Protecting proprietary website logic

Browser and server experiments on confidentiality and copying

Emil Valiyev

Abstract

Keeping a website's private calculations on an authorized server gave the strongest demonstrated protection for its source code in this study. I compared programs sent to a browser, encrypted program delivery, server execution, browser extensions, separate browser components, and an attempted protected-execution environment. Each experiment asked what could be copied, which operations remained allowed, and what the returned answers revealed. Programs delivered to the user's device remained recoverable in the tested cases. Preventing a secret key from being exported did not prevent recovery of the program decrypted with that key. Separate browser components hid programs from selected observers inside the page, but browser-owner access recovered them. Server checks rejected tested unauthorized requests, changed inputs, expired requests, and duplicates. A copied interface could still use a legitimate account, and saved answers remained available after the service stopped. Learning the calculation from its answers succeeded for a simple arithmetic function but remained inaccurate for a more complex decision model under the tested methods. The protected-execution attempt stopped at program loading, so hardware protection remained unverified. These results support keeping sensitive implementations on the server where practical, while evaluating permitted use, visible copying, and disclosure through answers separately.

1 Introduction

A website operator may want to keep an algorithm private, prevent unauthorized use of a service, or stop someone from reproducing its interface. These are different goals. A screenshot can copy a result without recovering the program that produced it. Saving a program can reveal how it works without revealing the server behind it. A different program can also reproduce the same answers without sharing the original source code.

The distinction between the browser and the server matters throughout this paper. The browser runs the page on the user's device. The server carries out operations that the website keeps outside the page. The frontend is the material delivered to the browser, including its visible interface; the backend is the service behind it. Protecting backend code does not automatically protect everything shown or returned to the frontend.

I argue for keeping proprietary website logic, meaning code and rules the operator wants to keep private, on an authorized server whenever practical. The tested server-only design withheld the private program and checked permission before execution. An authorized user could still use a copied interface, save answers, or try to learn the calculation. The question is what a design withholds, from whom, and under what conditions.

1.1 Contribution

The study's novelty is a comprehensive, comparative mapping of the client and server isolation methods tested across both phases. Isolation means separating a program or its data from a particular observer.

Implemented prototypes and countertests bring code transformation, encrypted delivery, server authorization, extension isolation, execution in the main browser process, and learning from answers into one experimental framework. This combined system-level analysis connects each protection boundary to the capabilities that cross it and the information that remains available. It separates source secrecy, permission checks, and disclosure through answers when judging combinations of methods.

1.2 Related work

Robots rules guide cooperating collection tools; servers need not refuse other visitors [1]. Browser rules restrict one website's access to another website's responses. Content Security Policy restricts scripts a page may run; Subresource Integrity checks their fingerprints [2-4]. Site isolation separates websites into browser processes [5]. These controls can work while the browser owner retains delivered resources.

WebAssembly is a compiled browser format; obfuscation changes a program's form to make it harder to read. Neither alone establishes secrecy from its recipient [6,7]. TrustJS studied a JavaScript engine inside Intel SGX, a protected execution environment. Later SGX research examined confidentiality failures in its trusted software [8,9]. DPoP binds authorized requests to a key; Paillier supports restricted arithmetic on encrypted numbers; model-extraction research studies learning from answers [10-12]. These works address particular mechanisms or attacks. This study compares their boundaries together and tests selected combinations, tracing source secrecy, permitted use, and observable answers.

2 Experimental method and scope

The research has two phases. Phase I tests basic protection methods with a small arithmetic program. Phase II adds more complex applications and stronger forms of browser access. Both phases distinguish copying visible material, recovering a program, recovering its internal rules, and reproducing its behavior. Phase II also examines unauthorized service use and whether a separate execution component protects its program.

An adversary model states what the person attempting extraction can control. Phase I defines five models; Phase II defines sixteen independent capability models. Table 1 maps each Phase II capability to a protection boundary and the observed result. Phase I uses separate identifiers. Across methods, I ask the same questions: what can the observer recover, which operations remain permitted, and what do their answers reveal? Each result applies to its tested capability.

The Phase I reference calculation was f(x) = (7x + 13) mod 1009 for integer inputs 0 through 99. The expression multiplies the input by 7, adds 13, and takes the remainder after division by 1009. Tests ran on owned loopback services, meaning services reached on the same computer, principally at ports 4381 and 4382. A port identifies a service address on that computer.

The host reported Windows 11 build 26200 on x64, an Intel Core i7-14650HX, 24 logical processors, and 16,760,619,008 bytes of physical memory. Phase I used Chromium reporting Chrome/155.0.0.0 and Waterfox reporting engine version 153.5.0, with geckodriver 0.37.1 for browser automation. Ghidra 12.1.4 examined compiled Windows code; WABT 1.0.42 examined WebAssembly. These tool and version details identify the tested environment, rather than guarantee results on other systems.

Agreement means agreement on the stated inputs. Matching all 100 arithmetic cases does not establish identical results for negative inputs or numbers large enough to overflow the

formats being compared. The tests used real browser engines, including headless Waterfox runs, which operate without a visible browser window. Different browser modes prevent a controlled product ranking. The public portfolio website was also one limited example, used to confirm that capture worked before testing specific protections locally.

Phase II added a decision model made of eight branching sets of rules, called decision trees. Each tree had depth four, describing the number of branching steps along a path, and the model accepted six integer inputs from 0 through 1,000. Its initial JSON representation, a text format for structured data, occupied 7,252 bytes. A routing task used a graph with 24 nodes and connections between them. Connection costs changed with congestion.

An inventory application covered 16 products. It tested reservations, ownership checks, conflicting versions, commits, cancellation, expiry, and idempotent retries. Here, idempotent means that repeating a commit with the same request identifier returns its recorded result instead of applying the change again. The principal Phase II browsers were Chrome for Testing 155.0.8059.84 with V8 15.5.35.25, and Waterfox product version 6.7.6 with engine version 153.5.0. The phases use different tasks and cannot be treated as interchangeable benchmarks.

I retained programs, browser results, request records, capture inventories, recovered files, and raw timing samples. The unchanged DoppelGNG.me tool captured frontend material and replayed it in a browser [13]. An authorized read-only capture of emilvaliyev.com provided the public baseline. Active countertests targeted the owned applications. I checked normal operation, preserved failures, and used positive controls: separate examples showing that an observation method worked before interpreting what it failed to see.

The live Phase I capture completed in 873.403 seconds with 19 routes and 110 resource records. It recovered delivered frontend material, without establishing backend recovery or complete interaction coverage. The controlled plain-page capture took 64.495 seconds, retained four resources, and checked the calculation control. Of the live records, 86 came from browser-observed responses and 24 from later fetches. I kept that distinction. Fingerprint checks of saved original and rewritten resources found zero mismatches across the five Phase I capture directories.

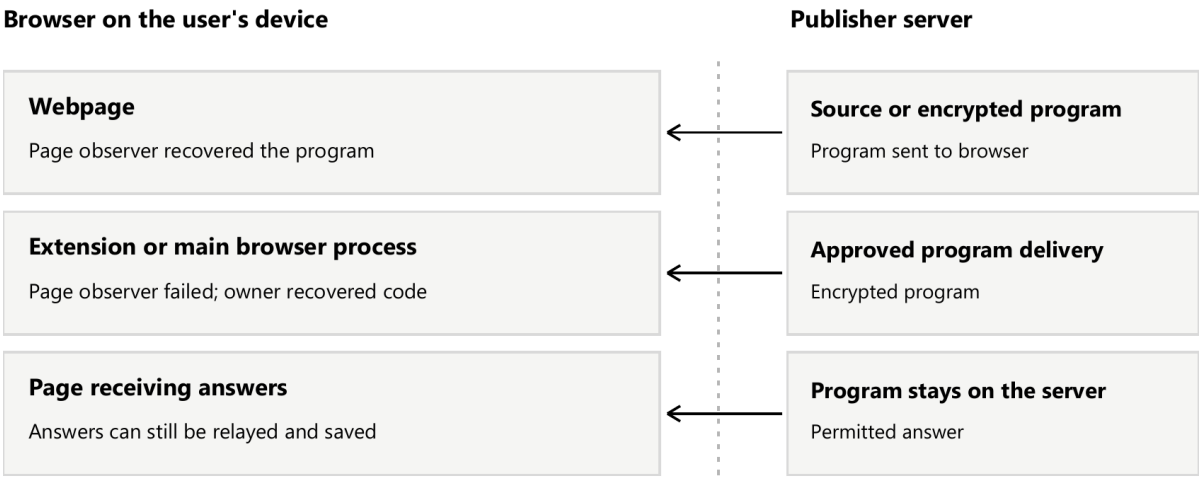


Figure 1. Server execution retained the program and returned answers. Extensions and the main browser process blocked selected page observers; browser-owner access recovered their code.

Table 1. Phase II capability-to-boundary matrix. A1-A16 retain the source report's identifiers. Outcomes describe the tested methods; untested attacks and unverified hardware protection remain explicit evidence gaps. The rows are independent capability models.

Capability	Protection boundary	Observed result and limit
A1 Resource collection	Server source	Server model and anonymous execution withheld; delivered assets remained collectable.
A2 Browser capture and replay	Complete application copy	Capture stayed partial; neither a complete clone nor resistance to copying was established.
A3 Authorized visitor	Source and permitted answers	Calculations and workflows worked; answers remained available without recovering server code.
A4 Browser and program inspection	Delivered and server code	Delivered implementations recovered; server-only model recovery was not demonstrated.
A5 Page observation code	Page to isolated program	Tested observation code missed extension and parent models; relaying through an authorized page remained possible.
A6 Extension with page access	Model secrecy and answer integrity	Observation code did not recover the model; messages allowed substitution of the page-visible answer.
A7 Debugging extension background code	Model in extension	Program state recovered and execution altered; the key's raw bytes still could not be exported.
A8 Browser debugging	Model in main browser process	Program state recovered and execution altered; refusal to export the key did not prevent extraction.
A9 Modified browser	Custom browser enforcement	Owner removed checks and independently cloned the recovered model.
A10 Device administrator	Administrator memory access	Not separately tested. Software-owner extraction already succeeded; hardware resistance is unknown.
A11 Recording, repeating or forwarding traffic	Request freshness and permitted reuse	Duplicates and expired requests rejected; fresh authorized requests and forwarded use succeeded.
A12 Learning from answers	Behavioral reconstruction	Fixed and adaptive learners gave poor exact recovery on the tested model; this is method-specific.
A13 Capture, forwarding and saved answers	Independent offline execution	Copied interface worked online and replayed saved answers offline; unseen offline inputs failed.
A14 Delivery authorization	Recipient and program approval	Unapproved keys and altered program packages rejected; revoked permission stopped normal execution after expiry, while recovered code stayed usable.
A15 Protected execution	Hardware trust	Unsigned protected program did not load; the software-only evidence model left hardware confidentiality unverified.
A16 Visible copying	Pixels and interface content	Displayed content and pixels copied; continued display left this information available.

3 What delivered code exposes

The simplest Phase I program was 65 bytes. Fetching the delivered file recovered its complete source, and its function matched all 100 reference inputs. I then replaced the direct expression with ten integers representing instructions and a small interpreter, a program that reads and executes those instructions. The delivered file grew to 269 bytes. Searching for the original expression failed, but reading the instruction list and interpreter recovered the constants and operations. The reconstruction again matched all 100 inputs.

That result concerns this small transformation. It does not estimate the effort needed to understand a commercial obfuscator or an unfamiliar large application. The observer received both the instructions and the code explaining how to execute them. Changing that representation removed a convenient text match, while leaving the information needed for reconstruction available.

Several simpler barriers also left the resource readable. Replacing the page's logging function and cancelling a synthetic context-menu event did not stop an ordinary fetch of the 65-byte program. Removing a recipient comment erased its marker while preserving all 100 outputs. The response directive Cache-Control: no-store did not prevent reading and saving the current response. Persistent cache eviction, or removal from stored browser data, was not tested.

Robots rules stopped the cooperating capture tool, which retained zero resources, while an ordinary browser obtained the excluded page's HTML. A simple automation rule rejected one collector header but returned the script when the request used a browser-shaped agent string, a label describing the requesting software. These tests evaluated the specific rules implemented here, not a commercial detection system [1,14].

Browser access rules require a more precise distinction. An origin combines a web address's scheme, host, and port. The two loopback ports were therefore different origins despite being on the same computer. A browser fetch from the other origin raised TypeError, but an independent HTTP client outside the browser obtained the resource with status 200, meaning success in this test. A classic script loaded from the other origin also executed. The outcome depended on which kind of request the browser was handling [2].

On the integrity page, a script matching the expected SHA384 digest ran. A digest is a fingerprint calculated from the script's bytes. The script with an incorrect fingerprint did not run, and an inline assignment forbidden by Content Security Policy did not establish its flag. The rejected script's 74 bytes were still retrievable. These controls restricted execution; the experiment did not show that they made the resource secret [3,4].

An iframe is a document embedded inside another page. The parent page could not read a second-origin frame's document: the browser raised SecurityError. It could send an allowed postMessage request, the browser's document-messaging mechanism, and receive 48. That permitted operation worked without granting direct access to the frame document. The frame's program was still delivered to the browser, so the result did not establish secrecy from its owner [5].

Encrypted delivery failed at a different point. The Phase I loader used ECDH to agree shared key material, HKDF to derive the encryption key, and AES-GCM to decrypt the program and check it for changes. Keys were nonextractable: the cryptographic interface refused to return their raw bytes. In the revised path, the shared secret stayed inside the component handling

cryptography instead of passing through a JavaScript buffer. Exporting the AES key raised InvalidAccessError; altered encrypted data raised OperationError.

I then installed a hook before decryption. A hook here is observation code that calls the original operation, copies its returned data, and passes the result back. It recovered the complete 104-byte plaintext program, meaning the readable program after decryption. That program returned 48 for input 5. The encrypted program and its authentication tag occupied 120 bytes; this excluded the response's public key shared for the exchange, salt, and initialization vector. The tag checks for changes, while the latter values support the key setup and encryption. Recovery required no key export or break of the cryptographic algorithms. The local exchange did not establish an authenticated Internet key-exchange protocol [15].

The loader used JavaScript's Function operation to turn the decrypted text into executable code. That exposed the transition to page observers and would matter for a policy forbidding dynamic compilation. I did not combine this loader with the strict integrity page or claim that the combination worked. The extension discussed below instead interpreted bounded JSON instructions and prohibited eval-like text compilation. Neither renaming the call nor locking its shared JavaScript object after startup was implemented as protection against an owner controlling the browser.

Compiled programs also remained recoverable in the tested cases. The browser obtained all 49 bytes of the Phase I WebAssembly module; WABT converted it back to a readable instruction listing containing its constants and operations. Ghidra recovered the formula and interpreter structure from a 2,560-byte Windows DLL, a compiled library file. Both exported functions, the library's callable entry points, had first passed the 100-input check. In Phase II, the native decision model contained 248 nodes stored as five-integer records. Assuming that layout and the tree dimensions, I recovered one candidate table and built an independent evaluator matching 1,000 of 1,000 native outputs. This was not a blind analysis of an unknown binary.

Visible presentation remained copyable. The Phase I canvas, which draws content as pixels, produced a 10,148-byte PNG image while its designated text node was empty. Enabling the accessibility control exposed ordinary text reading "Quote result 48". Phase II also captured screenshots and interface content. These results do not establish perfect interactive copies. They show that hiding text from the page's document structure did not hide the displayed information. The canvas test did not evaluate screen-reader tasks, optical character recognition, indexing, or usability.

4 Server execution and authorized reuse

Moving the arithmetic calculation to a service kept its source out of the frontend. The original page returned 132 for input 17. Its captured counterpart retained four resources but displayed "Service unavailable" after a 403 denial and a cross-origin error. Its screenshot still passed the visual check: the changed-pixel ratio was about 0.00367, below the tool's 0.01 tolerance. Looking similar did not establish working calculations. An earlier capture had stopped at the tool's rule against additional private network destinations; a separate run explicitly authorized the owned second origin.

The service initially checked the request's Origin header, a value naming the requesting origin. A native proxy, a separate program forwarding requests, supplied the expected value and received status 200 and result 132. A browser-based copy using that proxy returned the same answer. The service's code remained private, but this header did not establish that the

intended application sent the request. Phase II's stronger authenticated service behaved differently: forging Origin alone did not authorize execution.

Phase I separately tested digital signatures on fresh requests. The browser held a nonextractable P-256 signing key. The server issued a challenge, a random value valid for two seconds, and the signed message tied it to the session, request method, endpoint path, and input. A session identifies the current authorized interaction; an endpoint is the service address handling an operation. Including these fields prevented a valid signature from simply being applied to a different tested request.

A valid request returned 200. Reusing its consumed challenge returned 409, changing the signed input returned 403, and waiting beyond expiry returned 410. Omitting the session cookie returned 401. These statuses recorded success, duplicate rejection, denial, expiry, and missing authentication respectively in this test. A newly signed authorized request still succeeded. Twenty simultaneous copies produced one success and nineteen duplicate rejections in a single Node server process. This supports the tested request checks, without proving that an unchanged page produced the signature [10,16].

The first server stored only the latest challenge. Issuing another replaced the first, causing a legitimate overlapping operation to fail. I changed it to retain at most 32 live or recent challenges independently. Both legitimate operations then succeeded, while the twenty-copy test still allowed one success. Phase II added account authentication, permission checks, and protection against cross-site request forgery, or CSRF: a page causing the browser to send an unwanted request using an existing session. It also tied signing keys to sessions and signed a fingerprint of the complete submitted data, including its request identifier.

Phase II challenges expired after approximately 1.5 seconds. Replacing the session key invalidated older challenges. Requests missing authentication returned 401. Missing CSRF protection, changed inputs, foreign-account operations, and expired challenges were rejected. Twenty independently authorized concurrent operations succeeded; submitting one signed request twenty times allowed one execution and rejected nineteen. These results concern the implemented single-process service, rather than coordinated enforcement across multiple servers.

The inventory tests also checked operations that changed stored state. Reservations enforced ownership and versions; cancellation and expiry released stock. Repeating a commit with the same request identifier returned its recorded result. Signing in again did not reset the account's usage allowance. The research accounts had budgets of 5,000 and 12 execution attempts. One application-denied reservation consumed an attempt, leaving eleven successful calculations in that probe. A separate fresh-account proxy probe completed twelve calculations and rejected the thirteenth. The prototype held its records in one process's memory; persistent storage and coordination remain deployment requirements.

Stronger checks still allowed a legitimate account to operate a copied interface. Captured assets combined with a proxy matched all 100 online calculation references, performed routing, accepted twenty fresh concurrent operations, and completed a reservation and commit. Requests without a session failed, and account allowances remained effective. The proxy received permitted answers without recovering the backend model. Its run recorded 284 requests to the original service, 133 successful pure calculations, and 102 cached input and output entries. These counts describe this workload, not a general cost of copying a service.

After the original service stopped, the cache returned all 100 previously observed answers but failed on an unseen input. An uncached ordinary request also failed. This was a saved set of

answers, not an independent implementation. Separately, copied broker interfaces rendered offline but their calculations timed out because the replay browser lacked the required execution component. They did not contact the original service. The public baseline reproduced all 100 calculation references offline and performed routing. The server-only copy displayed its interface without independently reproducing the authenticated backend workflow.

A signature therefore needs a careful interpretation. It showed possession of the registered key and acceptance under the implemented checks. It did not show which interface the user operated or whether the browser was unmodified. The authorized browser could sign a new request without exporting its key. Phase II improved login and account limits over Phase I's freely issued research sessions, but the legitimate relay still met its permissions. Preventing that use would require a further demonstrated policy.

5 What browser isolation protects

The Phase I extension moved decryption and instruction execution outside the page. This separate execution helper is called a broker. A content script, extension code running alongside the page, forwarded a narrow calculation request to the extension's background component and returned its numerical answer [17-19]. The hook observing the page's own decryption calls captured no broker plaintext. Positive controls confirmed that the hook worked. Export and invalid-input requests were denied, and directly fetching the hidden background script failed. This protected against the tested page observer, without establishing secrecy from the browser owner.

The service also had to control who received an encrypted program. An open delivery endpoint accepted a key generated by the page, encrypted the program for that key, and thereby let the page decrypt the instructions itself. Provisioning, the step that approves and registers a recipient key, closed this path for an unapproved page key. That request returned 403; enrollment without the research credential returned 401. The initial comparison deliberately kept open delivery available, so I restarted the service with only approved delivery enabled for the final restricted results.

The approved broker continued to work while open delivery was rejected. Another origin also tried supplying a false claimedOrigin field. The broker ignored that claim and checked the sender URL provided by the browser's extension system, so the request failed. Normal calculations matched all 100 reference inputs. These checks established the tested delivery and caller restrictions; they did not prove that all browser inspection paths were closed.

Phase II examined an extension worker through the browser's real debugging interface. A worker is the extension component executing the protected operation outside the page. Debugging recovered the initial model and worker source, reproduced all 100 initial held-out outputs offline, and changed execution so the page received score -777. Held-out inputs are cases reserved for checking results rather than fitting a model. Private-key export still failed. Recovering the decrypted program was enough.

A later prototype required a separate administrative authority to approve recipient public keys. Copying the enrollment credential no longer authorized an unapproved key. The worker checked issuer signatures, rejected modified program packages and older versions, and required approval of replacement keys. These controls improved who could receive a program. They did not remove the readable program from an approved worker while it executed.

The prototype also tested restart recovery and revocation, meaning withdrawal of permission. An initial extension reload timed out. A complete clean browser restart with the same isolated profile then retained the approved recipient identity, restored calculation, and continued to deny key export [20]. After revocation, the unmodified broker stopped executing when its approximately 1.2-second lease, a short-lived permission, expired. An extracted model could still run offline and matched 1,000 of 1,000 independent references. Software version checks and stored state had no demonstrated protection against an owner changing them.

The provisioning stages also have different limits. Phase I used software credentials in the temporary extension and registered a key for one service process. Phase II placed approval outside the extension and rejected an unapproved replacement even with a copied enrollment credential. That closed the tested shortcut. Approval still did not prove that the recipient ran an honest browser or kept a program secret after decrypting it.

Phase I proposed a broker in the browser itself. Phase II implemented one in a separate Waterfox installation using Mozilla's actor mechanism for communication between processes [21]. The parent process is the main browser process, outside the page's ordinary execution process. Requests crossed that real boundary, and the reported execution process matched the actual parent. The browser executable's fingerprint stayed unchanged; configuration and privileged JavaScript modules changed. This was a working browser component, not a newly compiled C++ engine.

The ordinary page could not fetch the privileged module or read its program state. Calculations matched all 100 references; buttons and keyboard input worked. Direct calls from an unauthorized origin or frame were rejected. There was an indirect-use limit: a same-origin frame could ask its authorized parent page to call the broker. Cross-origin parent access failed. The extension showed the same distinction between restricting the immediate caller and preventing every relay through an authorized page.

Browser-owner access defeated the software boundary. Privileged debugging recovered the parent model and substituted score -888. In another browser copy, I removed the custom origin, frame, and export checks. Its recovered program matched 1,000 of 1,000 references offline. The debugging driver used explicit system-access mode [22]. I did not perform a separate administrator memory-extraction attack: browser-owner access had already recovered the software-held program.

Keeping a program separate also did not guarantee an honest display. Another extension had ordinary page access and no debugger permission. It observed the broker messages and substituted score -666, which the page accepted while the real broker produced 522. Disabling substitution restored the ordinary result. Its decryption hook and direct request to export another extension's model did not recover that model. The failure concerned the message and display path. No trusted display path was implemented.

6 Disclosure through permitted outputs

Keeping source code private still leaves the answers available to study. In Phase I, I knew the function belonged to an affine family modulo 1009: a multiplier and an offset followed by a remainder operation. Answers at inputs 0 and 1 revealed offset 13 and multiplier 7. The reconstructed function matched the remaining 98 controlled reference inputs. Those were local comparisons, not 98 extra remote queries. The extension's allowed answers revealed the same function. An eight-request session limit stopped later requests, but a new research session restored access.

This depended on a small known family. It does not establish easy reconstruction of arbitrary applications or meaningful account-based abuse control. Another Phase I service returned shortened HMAC outputs, values calculated using a server-held secret key. Four recorded samples established neither reconstruction nor general resistance to analysis. Both this unresolved comparison and the later learning failures need to be read within their tested methods and prior knowledge.

The Paillier experiment tested limited calculation on encrypted numbers [11]. With a 2048-bit modulus, the browser combined encrypted coefficients to form an encrypted affine result, while the server retained the private decryption values. The modulus is a number defining the cryptographic arithmetic; its size identifies the tested configuration. Input 5 normally returned 48. But an unrestricted result endpoint also decrypted the original encrypted coefficients when submitted directly, returning 7 and 13. This exposed the coefficients through the interface without breaking the encryption.

I revised the endpoint to check that the submitted encrypted result was exactly the allowed expression for the stated input. A mismatched coefficient request then returned 403. Legitimate requests at 0 and 1 still returned 13 and 20, which revealed the same coefficients through ordinary answers. The repair stopped the direct misuse, while leaving this simple function learnable. The check also repeated the public encrypted calculation on the server, so it established no saving from outsourcing that work.

Returning categories exposed less detail. I compared 1,024 functions with integer coefficients from 0 through 31. Categories used floor(value/337), capped at two: divide the result by 337, round down, and allow no category above two. At inputs 0, 1, 7, 15, 31, 47, 63, and 95, exact answers left one possible coefficient pair. Categories left 114 possible pairs. This preserved uncertainty about the internal parameters for those observations.

I then chose new inputs according to which answers would best separate the remaining candidates. In a fresh session, queries at 37, 41, 93, 94, and 46 reduced the candidate counts to 59, 32, 14, 7, and 6. No input from 0 through 99 distinguished the survivors' categories. One remaining model reproduced all 100 categories. Several internal formulas remained possible, but their visible behavior was the same in this domain. This adaptive follow-up, where each answer influences the next query, ran only in Chromium.

The more complex Phase II decision model gave a different result. A learner based on boosted regression trees, which combines decision trees to predict numerical scores, trained on 2,048 actual application queries. It matched 11 exact scores on 1,000 held-out inputs. Mean absolute error, the average size of a prediction's error regardless of direction, was approximately 35.4 score units. Category agreement reached 91.0%, but always predicting the most common category already achieved 85.8%. The category percentage therefore needs that simple comparison beside it.

I retained the query records and learned model. Held-out answers did not enter training or adaptive query selection. An independent audit of 100 online reference cases matched the application before the learning comparison. The experiment supports the reported approximation and poor exact recovery, rather than either complete reconstruction or a general claim that the model resists learning [12].

The follow-up compared adaptive queries with an equal-budget random control. Both learners shared 2,048 initial observations and received 512 more. The adaptive method used disagreement among partial versions of its current model to select four rounds of 128 queries. The control chose additional inputs randomly. At 2,560 observations each, the adaptive

learner matched three exact held-out scores and the random learner nine. Their mean absolute errors were approximately 35.25 and 34.66. This heuristic did not improve the result. It supplies no lower bound on the queries a stronger method would need.

Reduced answers also changed what the application provided. The held-out sample contained 351 distinct scores and three categories: 858 medium, 136 low, and 6 high. Empirical marginal entropy, calculated from how often each answer appeared in that sample, was approximately 8.15 bits for scores and 0.63 bits for categories. This describes the observed output distribution. It does not measure information about the source code or prove an extraction limit. Categories may meet a product's needs, but they do not preserve an exact-score interface. This is a change in disclosed functionality, not a benefit attributable solely to moving execution.

7 Performance and deployment implications

Each timing comparison needs its own task and browser. Phase I measured thirty batches of one million local calls after ten warmup batches. Warmups let the program run before retaining measurements. The small interpreter's median was about 3.98 times the plain function in Chromium and 4.50 times in Waterfox. WebAssembly was slower than plain JavaScript in Chromium and faster in Waterfox. Browser modes, clocks, and scheduling were not standardized for a product ranking. Early 10,000-call measurements rounded to zero; longer batches corrected that resolution problem without removing other noise.

Table 2 keeps these tasks separate. The median is the middle timing; p95 marks the reported point covering roughly 95% of the timings. Phase I used nearest-rank p95: sort the samples and select the sample at the required rank. Phase II values remain as reported. All request timings used local services, excluding Internet latency. One million local calls, a signed request sequence, and a broker-message batch do different amounts of work and should not be ranked as equivalent operations.

Table 2. Selected elapsed times in milliseconds. Compare the ordinary and protected paths within each row. Sample counts describe retained measurements, excluding warmups.

Phase and workload	Ordinary median / p95	Protected median / p95	Samples
I Chromium million local calls	8.750 / 9.900	34.800 / 38.800 interpreter	30
I Waterfox million local calls	6.000 / 8.000	27.000 / 28.000 interpreter	30
II server request sequence	~0.9 / ~1.1 anonymous	~2.2 / ~2.6 signed	60
II sixteen authorized operations	27.05 / 32.30 issued separately	15.15 / 18.10 issued together	30 pairs
II Chrome extension 100 calls	~0.268 / ~0.404 page	~32.52 / ~37.57 extension	30

Issuing challenges together improved the Phase II server path while keeping a separate signature and one-use value, or nonce, for every operation. Five warmup pairs were discarded; order alternated across thirty retained pairs. Requests fell from thirty-two to seventeen for sixteen operations. Sixteen prepared operations and sixty-four independently prepared concurrent operations succeeded. Another request reached the outstanding-challenge cap. Changed requests, replay, expiry, and key replacement still produced their expected rejections. Summed Node CPU time fell from 484 to 313 ms, excluding browser signing and total system CPU use. The final standard-web check measured approximately 2.4 ms median for its protected sequence.

The Phase I extension's fifteen measured batches of one hundred cached calls had median 34.000 ms and p95 39.000 ms after three warmups. The program was already loaded; first-load cryptography was excluded. Fifteen approved-expression Paillier round trips after three warmups had median 26.800 ms and p95 28.800 ms. Those included browser and JSON work in research code using JavaScript BigInt, its large-integer representation. They were not isolated measurements of the cryptographic calculation.

The Phase II extension comparison deliberately exposed the control model to measure it in the page. Waterfox's 500-call comparison recorded medians of approximately 3.95 ms in the page and 56.21 ms through the main-process broker, with p95 values of approximately 8.56 and 59.59 ms. Each batch included one WebDriver automation round trip. External timing avoided treating individual warm-call readings rounded to zero as zero latency.

Phase I also measured fifty AES-GCM decryptions per batch with 64 KiB and 256 KiB encrypted messages. They were encrypted with distinct fresh initialization vectors. Decrypting a saved message repeatedly did not reuse a nonce for a new encryption. Chromium medians were 1.500 and 4.250 ms; Waterfox medians were 3.000 and 7.000 ms. These timings included interface calls, returned buffers, and scheduling. CPU and resident-memory snapshots, which record a process's memory at particular moments, did not establish complete browser cost, peak memory, or production capacity. Limited first-call and startup observations also did not establish reliable startup distributions.

For deployment, I would identify the implementation and operations needing protection, then retain them on an authorized server where response time, availability, cost, and input processing permit. The tests support binding sessions to inputs, fresh independent challenges, account allowances that survive login, and disclosure limited to product needs. They leave persistent storage, sustained load, and coordinated duplicate rejection across servers untested. All eight Phase II variants retained tested calculation and routing behavior, avoided horizontal overflow at 320, 390, and 1,440 pixels, and passed the reported label, live-result, and keyboard checks. This is useful accessibility evidence, without being a complete certification.

Server execution also changes where inputs are processed. The prototypes used controlled research data; a deployed service could receive personal or commercially sensitive inputs. Privacy and data-use requirements therefore belong in the decision. For client execution, the proposed broker would deny network access, storage, and other side effects unless narrowly authorized. It would explain which component is trusted, what it can access, and how to remove it. These are proposed requirements, not facilities validated by the prototypes.

8 Limits and unresolved protected execution

The attempt at stronger client protection remained unfinished. A VBS enclave is a Windows facility intended to provide isolated execution. Phase II compiled a non-debug program image for it and successfully allocated an enclave, meaning that the system created the execution area. Windows then rejected loading the unsigned program with error 577. The protected program never executed, and no verified hardware attestation was obtained. The host reported VBS support but lacked the probed SGX and SGX2 facilities, other forms of protected execution. Boot, certificate, and signing policies stayed unchanged. Appropriate enclave signing is a development requirement [23]; allocation alone did not establish confidentiality.

Attestation is evidence about the execution environment, including which software is running. A separate software model tested how such evidence might be checked. It used

signed claims, encryption tied to a recipient, fresh challenges, account and operation checks, program versions, revocation versions, and duplicate rejection. Tested modifications and mismatches were rejected. A twenty-way duplicate test accepted one submission and rejected nineteen inside one Python process with a lock coordinating the decision. This did not test multiple independent verification servers.

Giving the owner the software issuer's signing key allowed false measurement claims to pass. A measurement here identifies the software the claim says is running. The result exposed the missing independent trust anchor, meaning a source of trust outside the owner's software control. It did not show a break of hardware attestation. RATS describes the established architecture for independently checking such evidence [24].

A future protected broker would need independently checked execution evidence, approved recipients, authenticated delivery, and protection against reverting to an older program. It would also need revocation and explicit assumptions about protected storage and side channels, such as information exposed through timing or message lengths. The Phase I proposal adds restricted operation handles, references that let a page request a named operation without receiving its code, together with accepted input formats, allowed outputs, resource limits, and lifetime rules. Combining this broker with signed short-lived permissions was not validated as one system. Even successful hardware protection would leave answers, errors, relaying, and visible presentation to evaluate.

The study used specific Windows desktop builds, repeatable research programs, local services, finite input samples, and selected adversaries. I wrote both the mechanisms and countertests. Native recovery assumed a known table layout; learning tested particular algorithms and budgets. There was no blinded study of human analysis effort, exhaustive side-channel analysis, production deployment, or general reliability certification. Attaching a debugger can also change worker lifetime and timing. A modified browser loaded the public baseline with certificate verification enabled; that visit does not establish wider compatibility.

Possessing a signing key does not by itself identify an honest execution environment. Web Authentication can establish possession of a public-key credential. It does not generally prove that arbitrary page code ran unchanged [16]. In the software model, trusting the issuer's key meant trusting software the owner could control. Before releasing a private program on the strength of attestation, a stronger design would need a verified connection between the measured software, its execution key, and its protected execution. The key-export rejections elsewhere did not establish that connection.

I also separated faulty tests from faulty protections. Incorrect key permissions initially prevented deriveKey, the operation creating a derived key. Another test signed with an old session key, and Ghidra initially failed at a configuration path. These were corrected rather than counted as protection. Actual mechanism failures included replacing a live challenge, delivery to any supplied key, unrestricted decryption, and the spoofable Origin check. Phase II retained the failed reload, capture-discovery and interaction limits, quota-accounting correction, and enclave rejection. Broker captures kept the tool's incomplete verdict because calculations were untested, even where independent replay displayed the interface. A tool limitation or unsuccessful attack cannot establish secrecy by itself.

9 Conclusion

The tested server design gave the strongest demonstrated protection for private website implementations in an ordinary browser. It kept the model and enforced the configured permissions. Separate browser components hid programs from selected page observers, while

browser-owner access recovered the software-held models. Encryption, nonextractable keys, duplicate rejection, and recipient approval each protected a particular boundary. None established that permitted answers or the visible interface could not be copied.

I therefore propose keeping sensitive computation on the server where practical, authorizing each exposed operation, and testing what its actual answers reveal. A copied interface may use a legitimate account; a cache may retain old answers; learning may reproduce behavior to different degrees. The phases show easy recovery of a known arithmetic family and poor exact recovery by selected learners of a more complex model. Hardware-backed confidentiality remains unverified. These limits define the supported proposal and the questions a stronger design still needs to answer.

References

[1] M. Koster, G. Illyes, H. Zeller, and L. Sassman. Robots Exclusion Protocol. RFC 9309, 2022. https://www.rfc-editor.org/rfc/rfc9309.html

[2] WHATWG. Fetch Standard. Living Standard. https://fetch.spec.whatwg.org/

[3] W3C. Content Security Policy Level 3. Technical specification. https://www.w3.org/TR/CSP3/

[4] W3C. Subresource Integrity. Recommendation, 2016. https://www.w3.org/TR/2016/REC-SRI-20160623/

[5] Chromium Project. Site Isolation Design Document. https://www.chromium.org/developers/design-documents/site-isolation/

[6] WebAssembly Community Group. Security. Official WebAssembly documentation. https://webassembly.org/docs/security/

[7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. Author manuscript, 2010; preliminary version at CRYPTO 2001. https://www.wisdom.weizmann.ac.il/~oded/PS/obf4.pdf

[8] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza. TrustJS: Trusted Client-side Execution of JavaScript. EuroSec, 2017. DOI 10.1145/3065913.3065917. https://www.mlsec.org/docs/2017b-eurosec.pdf

[9] K. A. Kucuk, D. Grawrock, and A. Martin. Managing confidentiality leaks through private algorithms on Software Guard eXtensions enclaves. EURASIP Journal on Information Security, article 14, 2019. DOI 10.1186/s13635-019-0091-5. https://link.springer.com/article/10.1186/s13635-019-0091-5

[10] IETF. OAuth 2.0 Demonstrating Proof of Possession DPoP. RFC 9449, 2023. https://www.rfc-editor.org/rfc/rfc9449.html

[11] P. Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. EUROCRYPT 1999, LNCS 1592, pages 223 to 238. https://www.cs.tau.ac.il/~fiat/crypto7/papers/Pai99pai.pdf

[12] F. Tramer, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing Machine Learning Models via Prediction APIs. USENIX Security, 2016. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer

[13] DoppelGNG.me. Source checkout version 0.1.0, revision ec8f82bf380367b7c6a5d184c3505a25a331f329. README, architecture documentation, capture implementation, and validation implementation inspected locally. https://github.com/emillvl/DoppelGNG.me/tree/ec8f82bf380367b7c6a5d184c3505a25a331f329

[14] M. Musch and M. Johns. U Can't Debug This: Detecting JavaScript Anti-Debugging Techniques in the Wild. USENIX Security, 2021. https://www.usenix.org/system/files/sec21-musch.pdf

[15] W3C. Web Cryptography API. Recommendation, 2017. https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/

[16] W3C. Web Authentication An API for accessing Public Key Credentials Level 3. Technical specification. https://www.w3.org/TR/webauthn-3/

[17] Mozilla. Content scripts. MDN WebExtensions documentation. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts

[18] Mozilla. Background scripts. MDN WebExtensions documentation. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Background_scripts

[19] Mozilla. content_security_policy manifest key. MDN WebExtensions documentation. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/content_security_policy

[20] Google. The extension service worker lifecycle. Chrome for Developers. https://developer.chrome.com/docs/extensions/develop/concepts/service-workers/lifecycle

[21] Mozilla. JSActors. Firefox Source Docs. https://firefox-source-docs.mozilla.org/dom/ipc/jsactors.html

[22] Mozilla. geckodriver Flags. Firefox Source Docs. https://firefox-source-docs.mozilla.org/testing/geckodriver/Flags.html

[23] Microsoft. VBS Enclaves Development Guide. Microsoft Learn. https://learn.microsoft.com/en-us/windows/win32/trusted-execution/vbs-enclaves-dev-guide

[24] IETF. Remote ATtestation procedureS RATS Architecture. RFC 9334, 2023. https://www.rfc-editor.org/rfc/rfc9334.html