged kernel that we have not demonstrated.[1] the image offers no supported way to administer a running engine, so an unhealthy engine is replaced, not repaired. the production command line names no serial console and the system journal does not forward to one. serial console output saved during the qualification of the image before this one held the kernel boot stub's start-up messages and nothing after boot; we have not read the serving image's, and have not tested what a failing driver or service could write there. qualification of an earlier production image showed the cloud's run-command and extension features do not execute on it; the serving image removes the same doors.

5.3 inside the engine

three services handle requests:

- **the gateway** listens on port 9443. for each connection it makes a new TLS key, puts evidence into its certificate (§ 7: a fresh TPM quote, with the processor's report and the GPU's evidence), completes the exporter proof, checks the owner's credential once, then relays exactly two kinds of request, chat completion and transcription, to the services behind it over the local loopback interface. it terminates TLS, so it handles content in the clear in memory, as the model does. it derives the exporter value from its own TLS connection, and the evidence collector is a separate program it runs once per quote, not a network service.
- **the model server** (SGLang serving Qwen3.5-4B, with a 262,144-token context) and **the transcription service** (Parakeet), each listening on loopback only. the model server holds a cache of recent prompt state in GPU memory to speed up repeated prefixes. it is volatile and is lost on restart. it takes images, audio and video only as data carried inside the request: a request that names a web address or a file path for any of them is refused before anything is opened or fetched. we set no shorter lifetime for the prompt cache: an entry stays until the model server evicts it to make room or the engine restarts, and nothing we have built clears a journal's entries when its channel closes. the cache is kept per journal: the gateway removes any label a request arrives with and adds one derived from the journal identifier our services portal returns for the owner's credential, and the model server refuses a request that does not carry exactly one label, so one journal's requests cannot reuse another journal's cached prompt state. the label comes from our portal, which is not measured, so this is a separation we configure, not one the hardware enforces. each answer reports how many of its prompt tokens came from that journal's cache. that separation covers the prompt cache only. the model server's published source also keeps a second, smaller cache of the values it computes from recent images, keyed by the image and not by journal, and our recipe does not turn it off. if two journals send the same image, the second can be answered from the first's cached values: the answer is the same, but how fast it comes can show that the image was seen recently. we have not confirmed in the built image that this cache is in use.

the outbound firewall drops every packet by default. the model server and the transcription service may reach the loopback interface and nothing else; they also cannot reach the gateway's own two loopback ports, and they cannot send a reply off the machine. inbound traffic is filtered too: the engine first drops any

IPv4 TCP packet that carries IP options, then accepts only loopback, replies to its own connections, DHCP and the gateway's port. every other process may use DHCP and may query Azure's platform DNS resolver. the gateway may also make outbound connections on port 443, which its code uses for three things only: the credential check with our services portal, the hourly NVIDIA status fetch and the per-boot AMD certificate fetch. a separate health reporter, running as its own unprivileged user, shares that rule and uses it for one thing: the health report. none of the four carries content. the port 443 rule is by port, not by destination host.

the credential check. on the first request of each admitted channel, the gateway forwards the owner's credential to our services portal and receives yes or no. a yes also carries an identifier for the journal the credential belongs to, which the portal derives from its record of that journal with a key the engine does not hold; a yes without exactly one such identifier is treated as a no. the answer can also carry an allowance policy (a daily amount, a cooldown, an audio rate and a limit on simultaneous requests); the portal's source has that switched off, so none is sent and no allowance is enforced. if the portal cannot be reached, the answer is no. it sends no content and no usage figures. the engine holds no credential of its own. channels have a fixed lifetime: no new request after 30 minutes, and the connection is closed at 40 minutes even mid-request, so withdrawn access stops being honoured within about 40 minutes. the journal's own check of its access, made at most once a minute while the owner has the journal's thinking view open, can show the owner sooner; it does not close an open channel.

the in-engine count. the engine counts how much it has processed for each journal, under a label derived with a key generated at start and never stored, so the label cannot be linked to the journal outside the engine and does not survive a restart. the per-journal count lives in memory, is reachable only on the engine's loopback interface, and is never sent out.

the health report. a separate reporter sends our monitoring a fixed-form report about once a minute. its fields are set by a published schema: whether the gateway, model server, transcription service, GPU set-up and firewall are running, how often they have restarted, whether the GPU is in confidential-computing mode, whether each model answers, error codes, the evidence collector's state, the engine's token totals across all journals, counted from its start and copied once a day at midnight UTC, and, once a month, how many journal-days fell into each of three bands of use. a day counts toward that spread only if at least ten journals used the engine that day, and a band with fewer than five journals is dropped. the report names no journal, credential or sign-in and carries nothing an owner sent, but its totals are derived from use: on a day when one journal was the only one to use an engine, that engine's totals for the day are that journal's. the report is the engine's own statement, not evidence; nothing in the journal or the engine acts on it.

---

6. what is measured, register by register

the engine's virtual TPM quote covers fourteen of its twenty-four registers. figure 2 shows where each value comes from. for each register, the table says who determines it, whether someone outside sol pbc can compute it from published material, and how the journal checks it. the full values are in the journal's source and in the image's published record; appendix b gives where.

register	what it records on this image	who determines it	outside-derivable?	the journal checks
0	platform firmware code	Microsoft	no	whole-file fingerprint only: one accepted firmware state
2	platform option code	Microsoft	no	whole-file fingerprint only; as register 0
4	the firmware's measurement of the boot application: the UKI's PE (Authenticode) digest, which covers its contents but not its signature	the image	yes, from the published UKI	must equal the image's published value
7	Secure Boot state, the Secure Boot databases, and the certificate that verified the UKI	Microsoft's firmware policy and our enrolled signing certificate	no; published as observed	must equal the published value
8	expected unused	none	trivially	must be zero
9	the kernel command line and the ramdisk, as measured by the linux kernel's EFI stub	the image	yes	must equal the image's published value
11	the UKI's sections, measured by systemd's boot stub, and the boot-phase words "sysinit" and "ready"	the image	yes	must equal the image's published value
12	extra command-line, credential or configuration additions	the image	yes	must be zero
13	system extensions	the image	yes	must be zero
14	shim and machine-owner-key events	the image	yes	must be zero
15	the machine identity, a fixed value the recipe sets	the image	yes, from the recipe	must equal the published value
16, 23	debug and application registers; resettable from inside the machine, so zero shows only that they are empty at quote time	none	trivially	must be zero
22	a register the machine's software cannot extend (our test of an extension was refused); it starts at all `FF`	none	trivially	must be all `FF`

the journal recomputes the quote's signed register digest from the register values it is given, requires the quote to cover exactly this list in one SHA-256 bank, and hashes the whole register file to compare with the fingerprint pins first (§ 8.1).

what the registers establish. registers 4, 9 and 11 to 15 pin the boot file and everything it measures. register 4 records the firmware's digest of the UKI, which covers its contents; the UKI contains the command line, and the command line contains the root filesystem's verity root hash. so a matching register 4 means the engine booted exactly the published boot file, and that file will only read root-filesystem blocks that match the published hash. the same file also contains the expected partition entries, so a matching register 4 also means the engine will not boot from a disk laid out differently from the published one. registers 12 to 14 at

zero mean the boot stub and systemd measured no command-line additions, credentials, system extensions, or shim and machine-owner-key state. register 7 pins Secure Boot on, with our certificate as the one that verified the boot file. registers 0 and 2 pin the platform firmware to a state we have accepted.

what they do not establish. they say what was loaded at boot. they do not say what the software has done since. they do not cover the GPU's firmware (that is the GPU leg's job, § 7), and nothing covers what is in GPU memory. the processor's report says which signer authorised the firmware launch (its ID key digest, which the journal pins) but the journal does not compare the launch measurement itself (§ 11).

anyone can recompute the application registers. the image's record publishes the signed UKI, and the image's repository includes the derivation program. running it on the published UKI, with the fixed machine identity the recipe sets, reproduces the published values for registers 4, 9, 11 and 15 and the zeros for 12 to 14. we repeat this from a fresh download each time this paper is updated, and appendix c records the result. registers 0, 2 and 7 cannot be derived this way: they come from Microsoft's firmware and Secure Boot policy, and we publish them as observed on the qualification machines.

the root filesystem can be rebuilt from published inputs. the root filesystem and the virtual disk are not published as downloads. what is published is the recipe, every input's digest, the pinned toolchain, the full file list (an SPDX software bill of materials) and the verity root hash inside the signed UKI. the build is designed to be reproducible: our own two production builds from separate fresh clones, with the same inputs, toolchain and signing key, produced byte-identical boot files, disks, root filesystems, verity trees and file lists. a builder without our signing key would get different signature bytes and a different register 7, and should get the same register 4, root filesystem and verity root hash, because register 4's digest leaves the signature out. no one outside sol pbc has rebuilt it yet (§ 11).

---

7. what the journal checks before it sends

this section follows the journal's code in order, from the moment the journal decides to use confidential processing to the moment the first byte of content leaves. every expected value it compares against is compiled into the release; the journal fetches nothing from our transparency site or from AMD at run time. the links in appendix b point at the exact lines, and figure 3 summarises the sequence.

in the vocabulary of the IETF's remote attestation architecture (RFC 9334), the engine is the attester, the journal is both verifier and relying party, a pin is a reference value, and AMD, NVIDIA and Microsoft are endorsers. the channel is a form of RA-TLS: evidence is carried in the server's certificate, and a second piece of evidence, bound to the session's TLS exporter, follows the handshake (appendix e gives the prior work).

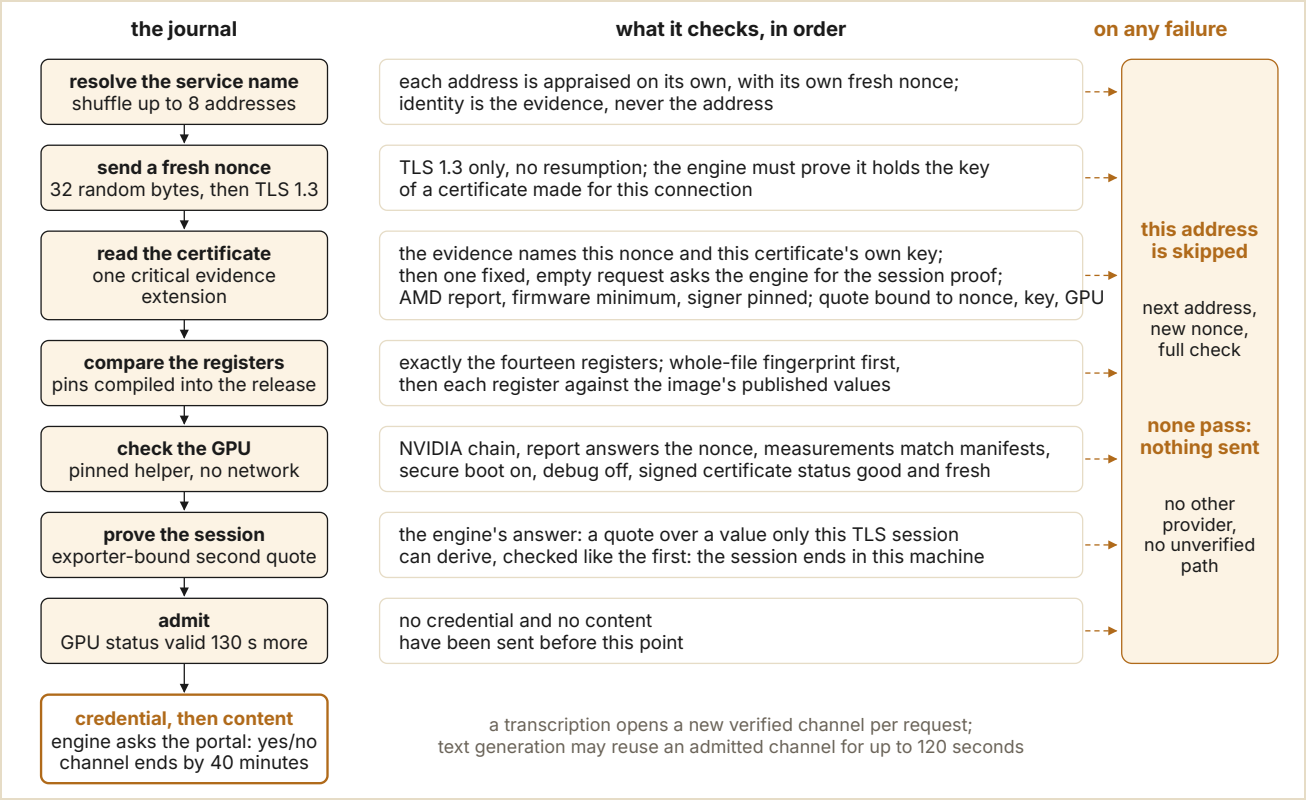


figure 3. the journal's checks, in the order the code runs them. a failure at any step skips that address; if no address passes, nothing is sent.

1. where to connect. when the owner turns the service on, our services portal hands the journal three things: the service address, the model name and the owner's service credential. the journal stores them locally. no engine address is compiled into the journal, so our portal decides **where** the journal connects. it does not decide **what** the journal accepts: identity is the evidence, so a wrong or hostile address fails the checks below.

2. every address, appraised on its own. the journal resolves the name, takes every address it returns (at most eight), shuffles them and tries them one at a time. each attempt draws a fresh 32-byte random nonce from the owner's operating system and runs the full appraisal below. the first address that passes everything is used. an address that fails is simply passed over; no credential or content is sent to it. a failed check at one address never stops the walk. two things do: a failure on the owner's own device, such as a missing verifier helper, and time, since no new address is tried once 120 seconds have passed since the walk began.

3. the connection. after the TCP connection opens, the journal sends a short fixed preface and its nonce, in the clear. the nonce is not secret; its job is freshness. the engine then builds a new TLS key pair and a self-signed certificate for this connection only. the journal speaks TLS 1.3 only, with session resumption disabled. it does not check the certificate against any certificate authority, because the certificate's authority is the evidence inside it, but it does require the engine to prove possession of the certificate's private key in the TLS handshake.

4. the evidence in the certificate. the certificate must carry exactly one critical extension holding the composite evidence, and that evidence must name the journal's own nonce and the certificate's own public key. once those three things hold, the journal sends one fixed, empty request inside the TLS session, asking the engine for the session proof of step 5, so that the engine can prepare it while the journal works. that request is the same bytes every time: it carries no credential, no content and nothing derived from the owner. the journal then appraises the processor, the TPM quote and the GPU, in that order.

- **the processor (AMD SEV-SNP).** the evidence carries Azure's attestation report, which wraps an AMD SEV-SNP report and a block of runtime data. the journal verifies the chip's certificate (VCEK, carried in the evidence) against AMD's signing key and root for the Milan, Genoa and Turin processor families, which are compiled into the journal; verifies AMD's signature over the report; requires the firmware (TCB) level in the chip's certificate to equal the level the report states; and requires that the report was produced at the most privileged guest level (VMPL 0), that the guest's debug policy is off, and that the report's own data field equals the hash of the runtime data. it then requires that the report's **ID key digest** equals the one value pinned in the journal. that digest fingerprints the key that signed the launch's ID block, which names the expected launch measurement and policy of the firmware Azure launched the machine with. this ties the TPM to firmware launched under the signer we accept, not merely to some SEV-SNP machine. it is a signer pin: it accepts whatever firmware that signer authorises. last, it requires each of four parts of the processor's firmware level, as read from the chip's certificate, to be at or above a minimum compiled into the journal (§ 8.1). this report is produced by Azure's firmware, not for each connection; the freshness of the evidence comes from the TPM quote below, which answers the journal's nonce.
- **the virtual TPM.** the runtime data that AMD's report vouches for names the virtual TPM's attestation key. the journal requires that the evidence's attestation key is that key, verifies the TPM quote's signature with it, and requires that the quote's "extra data" field equals a hash over a fixed domain string, the journal's nonce, the hash of the TLS public key and the hash of the GPU evidence. that single value ties the fresh nonce, this TLS key and this GPU's evidence into one quote signed inside the verified machine.
- **the boot measurements.** the quote must cover exactly the fourteen registers listed in § 6, in one SHA-256 bank. the journal hashes the whole register file and requires it to equal a fingerprint pinned in this release; then it looks up the set of register values published for the image that fingerprint belongs to, and compares the registers one by one with that set (§ 6, § 8). a fingerprint with no set of values, or with more than one, is refused.
- **the GPU.** a helper program shipped inside the journal, built from our public fork of NVIDIA's attestation SDK, verifies the GPU's signed attestation report against NVIDIA's device root, checks that the report answers the journal's nonce, and compares the GPU's firmware measurements, for its VBIOS and for the firmware its driver loads, with NVIDIA-signed reference manifests that are compiled into the journal and checked against pinned digests before use. the journal requires that the measurements match, that the GPU reports secure boot on and debug off, and that every certificate below each root in the three chains involved, except the GPU's own per-device alias certificate, has a current, good status from NVIDIA. that status arrives as NVIDIA's own signed status responses, which the engine fetches and carries in its certificate; the journal judges their age against the owner's own clock and accepts a response for at most 24 hours from its own status time, allowing a minute of clock difference. the helper runs with an emptied environment and is given no network address to contact. the GPU's report names no virtual machine. it is tied to this machine because the measured engine software collects it from its own GPU and hashes it into the quote.

5. the channel itself. the first quote binds the TLS key the engine generated. the second binds the session itself. after the appraisal passes, the journal derives a TLS exporter value from its side of the live session and reads the engine's answer to the request it sent in step 4: a second TPM quote, signed by the same attestation key. the engine derives the same exporter value from its own side of the connection and puts it in the quote's extra data; the journal never supplies it. the journal checks this quote exactly as it checked the first, register appraisal included. this establishes that the session the journal actually holds ends inside the verified machine, without relying on where the TLS key went.

6. admission, then the credential, then content. if the GPU's signed status will expire within the next 130 seconds, the journal refuses and tries again later. otherwise the channel is admitted. only now does the journal send the owner's service credential, and only after that, content. the engine checks the credential with our services portal, which answers yes or no.

so before admission the journal has sent an engine three things: the preface and nonce, the TLS handshake, and the one fixed request of step 4. an engine that fails the appraisal receives no credential, no content and no identifier from the journal. it does see the connection itself: the owner's network address, the timing and the shape of the TLS handshake.

reuse. a transcription opens a fresh, fully verified channel for every request. text generation may reuse an admitted channel for up to 120 seconds from admission, and never after the GPU status it was admitted on stops being current. one process of the journal holds the channel and also carries the text-generation calls of the journal's other processes on the same device, over a local socket or pipe that only the owner's account can open. so a journal may start new work on an admitted channel for up to about two minutes after an owner's access ends, and a request already in progress can run until the engine closes the channel (§ 5.3).

---

8. pins, rotation and release

8.1 what a pin is, and where it lives

every value the journal compares against is compiled into each release from the journal's public source:

- **a fingerprint pin** for each accepted state: the SHA-256 of the whole register file the engine's quote covers. a current release carries two: one for the serving image and one for the image before it, each in the one Azure firmware state it has been qualified on. each image's record also publishes the value it would show on an earlier firmware state, marked as a prediction; that value is not pinned.
- **the firmware signing key pin**: one ID key digest, the signer of Azure's confidential firmware launches.
- **the minimum processor firmware level**: a lowest accepted value for each of four parts of AMD's firmware level, the same for every accepted image.
- **the register list and the per-register values** for registers 4, 7, 9 and 11 to 15, taken from the image's published record, plus the fixed values the structural registers must hold. there is one set of values per accepted image, chosen by fingerprint.
- **AMD's root and signing certificates**, a set of NVIDIA's reference manifests for each accepted image, with their digests, and the map from each fingerprint to the set and GPU profile it must be admitted with.

the whole-file fingerprint is checked **first**. any change to any covered register therefore fails as a fingerprint mismatch before the per-register comparison runs. the per-register comparison is a second layer, and for a pinned image it catches nothing the fingerprint misses. what it adds is that each pinned fingerprint is **explained**: tied, register by register, to values an outsider can derive from the published image (§ 6).

nothing in the journal fetches, updates or overrides these values at run time, and none of them expires. a new value reaches an owner only by a new journal release, which is signed, and normally built from public source and recorded (§ 9.4 and appendix d list the gaps).

8.2 rotating to a new build

a new image measures differently, so moving to it is a pin rotation, and we run it in an order where a journal release trusting the new build ships before any engine changes:

1. the new image is built, qualified on the production hardware (including a reboot, because a value measured on a first boot can differ from every later boot) and its measurements recorded;
2. its values are reviewed and marked active in our internal ledger;
3. a journal release ships trusting **both** the old and the new values;
4. the new engine goes live; the old pin is marked retiring;
5. a later journal release drops the old pin;
6. the old pin is recorded as retired.

the one step that matters to owners is the third: journals learn the new value before any engine changes. an owner on an older release that does not know the new value is refused by the new engine and sends nothing to it. between steps 3 and 5 a journal holds both sets of values and may be served by either image. the old engine can stay in service for a while after step 5, so that owners who have not yet updated are not cut off, and its retirement is a separate decision. the sequence has run end to end twice, and a third rotation is at step 4 on the state-as-of date; appendix d gives the releases and dates.

8.3 what moves the platform registers

registers 0 and 2 record the cloud provider's firmware, and register 7 records Secure Boot policy. Microsoft can update firmware on its own schedule, and an Azure firmware update has moved registers 0 and 2 with nothing about our image changing (appendix d). each image is pinned in one firmware state, so an engine on a host with different firmware is refused by every journal; we place engines only on hosts whose evidence passes. the minimum processor firmware level is a compiled-in value too, set at the level our hosts reported when the serving image was qualified: a host below it is refused, and raising it takes a journal release. register 7 records the Secure Boot databases, so a change to those would move it too. the journal also pins one firmware signer, so if Microsoft rotated that key, every engine would be refused until a release accepted the new one. each accepted firmware state is a separate fingerprint pin, so a new firmware state is refused until a journal release accepts it. that is safe, and it is an availability risk: a provider roll can stop the service for every owner until a release ships. our design for this, a separately published and signed platform baseline with a firmware roll no longer requiring a journal release, is not built; see § 11.

8.4 which journals check what

the protection an owner gets is the protection in the release they are **running**, and owners update on their own schedule.

- **a current release** is from the overlap of a rotation (§ 8.2, steps 3 to 5). it admits the serving image and the one before it, and runs every check in § 7 on either. the address walk (§ 7, step 2) takes engines in a random order, so such a journal may be served by either image until the earlier engine is retired or a release drops its pins.
- **a release from before this rotation** admits the image before the serving one, or an image earlier still, and not the serving image. once the engine of every image it pins is retired, such a journal finds no engine of ours it accepts, and sends nothing. the oldest releases also run fewer checks.

appendix d lists, release by release, which images each admits and what it checks. owners on older releases should update: § 11, item 14 says what they are exposed to until they do.

---

9. the transparency surface

9.1 two sites

- **trust.solstone.app** explains, in plain language, what we have published and links to the originals. it states that where it and the signed records disagree, the records win.
- **transparency.solstone.app** holds the signed records and their evidence. its bare address redirects to the trust site.

9.2 how records are signed

the evidence site is a TUF repository (The Update Framework) with in-toto statements in DSSE envelopes. it has one root key, a separate key for each top-level role (targets, snapshot, timestamp) and a separate key for each delegated area: software releases, service images, verification results and a legacy migration. only the timestamp key is used automatically, to re-sign a 7-day freshness window; the root authorises it for nothing else. the keys that sign release records are separate from the roles reserved for independent rebuild and audit results, which today have no outside keys enrolled (§ 11).

the root's fingerprint is published at more than one location sol pbc controls: pinned in the public `solstone-transparency` repository and in a mirror file on solpbc.org. (transparency systems use "witness" for an independent party that co-signs what a log publishes. these copies are ours, so we call them mirrors.) each time this paper is updated we compare both with the live root; appendix c records the result and the root's version and expiry. a verifier pins the root out of band and checks everything else against it. TUF makes records authentic and fresh under these keys. it does not keep an append-only history: a record signed with our keys verifies whether or not it is true, the mirrors would show a replaced root but not such a record, and nothing outside sol pbc yet records what was published when (§ 11).

9.3 the serving image's record

the record for the serving image is the release record for its commit on the evidence site (appendix b, T3). it binds fourteen files by length and SHA-256: the signed UKI, our Secure Boot signing certificate, the kernel command line, the derived registers, the expected registers for the firmware state it is pinned in, a prediction for an earlier firmware state that is marked as unqualified and is not pinned, the build manifest, the input and toolchain manifests, the file-origin list, the software bill of materials, a build note, a test report and the release notes. its claim fields say what it proves and, at more length, what it does not, including: *"these checks establish load-time build integrity. they do not prove subsequent control flow, absence of defects or that sol pbc chose the right build to publish."* the records of earlier images stay published, unchanged. the serving image's publication was audited internally, in a separate review from the step that published it, and passed.

9.4 the journal's own records

the journal is the verifier, so its own provenance matters too. a journal release normally gets a signed record binding every file it delivers on every platform, and each record is then audited in a separate review from the step that published it. these audits are internal: we use the word in our own records, and it means a release check made inside the same company, not an independent audit. sol pbc is a company of one: its founder. an AI agent helps him run it, and the audits are run by that agent, so the separation is between processes, not between independent people. they establish that the published bytes and records are complete and consistent, not that the code is correct. the records do not claim reproducible builds.

the release this version was checked against has a signed record binding 26 files across five platform targets (linux on arm64 and x86-64, macos on arm64, windows on x86-64 and arm64), and its internal audit, on 17 november 2026, passed every criterion with no findings.

the records are not complete. some earlier releases have no record, one audit's verdict stands at fail, some records leave out a platform that skipped the release, a run of earlier windows builds was published outside its versions' records, and one release was audited only by the step that published it. appendix d lists each one.

9.5 how to check what we have published

the transparency repository carries the verifier. from a clone of it, with a root pinned out of band:

```
bun run bin/solstone-transparency.ts verify-v2 --root protocol/tuf-root.json
bun bin/verify-release.ts --root protocol/tuf-root.json --product spp --version <image commit>
bun bin/verify-release.ts --root protocol/tuf-root.json --product journal --version <journal re
lease>
```

the image's version is the first seven characters of its commit. appendix b gives it and the journal release this version was checked against, and appendix c records that both verified this way. to check the image's registers, download its signed UKI from the record and run the image repository's `appliance/derive_pcrs.py` on it. to check what the journal compares against, read `pins.rs` at the release tag.

10. failure behaviour

if the journal cannot verify, it sends no credential and no content, and it does not send anywhere else.

condition	journal behaviour	what the owner sees
no address reachable	no credential or content sent	"can't reach confidential processing right now. nothing is being sent."
any address fails any check in § 7	that address is skipped; if none passes, no credential or content is sent	"couldn't verify the service. nothing is being sent."
the GPU's signed status is missing, not good, or about to expire	refused before content	as above
the journal's verifier helper is missing or altered on disk	refused before any connection	as above
the credential is refused	the channel is closed; nothing further is sent	the journal reports that access has ended
verification is pending or stale	text work waits; audio stays on the device and is transcribed later	"transcription is waiting. nothing is sent until your journal verifies the service. your audio stays on your device and is transcribed later, after the check passes."

no failure moves work to another provider or to an unverified path. the owner can turn confidential processing off at any time, which sends new work back to their own machine. the owner-visible status is deliberately coarse: almost every verification failure is recorded as "attestation rejected", with a reason code added, such as a fingerprint or register-list mismatch.

on the engine side, admission fails closed too: a missing or stale GPU status, or evidence that cannot be collected, means no certificate and no channel.

11. limits and open gaps

the limits below are where we think the design is weakest, or where its guarantees are narrower than a quick reading suggests.

1. **load-time, not run-time.** the check establishes what booted. anything that compromises the running software afterwards, through a defect in the kernel, the model server, the gateway or the transcription service, is invisible to it. root on a compromised engine could, for example, load other code without changing any measured register. the image narrows what can be loaded and removes ways in; it cannot rule out an exploit. the image is also fixed at build time: a defect found later in anything it contains stays in the serving image until a new image replaces it, and replacing it takes a build, a qualification and a journal release (§ 8.2). the image before the serving one is a case of this: two defects found after it was built are fixed in the serving image, and it has both for as long as its engine stays in service (appendix d). the gateway's port is reachable from the internet. the attacker need not be an outsider: an owner with a valid credential can send hostile text and images to the same model

server, and hostile audio to the same transcription service, that every other journal uses, and a compromised model server could put one journal's content into another's answer without sending anything off the machine.

2. **non-retention is a property of the build, not a proof, and it is about storage.** the image is set up to write nothing an owner sends to storage and to send it nowhere else. the attestation shows that this build is the one that booted; it does not prove that nothing was kept, and this design gives no per-request evidence of erasure. what the engine holds in memory has no stated lifetime: the prompt cache keeps state derived from a journal's prompts until the model server evicts it or the engine restarts, and nothing we have built clears it when the owner's channel closes. a second cache, of values computed from images, is not kept per journal, so the speed of an answer can show that another journal recently sent the same image (§ 5.3).[2]
3. **we choose the build.** the journal verifies that the engine booted the build we published and pinned. it cannot verify that the build we published deserves trust. the recipe and its pinned inputs are public (§ 5.2), but no party outside sol pbc has yet rebuilt the image or audited it, and the transparency system's roles for independent rebuild and audit results have no outside keys enrolled. the root filesystem and disk are not published as downloads, so checking them means rebuilding.
4. **a compelled or malicious change would normally be visible, and is not prevented.** if sol pbc, voluntarily or under legal compulsion, built an image that kept content and shipped a journal release trusting it, owners who installed that release would send to it. what stands in the way is that the change would normally be public: the new pin appears in the journal's open source, the new release is signed and normally recorded, and the image's measurements change. nothing technically forces that publication.[6] a release could be built from unpublished source or delivered to some owners only, and detection needs someone to be watching. owners update on their own schedule (a new linux install made with our installer updates itself daily unless the owner turns that off, and an earlier one does once its owner agrees), and nothing alerts anyone automatically when a pin changes. a desktop journal does fetch, once a day, a notice signed with our release key that names the release carrying the latest security fix, and tells the owner if theirs is older. the journal adds no identifier and no version to the request; the host that serves the notice does see the journal's network address. the notice changes no pin and refuses no channel. its one effect beyond informing is that a linux install's automatic update reports a failure when neither the installed release nor the one on offer carries the fix the notice names. it shows nothing if it cannot be fetched or verified, and it is signed by the same company this item is about.
5. **Microsoft's firmware layer is trusted.** the virtual TPM that records the boot is implemented by Microsoft's paravisor, which runs at the most privileged guest level and can read all of the virtual machine's memory. the journal binds the TPM's key to the AMD-signed report and pins the key that signs Azure's confidential firmware launches, but it does not compare the launch measurement against accepted values, does not check the attestation key against Microsoft's certificate for it, and does not check the configuration fields Azure signs into the same report.[3] the pin is on a signer, so it accepts whatever firmware that signer authorises. that the pinned key is Microsoft's is an inference: every Azure report we have observed carries it, and Microsoft's documentation lists only earlier values.
6. **AMD's report is checked narrowly.** the journal verifies the chain to AMD's pinned roots, the report signature, the privilege level and the debug policy. it requires the processor's firmware (TCB) level, read from the chip's certificate, to equal the level the report states and to be at or above a compiled-in minimum. that minimum is the level our hosts reported when the serving image was qualified. it refuses

a host below that level; it is not a statement that the level is the newest AMD has released, and raising it takes a journal release.[4] the journal does not check AMD's certificate revocation list[9] and does not examine the other guest policy fields. it accepts reports from three processor families, though the machine size our engines use runs on one.[5]

7. **the GPU leg rests on a helper, on NVIDIA, and on the engine's own software.** cryptographic checks of the GPU evidence are made by a helper built from our fork of NVIDIA's attestation SDK, which the journal runs from its verified install and whose output it trusts. the journal does not check an explicit confidential-computing-mode claim; it relies on the measurement match, secure boot and debug claims (the engine also refuses to start serving unless the GPU reports confidential computing on, but that is inside what is being verified). NVIDIA's revocation delay is not published, and our accepted status age adds up to 24 hours to it. the GPU's evidence is also tied to this machine by the measured engine software that collects it, not by the hardware: the GPU's report names no virtual machine.
8. **the platform registers are accepted, not derived.** registers 0, 2 and 7 come from Microsoft's firmware and Secure Boot configuration and are published as observed. each accepted firmware state is its own pin, and a current release holds one for each image it admits, so a provider firmware update stops the service for every owner until a journal release accepts the new state; a rotation of Microsoft's firmware signing key would do the same. the design for a separately signed platform baseline, which would remove that coupling, is not built.
9. **the gateway handles plaintext and has outbound HTTPS.** the model and transcription services are confined to loopback; the gateway, which terminates TLS, may connect out on port 443, and its code uses that for three fixed purposes; the health reporter shares that rule for a fourth. they and the other services outside the model's confinement may also use DHCP and Azure's platform DNS resolver. the confinement is by port, not by destination, so after a compromise of the running software, outbound HTTPS and DNS are paths out.
10. **the credential links access to channels.** our services portal learns, for each channel, that a given owner's credential was presented, when, and from which engine. the journal also asks the portal about its own access, at most once a minute while the owner has the journal's thinking view open, from the owner's network address. the portal also tells the engine, for each channel, an identifier for the journal the credential belongs to; the engine uses it only in memory, to key the count, the prompt cache and an allowance that is not enforced today. the portal is not measured, so the separation of one journal's cached prompt state from another's is only as good as the portal's answer.[2] withdrawn access can still be honoured for up to about 40 minutes by an open channel, and a journal may reuse an admitted channel for up to two minutes. the service offers no anonymity from us: no relay hides the owner's network address from the engine, and the credential is not blinded, as it is in designs that pair an outside relay with anonymous tokens (appendix e).[10]
11. **the engine counts per journal, and reports engine-wide figures.** it keeps an in-memory count of how much it has processed per journal, for capacity management. the count is unlinkable outside the engine and never leaves it, but it exists. engine-wide totals and a monthly spread of journals' use, both derived from those counts, leave the engine in the health report (§ 5.3): the spread is held back below ten journals a day, but the totals, from which a day's figures follow by subtraction, are not, so on a day when one journal was the only one to use an engine, the report carries that journal's totals for the day, without saying whose they are.

12. **the evidence plane has one custody boundary.** the transparency root is a single key (threshold 1 of 1), and it and every role key sit inside one sol pbc-controlled custody boundary. someone who breached that boundary could sign false records that verify. the root mirrors we publish, all of them ours, would show a replaced root, not records signed with the existing keys. there is no append-only log and no outside witness of what we publish, so detection depends on someone keeping their own copies.[6] the Secure Boot signing key sits in the same boundary; a breach there alone would not admit a different boot file, because the journal checks the boot file's own measurement.
13. **the journal's own records have gaps.** see § 9.4 and appendix d: some releases with no record, one audit whose verdict stands at fail, some records that leave out a platform, a run of earlier windows builds outside their records, and one release audited only by the step that published it. the audits are internal. journal builds are also not reproducible, so an owner cannot confirm from the installed binary alone that it carries the pins in the public source at its tag.
14. **protection depends on the installed release, and on the oldest image it pins.** an owner gets the checks in the release they are running, and we do not know which releases owners run. a journal admits every image its release pins. a current release pins two: the serving image and the one before it, whose engine stays in service while owners update. that earlier image lacks two fixes the serving image carries, one in the kernel's network code and one in how the model server loads media (appendix d). so a journal on a current release may be served by an engine that has both defects, until that engine is retired. retiring an image's engine stops us serving it, but the pins still describe a published image, so someone who can redirect the service name could run a copy of it (§ 3.2), and a journal that pins it stays exposed to whatever that image lacks until a release drops the pins. the two images before those have a further gap: the boot disk's partition table was read at boot but not measured, so whoever controlled the disk could add a swap partition and have parts of the engine's memory written to it, with every quoted register unchanged. for an owner, that gap ends when their journal runs a release that no longer pins either of them. appendix d says which releases pin which image, and what the oldest releases do not check. the security notice in item 4 does not reach the oldest releases: they predate it.
15. **evidence is not tied to a place or to our machines.** the journal checks what a machine is and what it booted, not where it is or whose it is. it accepts any processor of an accepted family whose evidence verifies, so an adversary who has such a machine under their physical control and can read or alter its memory, or who holds a signing key extracted from one, could present evidence that passes without being in Azure or running our build (§ 3.2). published attacks have read and altered the memory of confidential virtual machines of this class with equipment attached to a server's memory bus (appendix e). the GPU's evidence does not close this, because it names no machine (item 7). what stands in the way is physical control of the machines, which is Microsoft's for ours and no one's for a machine an attacker owns.[11]
16. **some inputs sit outside the quote.** the quote covers fourteen of twenty-four registers. this paper does not account, register by register, for the ten it leaves out, and we have tested little of what can reach the cloud's serial console (§ 5.2).[7] after boot the engine also takes in things no register covers: its network configuration, DNS answers, the time, our portal's answers and the certificates and status it fetches. the partition table was an input of this kind until the image before the serving one pinned it (appendix d).

17. **the shape of the traffic is visible.** answers can be streamed, and the service does not pad them. published research has recovered topics and text from the sizes and timing of encrypted, streamed model output (appendix e). anyone on the network path sees those sizes and timings.[12]
18. **the key agreement is not post-quantum.** the journal's TLS uses classical elliptic-curve key agreement. traffic recorded today could be decrypted by someone who later has a large enough quantum computer.[8]
19. **availability is not protected by attestation.** anyone who can reach the engine can make it collect evidence, which is slow; admission can be crowded out. the service also stops if NVIDIA's status service is unreachable for long enough. an evidence collector that cannot be cleaned up after a failure leaves the engine refusing every new channel until it is replaced.

notes: planned work

a note here says what we plan for a limit above or a gap named earlier, not what the system does. the body of this paper describes only what is built. when an image or a journal release carries one of these, its note is removed, the limit is rewritten or removed, and appendix d records the change. "for the next image" marks a change to the image; "with the next image" marks a change to the journal that ships in the release pinning it.

1. planned for the next image: the kernel command line requests lockdown explicitly, and the image's qualification confirms it is in force.
2. planned for the next image: the prompt cache is kept per channel, under a name our portal cannot compute, and is released when the channel ends. the image cache is treated the same way or turned off. releasing memory is not erasing it.
3. planned with the next image: the journal checks more of the configuration the processor and Azure sign. also planned for that image's qualification: we collect Microsoft's certificate for the attestation key, and its chain, to establish what it certifies. collecting it adds no check, and whether the journal should require it is not decided.
4. planned with the next image: its qualification reads the level its hosts report, and the journal release that pins it raises the minimum to that level when it is higher.
5. planned with the next image: the journal accepts only the processor family our engines use.
6. planned, with no date: we publish release records to an append-only log run outside sol pbc, and the journal then checks that log's proof before it sends.
7. planned for the next image's qualification: we record what the ten registers outside the quote hold and what reaches the serial console, and this paper then accounts for them.
8. planned, with no date: hybrid post-quantum key agreement for the journal's TLS.
9. planned, with no date: the engine keeps a current copy of AMD's signed revocation list and presents it with its evidence, and the journal verifies it on each connection, so the journal asks no other party.
10. planned, with no date: a relay outside sol pbc, and a credential that does not identify the owner. neither is designed yet.
11. planned, with no date: evidence that ties a machine to sol pbc. how is not designed yet.
12. planned, with no date: padding for streamed answers, after we measure what this service's traffic shows.

12. where we would most value critique

1. is the boundary between what the journal derives (the application registers) and what it accepts on vendor signatures (the platform registers, the paravisor) drawn in the right place, and what would you pin that we do not?
2. what independent check would most constrain a compelled image and pin update: an outside rebuild, a third-party co-signature on pins, a public log of pin changes that journals consult, or something else?
3. does binding the TLS session to the verified machine through a second, exporter-bound quote close the relay question, or is there a gap we have not seen?
4. which of the limits in § 11 would you fix first?
5. is "set up to write nothing to storage and to send it nowhere else", with the properties in § 5.2 and § 5.3, the strongest honest description of non-retention for a design like this, or is there a stronger one we could earn?

write to security@solpbc.org. that address is also where to report a security problem in this service.

appendix a. glossary

term	meaning here
AMD SEV-SNP	AMD's Secure Encrypted Virtualization with Secure Nested Paging: hardware memory encryption and integrity protection for a virtual machine, with a signed attestation report
attestation	signed hardware evidence about a machine's state, checked by a verifier; here, the mechanism the journal checks
confidential computing mode	an NVIDIA H100 mode in which the GPU's protected memory is fenced off from the host, transfers between the machine and the GPU are encrypted, and the GPU can attest to its firmware state
DSSE	Dead Simple Signing Envelope, the signature wrapper used for the transparency records
dm-verity	a linux mechanism that checks every block read from a read-only filesystem against a hash tree whose root is fixed in advance
engine	the confidential virtual machine that runs the model
exporter proof	a second TPM quote that binds a value derived from the live TLS session, proving the session ends inside the verified machine
fingerprint pin	a SHA-256 over the whole set of TPM registers the quote covers, compiled into a journal release
GPT	GUID partition table: the table at the start and end of a disk that lists its partitions
HCL report	Azure's attestation report wrapper, holding the AMD report and runtime data that names the virtual TPM's key
ID key digest	a field in the AMD report fingerprinting the key that signed the launch's ID block: the expected launch measurement and policy of the firmware the virtual machine was launched with
journal	the owner's solstone journal: the software and the memory it keeps, on the owner's device
nonce	a fresh random value from the owner's device, used once, so evidence cannot be replayed
paravisor	Microsoft's firmware layer inside an Azure confidential virtual machine; it implements the virtual TPM
PCR	platform configuration register: a TPM register that accumulates measurements of what loaded
quote	a TPM-signed statement of register values plus caller-supplied data
RA-TLS	remote-attestation TLS: TLS whose server certificate carries attestation evidence bound to the TLS key
serving image	the newest disk image in service, built from a named public commit
TCB	the processor's trusted computing base; in the AMD report, the security version numbers of its firmware components
spp	this service's identifier in our repositories, file names and transparency paths; it appears in links and commands, not in prose
TUF	The Update Framework: a design for signed, role-separated, freshness-checked repository metadata
UKI	unified kernel image: one signed boot file holding the kernel, initial ramdisk and kernel command line

term	meaning here
VCEK	AMD's per-chip, per-firmware-level key that signs the processor's attestation report
vTPM	the virtual TPM inside the confidential virtual machine

---

appendix b. current values and evidence index

what this version was checked against.

serving image	`solpbc/conf-proc` at commit `1630bf441eea77e7e512742abcfeba65ca2fb977`; its version on the evidence site is `1630bf4`
its signed record	SHA-256 `41fe733c0fa8424227588b3d68aed2f77afc3db85675d1c8da3dba28bd651761` (T3)
the image before it, still in service	`solpbc/conf-proc` at commit `9968c210d94d8e41a35d73ea4e4d885e46ba5e96`, version `996 8c21`; its signed record is SHA-256 `2a210cf54c928325f6de22747935e1073c3f67ebbcf6df85 952985185fb8d5f7` (T11)
journal release	2.0.82, `solpbc/solstone-journal` tag `v2.0.82`, commit `02de4257a77da12e605324cf5934 40642bd842bc`: the newest release when this version was checked, and its internal audit had returned
its signed record	SHA-256 `abe84fc04514fb5a1504daa032b0e0fef7de35107040dc82d3ccb6dcb70cb7c4` (T4)
fingerprint pins	`8391a8217575b2c33f063565a2c60fae9c7b5a7d54c852e4bc4223eac3a6dfb9` for the serving image; `8779bff6e5cd2c9eadc41098fb073a425654bc10d505b4257125b750cd25eebf` for the image before it
register values the journal pins for the serving image	register 4 `b0d2d394…`, register 7 `6ea86642…`, register 9 `99c49cb4…`, register 11 `82faceec…`, register 15 `ee22ecf6…`; in full at J1 and in the image's record. the earlier image's values are at J1 too
minimum processor firmware level	the four values are at J26

links into code and records are commit-pinned or content-addressed, so they keep pointing at what this version describes. T1, T2, T6, T9, T10 and T11 are live pages.

the journal's verifier, `solpbc/solstone-journal` at `02de4257a77da12e605324cf593440642bd842bc` (abbreviated J):

ref	what	link
J1	pins: both fingerprints, firmware-signer key, register list, each image's register values, production policy	pins.rs L13–147
J2	per-register appraisal	pcr_appraisal.rs
J3	processor report policy (VMPL, debug, ID key)	snp.rs L389–420
J4	processor leg, in order: runtime-data binding, AMD chain, policy, firmware minimum, TPM key binding, quote, pins	snp.rs L496–692

ref	what	link
J5	AMD chain to compiled-in roots; the certificate's firmware level read and matched to the report's	snp.rs L794–905
J6	TPM quote verification	tpm_quote.rs L164–325
J7	binding of nonce, TLS key and GPU evidence	binding.rs L10–59
J8	certificate evidence (first checks, then appraisal) and exporter proof	verify.rs L75–303
J9	TLS 1.3 configuration	channel.rs L236–307
J10	address walk, fresh nonce per address	channel.rs L852–1048
J11	first checks, proof request, appraisal, exporter proof and admission, in order	channel.rs L1120–1200
J12	credential and content only after admission	channel.rs L1391–1519
J13	GPU claims the journal requires	claims.rs L114–461
J14	NVIDIA reference manifests: one set and one status mode per fingerprint	rims.rs
J15	the pinned GPU verifier helper	nvattest_authority_v1.json
J16	channel reuse bound	pool.rs L14–30
J17	the set of register values chosen by fingerprint; none or two refuses	pcr_appraisal.rs L69–91
J18	fingerprint check, then that fingerprint's values (certificate quote)	snp.rs L647–672
J19	the same for the exporter-proof quote	verify.rs L278–302
J20	what stops the address walk: a failure on the owner's device, or the time budget	channel.rs L967–1048
J21	the first checks: one critical evidence extension, nonce and TLS key match	verify.rs L103–177
J22	the one fixed request sent before the appraisal, and the appraisal that follows it	channel.rs L1120–1143
J23	GPU profile chosen by the verified fingerprint, then the helper	production_verifier.rs L183–224
J24	the 130-second status margin at admission	channel.rs L53–127
J25	the signed security notice: what it is, what signs it, what it can do	advisory.rs L4–160
J26	the minimum processor firmware level	pins.rs L149–158
J27	the minimum applied, after the report policy	snp.rs L579–590
J28	the comparison, part by part	snp.rs L118–145
J29	one journal process holds the channel and carries the others' text-generation calls	broker.rs
J30	the linux automatic update, and its use of the security notice	automatic.rs

the serving image, `solpbc/conf-proc` at `1630bf441eea77e7e512742abcfeba65ca2fb977` (abbreviated I):

ref	what	link
I1	the build recipe	appliance/spp_appliance.py
I2	how to build it	appliance/BUILD.md

ref	what	link
I3	the kernel command line, with the verity root hash and the discovery switches	spp_appliance.py L647–666
I4	the boot-time verity setup	spp_diag_handoff.c L467–640
I5	inbound and outbound firewall (the IP-options drop is the first inbound rule), in-memory journal, crash dumps off	spp_appliance.py L112–160
I6	login doors removed	spp_appliance.py L909–915
I7	the register derivation	appliance/derive_pcrs.py
I8	channel lifetime; the engine carries no credential of its own	ratls_gateway.py L114–119
I9	the credential check with the services portal, and the journal identifier it returns	ratls_gateway.py L186–247
I10	the in-engine count	usage_tally.py L213–307
I11	GPU must report confidential computing on before serving	gpu-bringup.sh L11–37
I12	the expected partition entries, built and placed in the signed ramdisk	spp_appliance.py L1807–1822
I13	the partition-entry check in the boot program, before the root is found	spp_diag_handoff.c L1445–1532
I14	the gateway removes client labels and adds the journal's	ratls_gateway.py L714–723
I15	the model server's per-journal cache key	sglang_patches.py L20–26
I16	the health reporter: what it reads, where it sends, the status code only	spp_health_push.py L293–343
I17	the health report's fixed schema	health-push-contract.json
I18	engine-wide token totals, frozen at midnight UTC, and the monthly spread with its floor	fleet_usage.py L50–110
I19	the allowance, and what keeps it off	allowance.py L44–121
I20	the model server's loaders refuse media that is not carried in the request	sglang_patches.py L27–44
I21	the pinned kernel and NVIDIA driver packages	acquire_boot_inputs.py L30–54
I22	the build-time test that packets carrying IP options are dropped	check_egress_ruleset.py L74–121

the published record:

ref	what	link
T1	the plain-language trust site	trust.solstone.app
T2	the serving image's evidence page	the evidence page for commit 1630bf4
T3	the serving image's signed record, content-addressed	41fe733c….release-record.json
T4	journal 2.0.82's signed record, content-addressed	abe84fc0….release-record.json
T5	the TUF root, version 1	1.root.json
T6	the root mirror file	solpbc.org/transparency/tuf-root.txt

ref	what	link
T7	the root pinned in the verifier's repository	tuf-root.json at 6554388
T8	the verifier	solpbc/solstone-transparency at 6554388
T9	our privacy policy, confidential processing section	solpbc.org/privacy
T10	our articles of incorporation, Article 8	solpbc.org/articles
T11	the evidence pages of the three earlier images in appendix d	commit 9968c21, still in service · commit 8e4291d · commit 34a6c82

---

appendix c. how the statements in this paper were checked

we write this paper from the code and the live record, not from our internal summaries, and where the two disagree we follow the code. the checks below were run for this version on 18 november 2026 unless a different date is given.

- **the journal's verifier** was read at the 2.0.82 tag (commit `02de4257a77da12e605324cf593440642bd842bc`) by a review agent working only from the code. it compared every file appendix b cites with the same file at 2.0.80, the release the previous check was made at, and found four changed: the pins, the processor leg, the reference-manifest table and the security notice. two more that appendix b now cites (J29, J30) are new since 2.0.80, and it read those whole. it placed every line range again by its content and printed the first and last line of each at that commit, recorded each statement in §§ 6, 7, 8 and 10 as unchanged or changed with the line it rests on, and said what releases 2.0.81 and 2.0.82 each changed. it ran the verifier's test suites at that commit in a clean clone: the default command ran 237 tests, all passing, and with the optional full-test feature turned on, 293 tests, all passing. the windows verifier's suite, the mobile feature sets and the real GPU helper binary were not run. we then read the two pins, the firmware minimum and its place in the order of checks in the source ourselves.
- **the serving image** was read at its commit by a second review agent, which compared it file by file with the image before it, placed every line range appendix b cites again, and read the firewall rules, the five model-server patches and the kernel and driver pins. it did not build the image or start any cloud resource. what § 5.3 says about the model server's image cache rests on the model server's upstream source at the version whose patched files match our pinned digests, not on the built image.
- **the published record** was fetched fresh between 17:52 and 18:04 UTC on 18 november 2026. the TUF chain was verified with the published verifier against a root pinned out of band; a root with one changed byte was rejected. the root was version 1, key id `050ecd6f…`, with no version 2, and both mirrors matched it; the timestamp was version 252, expiring 25 november 2026. the serving image's record, the records of the three images before it, and journal 2.0.81's and 2.0.82's records were all accepted, and a request for a release with no record was rejected. all fourteen of the serving image's files were downloaded and matched their recorded lengths and digests. the register held records for 75 journal versions, the newest 2.0.82.
- **the service name** `processing.solstone.app` was read at both of its authoritative name servers at 17:52 UTC on 18 november 2026, and each returned two addresses. our operating record says the second, for the engine that runs the serving image, was added at 17:46 UTC that day.

- **the application registers** of the serving image were recomputed by running the image repository's derivation program, at the image's commit and with its pinned dependency, on the downloaded boot file. the values it derived for registers 4, 9, 11 and 15 matched the published record. the fingerprint the journal pins was recomputed from the published expected registers and equals the value in the journal's source; the same construction on the earlier image's published registers gave that image's pin. a boot file with one changed bit derived a different register 4.
- **audit results** are quoted from the audit records with their dates.
- **owner-visible text** quoted in § 10 was compared with the journal's source at the 2.0.82 tag and matches it word for word.
- **the allowance switch** in § 5.3 was read in the services portal's source, not on the deployed portal, and was not read again for this state.

for an earlier version of this paper we ran two AI reviews separately, each given only the paper and public documentation, and asked to check its statements about AMD, Microsoft, NVIDIA, TPM, systemd, TLS and TUF behaviour against public sources. we checked each of their findings against the code before changing the paper.

these statements rest on our own testing or records rather than on anything you can check without us: the qualification results (that each published image served, reproduced, and refused to serve from a corrupted disk; that an earlier image did not execute the cloud's run-command and extension features; that the serving image kept each journal's prompt cache separate on the hardware, and that it refused media sent as an address), our test that the machine's software cannot extend register 22, the observation that every Azure report we hold carries the firmware signer the journal pins, the internal audit results in § 9.4 and appendix d, the operating history in appendix d (when each image served and was retired, and when each release was published), and what our services portal and our monitoring keep, accept and delete (§ 4.1, § 5.1, § 5.3), the rotation ledger in § 8.2, the serial output in § 5.2, and the internal audit of the serving image's publication (§ 9.3). the qualification results, the register 22 test, the firmware-signer observation and the audits are marked where they appear; the rest are marked here.

---

appendix d. changelog

the body of this paper describes the service as it is now. this appendix records how it got here: what each serving image changed, which journal releases changed what the journal checks, which releases admit which image, and where the journal's own records have gaps. a fix is recorded here once an image or a release carries it.

d.1 serving images, newest first

images are named by the `solpbc/conf-proc` commit they were built from. each of the four images has a signed record on the evidence site; the first engine has none.

image	served	what it changed
`1630bf4`, the serving image	from 18 november 2026	**a security fix for two defects in the image before it.** it runs a kernel with the fix for CVE-2026-98432, a flaw in how the linux kernel handles incoming connection requests that carry IP options. anyone able to send such requests to an engine could in principle have run their own code on it; the published exploit relies on MPTCP, a network feature the gateway does not use. the image's firewall also drops those packets before they reach the gateway. and its model server refuses images, audio or video sent as a web address or a file path (CVE-2026-15974, in the SGLang model server). on the image before it, a request from a journal with a valid credential can make the model server open a file its own unprivileged user can read, or fetch from the engine's own loopback services. we found that the second defect reached us while preparing this paper, by reading the published advisory against the image's source, not on a running machine. the engine keeps no record of what requests contained or where connections came from, so it cannot show whether either defect was used. the fixed kernel needs NVIDIA driver 595.91.07, so the driver and its reference manifest moved with it
`9968c21`	from 30 october 2026, alone from 16 november, beside the serving image from 18 november 2026	has the two defects the serving image fixes, for as long as its engine is in service. when it replaced the image before it, it **closed a gap in what the check covered.** before it, the boot disk's partition table was read at boot but not measured, and automatic partition discovery would switch on any swap partition it found, so whoever controlled the boot disk could add one and have parts of the engine's memory written to it with every quoted register unchanged. we found this while writing the first version of this paper, by reading the source, not on a running machine. this image turns partition and swap discovery off on the measured command line, and its measured boot program refuses to boot unless the disk's partition entries match the copy inside the signed boot file. it also stops the model and transcription services reaching the gateway's own loopback ports, filters inbound traffic, keeps the prompt cache per journal instead of shared, and bounds the time evidence collection may take. it adds the health report (§ 5.3), a new outbound flow, and an allowance mechanism that is built and switched off
`8e4291d`	from 7 october 2026, alone from 9 october, beside `9968c21` from 30 october, until 16 november 2026	serves the full 262,144-token context the journal expects, and made the disk image reproducible byte for byte. the disk image its engine was created from matched the digest in its record, with no added partition; we could not read the disk under the running engine from outside, so we cannot rule out that one was ever added
`34a6c82`	6 october 2026, for about five and a half hours	the first image of the design this paper describes to serve: a verified read-only root, one signed boot file, no way to log in, a fourteen-register quote, and NVIDIA's signed status carried by the engine so the journal contacts no one. withdrawn the same day because it served a shorter context than journals expect
the first engine, before the verified-root design	from july 2026 until 9 october 2026	the engine that predated the verified-root image. its quote covered ten registers. it was deleted on 9 october 2026

an Azure firmware update in september 2026 moved registers 0 and 2 with nothing in our image changing. `8e4291d` was accepted in both firmware states; the two images after it have each been qualified, and are pinned, in the later one only.

d.2 journal releases that changed what the journal checks

release	change
2.0.82, 17 november 2026	admits the serving image `1630bf4` beside `9968c21`, with its own register values and its own set of NVIDIA reference manifests, each chosen by fingerprint. adds the minimum processor firmware level for either image: the level in the chip's certificate must equal the level the report states and be at or above a compiled-in minimum (§ 7). a linux install's automatic update now reports a failure when no release on offer carries the fix the security notice names. on windows, the journal's own processes pass text-generation calls over a pipe only the owner's account can open
2.0.81, 16 november 2026	no change to any pin or check. on linux and macos, one journal process now holds the admitted channel and carries the other processes' text-generation calls over a local socket (§ 7, reuse)
2.0.75, 11 november 2026	desktop journals begin fetching the signed security notice (§ 11, item 4). no change to the verifier
2.0.74, 10 november 2026	on android only, the verifier helper's files are read and hashed once per operation instead of twice. no change to what is checked about the engine
2.0.65, 1 november 2026	drops the pins of image `8e4291d`: from this release the journal admits only `9968c21`. the journal now sends its request for the session proof after the first checks on the certificate and before the processor, TPM and GPU appraisal (§ 7, step 4); every check still gates admission
2.0.64, 1 november 2026	no change to any check. the journal's channel code changed: a refused attempt's error now carries a fixed-form summary of how far the check got
2.0.62, 30 october 2026	admits image `9968c21` beside `8e4291d`, with a separate set of register values for each, chosen by fingerprint
2.0.41, 11 october 2026	drops the first engine's pin
2.0.38	adds the register-list check and the per-register comparison of § 6
2.0.35, 7 october 2026	admits image `8e4291d`; drops `34a6c82`
2.0.34	adds the firmware signing key pin
2.0.33	admits image `34a6c82`

release 2.0.77 was never published. no release after 2.0.65 and before 2.0.82 changed a pin.

d.3 which release admits which image

journal release	1630bf4 (serving)	9968c21	8e4291d	34a6c82	first engine	checks
2.0.82 and later	yes	yes	no	no	no	everything in § 7

journal release	1630bf4 (serving)	9968c21	8e4291d	34a6c82	first engine	checks
2.0.65 to 2.0.81	no	yes	no	no	no	everything in § 7 except the two checks of the processor's firmware level
2.0.62 to 2.0.64	no	yes	yes	no	no	as above, in the earlier order: session proof requested after the appraisal
2.0.41 to 2.0.61	no	no	yes	no	no	as above, with the values of `8e4291d` only
2.0.38 to 2.0.40	no	no	yes	no	no	as above. these releases still list the first engine's pin, and refuse its quote by the register-list check
2.0.35 to 2.0.37	no	no	yes	no	yes	everything except the two firmware level checks, the register-list check and the per-register comparison
2.0.33 and 2.0.34	no	no	no	yes	yes	2.0.34 has the firmware signing key pin; 2.0.33 does not
2.0.32 and earlier	no	no	no	no	yes	no firmware signing key pin

what this means for an owner today: releases 2.0.82 and later may be served by either engine. releases 2.0.62 to 2.0.81 are served only by the engine that runs `9968c21`, and will find no engine of ours once it is retired. releases 2.0.61 and earlier accept only images whose engines are retired, so they find no engine of ours and send nothing. every release from 2.0.62 on, the current one included, holds the pins of `9968c21`, so the two defects in d.1 stay open for an owner until their journal runs a release that drops those pins (§ 11, item 14); no such release exists on the state-as-of date. every release from 2.0.33 to 2.0.64 still holds the pins of `34a6c82` or `8e4291d`, neither of which has the partition check, so the partition gap in d.1 stays open for an owner on one of them until they update to 2.0.65 or later. releases 2.0.37 and earlier also accept the first engine's measurements, which predate this design; we have not established what protection that leaves, and an owner on one should treat it as none. the solstone phone apps carry their own copy of the pins, and admit a new image only after their own update.

d.4 gaps in the journal's own records

read from the live register on 18 november 2026, which held records for 75 journal versions between 2.0.0 and 2.0.82, and from the audit records.

- **journal 2.0.38**'s record failed its audit on 9 october 2026, on three findings: the record did not declare a change in how one windows component was delivered; the record's format could only declare delivery-mode changes, not path, version or digest changes; and the public site had been deployed from a commit one behind, rolling back four pages.
- **journal 2.0.36**, **2.0.39** and **2.0.44** have no record. 2.0.39 was published on linux and macos on 10 october 2026 and replaced by 2.0.40 the same day; 2.0.44 was published on linux and macos on 13 october 2026.
- five audits, for 2.0.45, 2.0.53, 2.0.55, 2.0.56 and 2.0.60, did not pass when first closed, each on the criterion covering the public notes and records, and each passed on a dated amendment.

- some records leave out a platform that skipped the release: for 2.0.45, 2.0.52, 2.0.53, 2.0.55 and 2.0.60 the record covers linux and macos only, and for 2.0.61 and 2.0.74 it leaves out windows on arm64.
- windows journals from 2.0.21 to 2.0.35 were published outside their versions' records (2.0.21 and 2.0.25 were windows-only releases with no record at all).
- the only audit document for **journal 2.0.3** was produced by the same step that published it.
- we have not established whether two numbers missing from the register, 2.0.2 and 2.0.20, were ever published as journal releases. 2.0.77 was not.
- every release from 2.0.64 to 2.0.82 has a record, and each record's audit passed when first closed, with no findings.

d.5 versions of this paper

version 2.0 is the first public version. versions 1.0 to 1.4.1 were written between 10 october and 7 november 2026 and shared privately for review. they described the system at earlier journal releases and images, and their release-by-release detail is what this appendix now carries.

---

appendix e. related work and references

this design builds on other people's work, and other services have published designs for the same problem. this appendix places ours beside them and lists what we cite. every reference was fetched and its title checked on 16 november 2026; the comparison in e.2 was read again on 18 november 2026, and references 62 to 66 were added then. a description of another service is our reading of what that service has published, not a test of it.

e.1 the vocabulary and the channel

the roles used in § 7 (attester, verifier, relying party, endorser, reference value) are those of the IETF's remote attestation architecture [1]. carrying evidence in a TLS certificate goes back to Knauth and others [2]. binding evidence to a live session through the TLS exporter [3] [4] is the subject of current IETF work on attestation with exported authenticators [5] [6]. a formal analysis of TLS designs that carry attestation found identity and diversion problems in several deployed ones [7]; ours has had no formal analysis. the transparency records use TUF [8] with in-toto statements [9] in DSSE envelopes [10]. the register layout follows the TCG firmware profile [11], systemd's measurements [12] and the Linux registry of register use [13]; the root filesystem is verified with dm-verity [14]. AMD documents the report format [15] and the chip certificate [16], Microsoft documents the Azure report and its runtime data [17], the machine size [18] and its paravisor [19], and NVIDIA describes the GPU's confidential mode and its memory [20] [21].

e.2 other services

services whose client checks hardware evidence. Apple's Private Cloud Compute has a device release a request only to nodes whose software appears in an append-only log, routes requests through an outside relay with unlinkable tokens, and records each node's identity in that log, which the device checks before it trusts the node [22] [23] [62]. Meta's Private Processing for WhatsApp is the closest published design to ours in hardware (AMD SEV-SNP with NVIDIA confidential GPUs): its client carries a minimum firmware level, AMD revocation lists and an expiry for each accepted artefact, checks its measurements against a log

kept by an outside party (a signed log, which Meta says is not a Merkle-tree ledger), reaches the service through an outside relay with anonymous credentials, and turns the persistent prompt cache off [24]. two outside reviews of it are public [25] [26]. Trail of Bits reported 28 issues, eight of them high severity, including inputs read after measurement and a firmware level taken from the report and not from the chip's certificate [25]; that list is the best public checklist we know for a design like ours, and this paper's limits should be read against it. Google's Private AI Compute [27] has a public assessment by NCC Group [28]. Tinfoil [29] and Privatemode [31] publish open, reproducible images and client-side verification; Privatemode's client enforces a minimum firmware level and an allow-list of machines, from a manifest it fetches [33] [65], and Tinfoil's fetches the measurements to accept on each connection [64], and both key the prompt cache with a secret the client holds [30] [32]. the dstack framework [34] has a public audit [35]. Marlinspike's description of Confer [36] and the design paper from Pattern Labs and Anthropic [37] cover the same ground.

where ours stands. like Apple's and Meta's clients, the journal checks against values it already holds and fetches nothing at the moment of the check; it appraises the GPU's evidence on the owner's device too. each of the designs above ties the encrypted session to the machine it verified through a key named in the evidence [24] [29] [62] [65]; ours does the same and adds a second quote over a value derived from the live session (§ 7). against that, the designs above have things ours lacks: AMD revocation checks, evidence tied to the operator's machines, a public log the client enforces, an outside relay and anonymous credentials, a bounded or client-keyed prompt cache, and published outside review. § 11 says so item by item. Privatemode's reviews are available on request [66]. the Trail of Bits review of Meta's design is public in full [63] and records findings that were open when it was written.

promises made by contract. several large model providers offer zero data retention as an agreement, granted on request and with stated exceptions. it rests on the provider's word and gives the customer nothing to verify. Knodel and others set out why that differs from end-to-end encryption and what confidential hardware can and cannot add [38].

e.3 published attacks on this class of system

naming a class here says that the class exists and that a reader should weigh it. it says nothing about whether a particular attack applies to this service.

- **physical access to a server's memory.** interposers on the memory bus, and tampered memory modules, have been used to read and alter the memory of confidential virtual machines, to extract keys, and to pair forged processor evidence with a borrowed GPU's evidence [39] [40] [41] [42]. § 11, item 15.
- **a malicious host attacking AMD SEV-SNP in software.** published attacks have corrupted or disclosed guest state through cache resets, injected interrupts and the tables that protect guest memory [43] [44] [45] [46].
- **the virtual TPM and the cloud's own trust relationships.** independent work has examined how much a cloud's confidential virtual machine asks its customer to take on trust, and how a virtual TPM can be isolated from the host [47] [48]; one proposal addresses proving that a machine is in a given data centre [49]. § 11, items 5 and 15.
- **the GPU.** an independent security study of NVIDIA's confidential GPU design sets out what its evidence does and does not bind [50]. § 11, item 7.

- **the serving software.** a model server that every journal shares is ordinary software and can have ordinary defects. hardware isolation does not protect one journal from another through it. § 11, item 1.
- **shared prompt caches.** timing differences in shared caches have been used to recover other people's prompts [51] [52] [53], which is why our prompt cache is kept per journal; the image cache in § 5.3 is not. § 5.3 and § 11, item 2.
- **the shape of encrypted, streamed output.** the sizes and timing of streamed model output have been used to recover its text and topic [54] [55]. § 11, item 17.

e.4 mechanisms we do not have

an append-only log with outside witnesses [56] [57], a relay that hides the requester's address [58] with anonymous tokens [59], and post-quantum key agreement for TLS [60] are specified and in use elsewhere. we use the word "reproducible" in the sense of [61], and the summary says how far our own reproduction goes.

e.5 references

1. H. Birkholz and others. *Remote ATtestation procedureS (RATS) Architecture.* RFC 9334, 2023. https://www.rfc-editor.org/rfc/rfc9334
2. T. Knauth and others. *Integrating Remote Attestation with Transport Layer Security.* arXiv:1801.05863, 2018. https://arxiv.org/abs/1801.05863
3. E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3.* RFC 8446, 2018 (since replaced by RFC 9846). https://www.rfc-editor.org/rfc/rfc8446
4. S. Whited. *Channel Bindings for TLS 1.3.* RFC 9266, 2022. https://www.rfc-editor.org/rfc/rfc9266
5. N. Sullivan. *Exported Authenticators in TLS.* RFC 9261, 2022. https://www.rfc-editor.org/rfc/rfc9261
6. M. U. Sardar and others. *Remote Attestation with Exported Authenticators.* IETF Internet-Draft draft-fossati-seat-expat-05, november 2026. https://datatracker.ietf.org/doc/draft-fossati-seat-expat/
7. M. U. Sardar and others. *Identity Crisis in Confidential Computing: Formal Analysis of Attested TLS.* ACM ASIA CCS 2026. https://doi.org/10.1145/3779208.3785387
8. J. Cappos and others, editors. *The Update Framework Specification.* https://theupdateframework.github.io/specification/latest/
9. *in-toto Attestation Framework.* https://github.com/in-toto/attestation
10. *DSSE: Dead Simple Signing Envelope.* https://github.com/dsse-lf/dsse
11. Trusted Computing Group. *TCG PC Client Platform Firmware Profile Specification*, version 1.06, 2023. https://trustedcomputinggroup.org/resource/pc-client-specific-platform-firmware-profile-specification/
12. *TPM2 PCR Measurements Made by systemd.* https://systemd.io/TPM2_PCR_MEASUREMENTS/
13. UAPI Group. *UAPI.7 Linux TPM PCR Registry.* https://uapi-group.org/specifications/specs/linux_tpm_pcr_registry/
14. *dm-verity.* The Linux kernel documentation. https://docs.kernel.org/admin-guide/device-mapper/verity.html
15. AMD. *SEV Secure Nested Paging Firmware ABI Specification.* publication 56860. https://www.amd.com/content/dam/amd/en/documents/developer/56860.pdf

16. AMD. *Versioned Chip Endorsement Key (VCEK) Certificate and KDS Interface Specification.* publication 57230. https://docs.amd.com/v/u/en-US/57230
17. Microsoft. *Azure Confidential VM guest attestation design detail.* https://learn.microsoft.com/en-us/azure/confidential-computing/guest-attestation-confidential-virtual-machines-design
18. Microsoft. *NCCads_H100_v5 size series.* https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/gpu-accelerated/nccadsh100v5-series
19. Microsoft. *OpenHCL: the new, open source paravisor.* 2024. https://techcommunity.microsoft.com/blog/windowsosplatform/openhcl-the-new-open-source-paravisor/4273172
20. E. Apsey and others. *Confidential Computing on NVIDIA H100 GPUs for Secure and Trustworthy AI.* NVIDIA technical blog, 2023. https://developer.nvidia.com/blog/confidential-computing-on-h100-gpus-for-secure-and-trustworthy-ai/
21. G. Dhanuskodi and others. *Creating the First Confidential GPUs.* ACM Queue 21(4), 2023. https://doi.org/10.1145/3623393.3623391
22. Apple. *Private Cloud Compute: A new frontier for AI privacy in the cloud.* 2024. https://security.apple.com/blog/private-cloud-compute/
23. Apple. *Expanding Private Cloud Compute.* 2026. https://security.apple.com/blog/expanding-pcc/
24. Meta. *Private Processing for WhatsApp Overview: Technical White Paper and Security Guide.* version 3, 20 october 2026. https://ai.meta.com/static-resource/private-processing-technical-whitepaper
25. Trail of Bits. *What we learned about TEE security from auditing WhatsApp's Private Inference.* 7 april 2026. https://blog.trailofbits.com/2026/04/07/what-we-learned-about-tee-security-from-auditing-whatsapps-private-inference/
26. NCC Group. *Public Report: Meta Whatsapp Message Summarization Service.* 2025. https://www.nccgroup.com/research/public-report-meta-whatsapp-message-summarization-service/
27. Google. *Google Private AI Compute: Enabled with Secure Server-Side Memory.* september 2026. https://services.google.com/fh/files/misc/private_ai_compute_technical_brief.pdf
28. E. Anderson and others, NCC Group. *Private AI Compute in the Cloud: Assessment Overview.* version 1.4, 2025. https://www.nccgroup.com/media/cbuhuhuu/ncc_group_google_e025353p_report_2025-11-08_v14.pdf
29. Tinfoil. *Backend infrastructure.* https://docs.tinfoil.sh/verification/attestation-architecture
30. Tinfoil. *Prompt caching.* https://docs.tinfoil.sh/sdk/prompt-caching.md
31. Privatemode. *Security.* https://docs.privatemode.ai/security/
32. Privatemode. *Prompt cache security.* https://docs.privatemode.ai/security/prompt-caching/
33. Privatemode. *The manifest.* https://docs.privatemode.ai/security/attestation/manifest/
34. S. Zhou and others. *Dstack: A Zero Trust Framework for Confidential Containers.* arXiv:2509.11555, 2025. https://arxiv.org/abs/2509.11555
35. zkSecurity. *Audit of dstack.* 2025. https://reports.zksecurity.xyz/reports/phala-dstack/
36. M. Marlinspike. *Private inference.* 2026. https://confer.to/blog/2026/01/private-inference/
37. Pattern Labs and Anthropic. *Confidential Inference Systems: Design principles and security risks.* 2025. https://www-cdn.anthropic.com/8bdb51c584ff7c2eaa286e44311b0a45b57c3ec4.pdf

38. M. Knodel and others. *How To Think About End-To-End Encryption and AI: Training, Processing, Disclosure, and Consent.* arXiv:2412.20231, 2024. https://arxiv.org/abs/2412.20231
39. J. Chuang and others. *TEE.fail: Breaking Trusted Execution Environments via DDR5 Memory Bus Interposition.* IEEE S&P 2026. https://tee.fail/
40. J. De Meulemeester and others. *Battering RAM: Low-Cost Interposer Attacks on Confidential Computing via Dynamic Memory Aliasing.* IEEE S&P 2026. https://batteringram.eu/
41. J. De Meulemeester and others. *BadRAM: Practical Memory Aliasing Attacks on Trusted Execution Environments.* IEEE S&P 2025. https://badram.eu/
42. A. Seto and others. *WireTap: Breaking Server SGX via DRAM Bus Interposition.* ACM CCS 2025. https://doi.org/10.1145/3719027.3765204
43. R. Zhang and others. *CacheWarp: Software-based Fault Injection using Selective State Reset.* USENIX Security 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/zhang-ruiyi
44. B. Schlüter and others. *WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP.* IEEE S&P 2024. https://arxiv.org/abs/2404.03526
45. B. Schlüter and others. *HECKLER: Breaking Confidential VMs with Malicious Interrupts.* USENIX Security 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/schl%C3%BCter
46. B. Schlüter and S. Shinde. *RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP.* ACM CCS 2025. https://doi.org/10.1145/3719027.3765233
47. J. Eisoldt and others. *SoK: A cloudy view on trust relationships of CVMs.* arXiv:2503.08256, 2025. https://arxiv.org/abs/2503.08256
48. V. Narayanan and others. *Remote attestation of confidential VMs using ephemeral vTPMs.* ACSAC 2023. https://arxiv.org/abs/2303.16463
49. F. Rezabek and others. *Proof of Cloud: Data Center Execution Assurance for Confidential VMs.* arXiv:2510.12469, 2025. https://arxiv.org/abs/2510.12469
50. Z. Gu and others. *Blueprint, Bootstrap, and Bridge: A Security Look at NVIDIA GPU Confidential Computing.* arXiv:2507.02770, 2025. https://arxiv.org/abs/2507.02770
51. G. Wu and others. *I Know What You Asked: Prompt Leakage via KV-Cache Sharing in Multi-Tenant LLM Serving.* NDSS 2025. https://www.ndss-symposium.org/ndss-paper/i-know-what-you-asked-prompt-leakage-via-kv-cache-sharing-in-multi-tenant-llm-serving/
52. C. Gu and others. *Auditing Prompt Caching in Language Model APIs.* ICML 2025. https://arxiv.org/abs/2502.07776
53. L. Song and others. *The Early Bird Catches the Leak: Unveiling Timing Side Channels in LLM Serving Systems.* arXiv:2409.20002, 2024. https://arxiv.org/abs/2409.20002
54. R. Weiss, D. Ayzenshteyn, G. Amit and Y. Mirsky. *What Was Your Prompt? A Remote Keylogging Attack on AI Assistants.* USENIX Security 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/weiss
55. G. McDonald and J. Bar Or. *Whisper Leak: a side-channel attack on Large Language Models.* arXiv:2511.03675, 2025. https://arxiv.org/abs/2511.03675
56. Sigstore. *Rekor.* https://github.com/sigstore/rekor
57. C2SP. *Transparency Log Witness Protocol.* https://github.com/C2SP/C2SP/blob/main/tlog-witness.md
58. M. Thomson and C. A. Wood. *Oblivious HTTP.* RFC 9458, 2024. https://www.rfc-editor.org/rfc/rfc9458

59. A. Davidson and others. *The Privacy Pass Architecture.* RFC 9576, 2024. https://www.rfc-editor.org/rfc/rfc9576
60. K. Kwiatkowski and others. *Post-Quantum Traditional (PQ/T) Hybrid Key Agreement Mechanisms for TLS 1.3.* RFC 10024, 2026. https://www.rfc-editor.org/rfc/rfc10024
61. *Definitions.* reproducible-builds.org. https://reproducible-builds.org/docs/definition/
62. Apple. *Private Cloud Compute Security Guide.* https://security.apple.com/documentation/private-cloud-compute/
63. Trail of Bits. *Meta WhatsApp Private Processing: Security Assessment with Fix Review.* 26 august 2025. https://github.com/trailofbits/publications/raw/master/reviews/2025-08-meta-whatsapp-privateprocessing-securityreview.pdf
64. Tinfoil. *How verification works in Tinfoil.* https://docs.tinfoil.sh/verification/verification-in-tinfoil
65. Privatemode. *Attestation: overview.* https://docs.privatemode.ai/security/attestation/overview/
66. Privatemode. *Trust and compliance.* https://docs.privatemode.ai/security/trust-and-compliance/

e.6 how to cite this paper

this paper changes when the system does, so cite the version you read.

```
@techreport{solpbc2026confidential,
  title       = {{confidential processing in solstone:
                  how your journal checks the service before it sends}},
  author      = {{sol pbc}},
  institution = {sol pbc},
  type        = {security whitepaper},
  number      = {version 2.0},
  year        = {2026},
  month       = nov,
  url         = {https://trust.solstone.app/static/confidential-processing-whitepaper.pdf}
}
```