White-Box Cryptography Opportunities for Secure Online Assessments

Matheus Gomes Martins Coelho[1], **Felix Carvalho Rodrigues**[1,2]

[1]Universidade Virtual do Estado de São Paulo (UNIVESP)

[2]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)

2201385@aluno.univesp.br, felix.rodrigues@univesp.br

Abstract. *Secure online assessments depend on software running on student-controlled devices, which leaves exam clients exposed to reverse engineering, tampering, and code lifting. This creates a natural, but still underexplored, use case for white-box cryptography. Using Safe Exam Browser as a concrete case, we discuss how white-box techniques can strengthen assessment workflows without assuming trusted hardware on every student device. We identify the assets that benefit most from software protection and define a pragmatic attacker model for digital examinations. We outline an initial pragmatic design in which a narrow protected client module supports device and application binding for short-lived operations such as token release and protected content access.*

1. Introduction

Secure online assessments increasingly rely on client-side lockdown software to restrict copy/paste, tab switching, screen capture, local resource access, and communication with unauthorized tools during an exam session. In practice, however, such software executes on devices owned or administered by the candidate, so the platform operator rarely enjoys a truly trusted execution environment.

Within this broader landscape, Safe Exam Browser offers a concrete and representative focal point for studying endpoint protection in secure online assessments. It is designed to lock down the candidate device during the exam session [Safe Exam Browser 2026], and it therefore concentrates exactly the kinds of client-side checks, secrets, and release mechanisms that may benefit from stronger software protection. This tension makes it an interesting application domain for white-box cryptography. In the classical white-box model, the attacker is assumed to have extensive visibility into the implementation and execution of the protected cryptographic mechanism [Chow et al. 2003, Wyseur 2007]. While this model is severe, it captures an important part of the endpoint risk faced by remote and bring-your-own-device examinations: secrets embedded in the client may be extracted or re-used unless their protection is tied to software diversity, code size, obfuscation, or frequent re-provisioning.

Our claim is not that white-box cryptography alone solves exam integrity. Rather, it can become one layer in a broader defense-in-depth architecture for secure online assessments, with the rest of this paper focusing on SEB as the running example. Prior work has shown that dedicated white-box designs can increase resistance to key extraction and code lifting by relying on large or incompressible lookup-table structures [Rodrigues et al. 2019]. At the same time, the literature also shows the fragility of many practical white-box implementations under strong analysis [Bos et al. 2016]. Therefore, the most promising role for white-box cryptography in this application is likely

pragmatic instead of absolute: raising attacker cost, shortening the useful lifetime of stolen material, and binding critical operations to a protected client implementation.

Our main contributions are as follows. First, we translate the white-box threat perspective from secure online assessments into the specific setting of SEB. Second, we identify concrete client-side assets for which software-only cryptographic protection may still be worthwhile. Third, we discuss architectural patterns that combine white-box protected components with server-side controls.

The remainder of the paper is organized as follows. Section 2 introduces Safe Exam Browser in more detail. Section 3 reviews the relevant white-box concepts and proposes a realistic attacker model for SEB in the context of secure online assessments. Section 4 discusses security opportunities for both that model and current Safe Exam Browser deployments. Section 5 presents our concluding remarks.

2. Safe Exam Browser

Safe Exam Browser (SEB) [Safe Exam Browser 2026] is a software application for delivering electronic assessments in controlled environments. It acts as a specialized browser that temporarily turns a general-purpose computer into a dedicated exam workstation by restricting access to external resources and disabling operating-system functionality that could facilitate cheating.

In practice, SEB is commonly deployed with a learning management system such as Moodle [Moodle 2026]. Configuration files determine the rules that govern the session, including permitted URLs, navigation policies, and interface restrictions. The goal is to reduce the attack surface and make the exam workflow controlled, even though no software-only client can fully guarantee integrity on a device controlled by the candidate.

For our purposes, SEB should not be seen as a complete security solution by itself. Its value lies in constraining the interface while stronger deployments still rely on authentication, proctoring, telemetry, anomaly detection, and server-side checks. This also makes SEB an interesting white-box target: it concentrates configuration data, session-state transitions, release conditions, and local integrity-relevant events in an environment that the defender does not fully control.

3. White-box Security Notions

White-box cryptography studies how to implement cryptographic functionality in software even when the adversary can inspect and manipulate the implementation. In contrast with the black-box setting, the attacker may observe intermediate values, analyze lookup tables, and attempt to reconstruct embedded keys or equivalent functionality [Chow et al. 2003, Wyseur 2007]. This is especially relevant when a client application must perform cryptographic operations offline or before contacting a server.

Several notions are particularly useful in our setting. The first is *key extraction resistance*: recovering the embedded secret or an equivalent representation should remain difficult despite binary access. The second is *incompressibility* or *space hardness*: an attacker should not be able to reproduce the protected functionality with only a small fraction of the implementation [Bogdanov and Isobe 2015, Rodrigues et al. 2023]. Two additional notions from the white-box literature are especially relevant for secure online assessments. *Application binding* aims to ensure that the protected cryptographic component

remains meaningfully tied to the surrounding application logic, such as authentication, policy checks, or message filtering, so that an attacker cannot simply lift the cryptographic core and bypass the controls that should precede its use. *Hardware binding* instead ties the usefulness of the protected component to a specific device or hardware-backed functionality, so that copying the binary to another endpoint is insufficient to obtain the same capability [Bock et al. 2020, Agrawal et al. 2023].

These binding notions are attractive because they address the code-lifting problem directly, but they do so in different ways. Application binding is conceptually appealing for SEB because the browser is valuable precisely as an orchestrator of checks: exam policy validation, session-state enforcement, and controlled release of assessment material. At the same time, the discussion in [Bock et al. 2020] shows why application binding is hard to formalize in a strong white-box model: if the adversary can observe execution, they may intercept the auxiliary inputs or approved messages that the surrounding application provides to the protected component. Hardware binding is easier to state more cleanly, because the white-box component can query a device-specific secret or hardware service before performing the protected operation. In practice, this means that a lifted copy of the client may fail off-device even if the binary itself is recovered.

For SEB in secure online assessments, a full white-box attacker is often stronger than necessary, yet most weaker assumptions would be unrealistic. We therefore consider a *pragmatic exam attacker* with substantial, but not unlimited, control over the endpoint. We consider an adversary capable of extracting the SEB binary, instrumenting execution, inspecting memory at runtime, replaying application messages, and attempting code lifting attacks in which protected functionality is reused outside the intended browser workflow or on a different device. The attacker may also attempt partial manipulation of local application state and replay of previously valid execution traces. However, the threat model does not assume compromise of server-side infrastructure or cryptographic break of the underlying primitives.

Under this model, a pure software defense can only aim for delay and cost escalation. That is still meaningful in examinations because the protected assets are often short-lived. A decryption key that is useful for only one exam window, or an attestation token that expires after a few minutes, benefits from any protection technique that forces the attacker to spend substantial analysis effort during a narrow time interval. This temporal asymmetry distinguishes secure online assessment clients such as SEB from long-lived content-protection settings such as DRM.

The model also clarifies what white-box techniques should *not* protect. User-interface restrictions, webcam monitoring and network isolation are better treated as system and policy mechanisms. The white-box model is most compelling when a small number of cryptographic capabilities act as bottlenecks for broader abuse. Examples include decrypting question bundles, unsealing answer caches after a crash, signing local event logs, or deriving per-session keys from institution-issued secrets.

4. Designing White-box Solutions for SEB

The most interesting lesson from recent work on device-bound white-box cryptography is that protection against code lifting should not be framed only as “hide the key better.” A stronger goal is to make the protected client functionality usable only in the right environment. For SEB, this suggests combining two ideas: *device binding*, so that a lifted

white-box component is less useful off-device, and *application binding*, so that the same component is less useful outside the intended browser workflow.

In the SEB setting, protecting isolated secrets is insufficient if surrounding execution state can be detached, replayed, or reconstructed outside the intended assessment workflow. Thus, the security objective shifts from static secret concealment to preservation of authenticated workflow continuity. The protected component should therefore behave as a context-dependent capability rather than a standalone cryptographic oracle.

One possible way to improve Safe Exam Browser is to move more assessment logic to the server. In this server-heavy model, the client acts primarily as a presentation and collection layer, while the backend remains authoritative for exam definition, attempt lifecycle, answer validation, grading, and consistency checks. The server stores the canonical question set, derives attempt-specific shuffled instances, binds each attempt to the authenticated session, and validates all answer persistence and submission requests against stable internal identifiers. The assessment workflow can therefore be modeled as a server side state machine controlling exam start, answer persistence, timeout handling, submission, and grading. Under this design, browser inputs are treated as untrusted claims rather than trusted facts, reducing the security value of client-side tampering.

However, stronger server centralization also increases backend complexity and infrastructure demands during the exam window, particularly because of continuous validation, replay checks, and real-time state enforcement. For this reason, a server-heavy architecture should be understood as one possible improvement path rather than a universal requirement. Moreover, even in a server-centric model, the browser may still process encrypted material, hold temporary session tokens, or cache answers during transient failures. These residual client-side functions remain security-sensitive and therefore continue to motivate selective white-box protection.

A concrete architecture for Safe Exam Browser can be described through five stages: device-bound enrollment, device-bound secret recovery, application-state validation, session-scoped capability release, and protected local persistence with authenticated state continuity.

The first stage is the enrollment layer. Instead of generating a distinct white-box binary for every student device, the institution or vendor could compile one global protected module implementing only a narrow set of security-critical operations: unwrapping short-lived session secrets, validating protected configuration state, authenticating local events, and optionally releasing encrypted assessment material. This follows the practical motivation of global device-binding constructions, namely reducing deployment complexity while avoiding the creation of a universally portable cryptographic oracle [Agrawal et al. 2023].

During installation or first launch, Safe Exam Browser can interact with a local device-backed primitive such as a platform keystore, secure enclave, trusted platform module, secure element, or attestation-capable service. The device generates or exposes a device-specific public value, and the backend returns an enrollment certificate or token binding that value to the user or installation. Conceptually, this mirrors the initialization and enrollment phases commonly discussed in device-binding constructions: a one-time process associates an otherwise global protected component with one concrete endpoint [Agrawal et al. 2023]. In educational deployments, this enrollment may occur when a student registers a device for a course or when an institution provisions managed

laboratory machines.

The second stage is device-bound secret recovery. At exam start, the backend sends short-lived session material together with authenticated exam context, including attempt identifier, exam identifier, configuration hash, temporal validity window, and policy metadata describing which local events must be enforced or recorded. Before the protected module releases any useful capability, it verifies evidence derived from the enrolled device primitive. Depending on platform availability, this may involve nonce-based attestation, keystore-mediated cryptographic operations, or recovery of protected encodings bound to device-specific material.

The third stage is application-state validation. In addition to validating the enrolled device context, the protected component verifies that it is being invoked from the expected SEB execution state. This includes confirming that the correct configuration is loaded, the authorized exam session is active, integrity-sensitive policy checks have completed successfully, and the browser workflow remains consistent with the expected assessment state machine. This is the core application-binding idea for Safe Exam Browser: the protected module should depend on the browser's authenticated internal workflow rather than acting as a detached cryptographic service.

Under this model, the protected component becomes a stateful capability conditioned on device identity, authenticated browser state, session freshness, and server-issued context. The objective is not merely to prevent binary extraction, but to ensure that extracted functionality loses operational value once detached from these dependencies.

The fourth stage is session-scoped capability release. The protected module should not attempt to replace the server or implement the entire assessment protocol locally. Its role is intentionally narrow. One function is session-token protection: derive or unwrap a token only after device and application validation succeed, while ensuring that the token remains short-lived and scoped to a single assessment attempt. Another function is protected content release: if encrypted exam packages or configuration bundles are distributed before the exam session begins, the module releases decryption capability only under valid runtime conditions.

The fifth stage is protected local persistence and authenticated state continuity. In realistic examination environments, SEB cannot assume uninterrupted connectivity or perfectly stable execution conditions. Temporary disconnections, operating system interruptions, process restarts, or client-side failures may require local persistence of sensitive assessment state. Consequently, locally stored material such as buffered answers, session metadata, event logs, and temporary assessment artifacts should remain protected under per-session cryptographic material derived through the authenticated workflow.

Buffered answers may be encrypted and authenticated using short-lived session-scoped keys that are never directly exposed by the protected component. Likewise, local event logs, including focus changes, abnormal termination attempts, connectivity interruptions, configuration deviations, and submission events, may be authenticated incrementally during execution.

Authenticated local persistence serves two purposes: it reduces exposure of sensitive assessment material during offline operation or transient connectivity loss, and it improves post-exam forensic analysis by helping the backend distinguish benign execution failures from suspicious workflow deviations or replay attempts.

Replay resistance is particularly important in this context because a strong adversary may capture valid execution traces, locally stored state, or authenticated messages and attempt later reuse. Consequently, all persisted state and protected operations should depend on freshness guarantees such as server-provided nonces, monotonic counters, short-lived capability tokens, or session-specific challenge material. Without such mechanisms, application binding risks degenerating into reusable workflow replay.

This design reflects the distinction between code extraction and code lifting. Even if an attacker breaks the local environment, a lifted copy of the module should fail or yield unusable outputs without the enrolled device primitive. Likewise, without the correct SEB state and exam context, the protected component should not release useful material.

There are, however, important limitations. Not all student hardware exposes equally strong device-backed features, and some environments may provide only weak identifiers rather than robust attestation or keystore functionality. Application binding is also hard to formalize perfectly in the white-box setting because a strong attacker may still observe execution and replay parts of the browser state. The proposed design should therefore be evaluated as a cost-raising mechanism, not as an absolute guarantee.

Even with those caveats, this path offers a plausible way to harden SEB as it exists today. It does not require moving every decision to the server, and it does not require trusting the client completely. Instead, it uses the current browser as the locus of interaction, then makes sensitive operations depend on two scarce resources that are harder to clone than the binary alone: the device-specific primitive and the browser's own authenticated assessment state. This is a more concrete and operationally meaningful goal than simply asking whether a secret is hidden inside the client.

5. Conclusion

Secure online assessments should begin from the premise that the client is untrusted, but that does not force a binary choice between server-only designs and unprotected clients. For SEB, the key question is which client-side operations remain necessary and how to make them less transferable across devices, sessions, and execution contexts.

The main design direction developed in this paper is selective white-box protection with explicit device and application binding. A plausible deployment path is to compile one narrow protected module, enroll each device with a local hardware-backed primitive, and require both device evidence and valid browser state before releasing any useful capability. In practical terms, this means that session-token derivation, protected content release, authenticated local logs, and temporary answer-cache protection are performed only when the enrolled device and the expected exam workflow are both present.

An important next step is to profile SEB to identify client-side functions that are both security-critical and stable enough to protect first. A sensible implementation strategy is to start from the last stage outlined in the previous section, protected local persistence and authenticated state continuity, because it is easier to prototype, test, and validate without disrupting the exam workflow. From there, later prototypes can move backward toward session-scoped capability release, application-state validation, device-bound secret recovery, and finally enrollment. This staged approach preserves functionality and testability while progressively introducing stronger protection where it matters most.

References

Agrawal, S., Alpírez Bock, E., Chen, Y., and Watson, G. (2023). White-box cryptography with global device binding from message-recoverable signatures and token-based obfuscation. In Kavun, E. B. and Pehl, M., editors, *Constructive Side-Channel Analysis and Secure Design*, pages 241–261, Cham. Springer Nature Switzerland.

Bock, E. A., Amadori, A., Brzuska, C., and Michiels, W. (2020). On the security goals of white-box cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):342–383.

Bogdanov, A. and Isobe, T. (2015). SPACE: A suite of cryptographic primitives for personal space security. In *Financial Cryptography and Data Security Workshops*, pages 1–18. Springer.

Bos, J. W., Hubain, C., Michiels, W., and Teuwen, P. (2016). Differential computation analysis: Hiding your white-box designs is not enough. In *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 215–236. Springer.

Chow, S., Eisen, P., Johnson, H., and van Oorschot, P. C. (2003). White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, pages 250–270. Springer.

Moodle (2026). Moodle lms. `https://moodle.com/lms`. Accessed: 2026-05-07.

Rodrigues, F., Dahab, R., López, J., Fujii, H., and Serpa, A. (2023). A minimal white-box dedicated cipher proposal using incompressible lookup tables: Space-hard aes. In *Anais do XXIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 125–138, Porto Alegre, RS, Brasil. SBC.

Rodrigues, F. C., Fujii, H., Serpa, A. C. Z., Sider, G., Dahab, R., and Lopez, J. (2019). Fast white-box implementations of dedicated ciphers on the armv8 architecture. In Schwabe, P. and Thériault, N., editors, *Progress in Cryptology - LATINCRYPT 2019 - 6th International Conference on Cryptology and Information Security in Latin America, Santiago de Chile, Chile, October 2-4, 2019, Proceedings*, Lecture Notes in Computer Science, pages 341–363. Springer.

Safe Exam Browser (2026). About overview. `https://safeexambrowser.org/about_overview_en.html`. Accessed: 2026-05-07.

Wyseur, B. (2007). *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven.

A. Illustration of Proposed Architecture

Below, we present a visual overview of the preliminary proposed architecture discussed in Section 4.

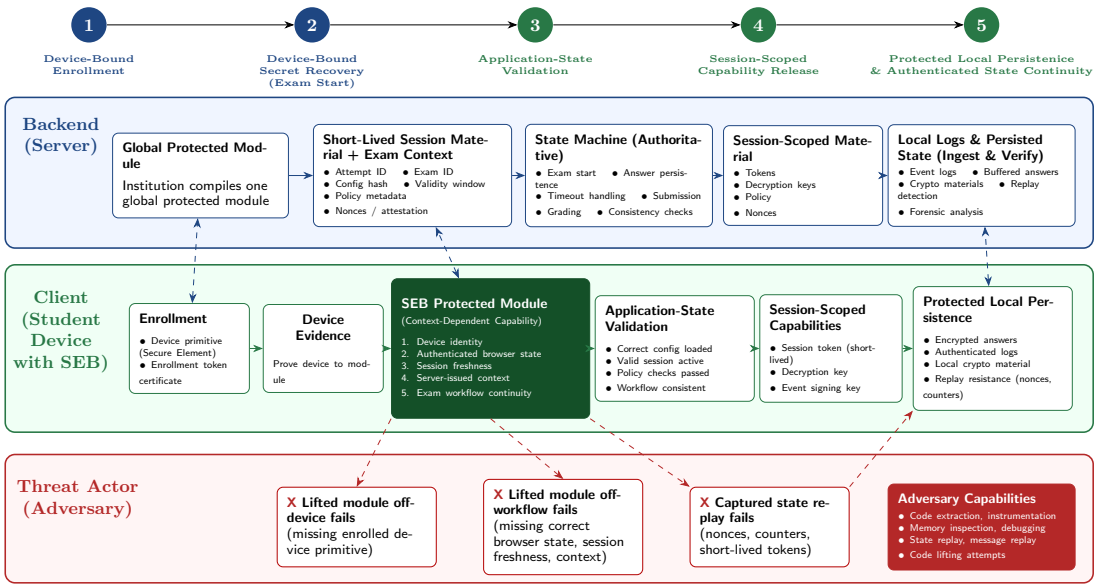

Figure 1. Preliminary design of Section 4 proposed architecture.