# Grant Proposal: Dash Shielded-Pool Payments for BTCPay Server

> **Status:** DRAFT — private to `sansbankdao/btcpayserver`. Not for public distribution. **Funder:** Joel Valenzuela ( `@TheDesertLynx` ) — direct, non-DAO funding. **Grantee:** Sansbank DAO ( `sansbankdao` ).

---

## 1. Summary

Bring Dash's new **shielded pool** (Orchard / ZK, protocol v12+) to BTCPay Server as a first-class payment method, so any merchant running BTCPay can accept **private** Dash payments natively — not by wrapping a transparent wallet behind a gateway, but by driving the real shielded spend stack.

The heavy cryptography (Halo2 proofs, Orchard circuit, Poseidon/Sinsemilla/RedDSA, grovestark STARKs) lives in the Rust engine at `dashpay/platform` and is **not** ported to C#. BTCPay is C#/.NET, so we bridge across the language boundary the same way the existing Swift (iOS) and Kotlin (Android) consumers already do: by calling the Rust crate's `extern "C"` ABI via P/Invoke.

---

## 2. Why this is worth funding

- **Dash shielded exists and is real.** `dashpay/platform` ships a complete Orchard-based shielded pool (protocol gate: `SHIELDED_POOL_INITIAL_PROTOCOL_VERSION` = v12). The wallet layer ( `packages/rs-platform-wallet` ) and a C FFI ( `packages/rs-platform-wallet-ffi` , `crate-type = ["staticlib","cdylib","rlib"]` , described as "C FFI bindings for platform-wallet") are already built and consumed by shipping iOS ( `dashwallet-ios` via `DashSDKFFI.xcframework` ) and Android consumers.
- **BTCPay is the reference self-hosted Bitcoin payment processor** (7,730 stars, MIT-licensed, 810 active discussions) and already has a Dash plugin — but that plugin is **transparent L1 only** ( `BTCPayServer/Plugins/Altcoins/AltcoinsPlugin.Dash.cs` , built on NBitcoin + NBXplorer). There is no shielded path today.
- **No C# Orchard/Halo2 port exists** (verified by GitHub code search; only unrelated card-game classes match). Porting the ZK stack to C# would be person-years of crypto work with a high chance of correctness bugs. P/Invoke into the existing audited Rust engine is the responsible path.

## 3. Technical approach (verified, not speculative)

### 3.1 Bridge the existing C ABI from C

`rs-platform-wallet-ffi` exposes `#[no_mangle] extern "C"` entry points across four files (`src/shielded_sync.rs`, `src/shielded_send.rs`, `src/shielded_types.rs`, `src/shielded_persistence.rs`). Verified signatures relevant to a payment processor:

**Sync (read the pool, discover received notes):** - `platform_wallet_manager_shielded_sync_start(handle: Handle) -> PlatformWalletFFIResult` - `platform_wallet_manager_shielded_sync_stop(handle: Handle) -> PlatformWalletFFIResult` - `platform_wallet_manager_shielded_sync_is_running(handle, *mut bool) -> PlatformWalletFFIResult` - `platform_wallet_manager_shielded_sync_sync_now(handle: Handle) -> PlatformWalletFFIResult` - `platform_wallet_manager_bind_shielded(handle, *const u8, *mut MnemonicResolverHandle, *const u32, usize) -> PlatformWalletFFIResult` - `platform_wallet_manager_configure_shielded(handle, *const c_char) -> PlatformWalletFFIResult` - `platform_wallet_manager_shielded_default_address(handle, *const u8, u32, *mut u8, *mut bool) -> PlatformWalletFFIResult`

**Prover (Halo2 proving key, ~30s warm-up, must be pre-built):** - `platform_wallet_shielded_warm_up_prover()` — builds the proving key - `platform_wallet_shielded_prover_is_ready() -> bool` - `platform_wallet_shielded_estimate_fee(handle, u8, usize, *mut u64) -> PlatformWalletFFIResult`

**Spend (send a shielded payment / deshield / withdraw):** - `platform_wallet_manager_shielded_transfer(handle, *const u8, *mut MnemonicResolverHandle, u32, *const u8, u64, *const c_char) -> PlatformWalletFFIResult` - `platform_wallet_manager_shielded_unshield(handle, *const u8, *mut MnemonicResolverHandle, u32, *const c_char, u64) -> PlatformWalletFFIResult` - `platform_wallet_manager_shielded_withdraw(handle, *const u8, *const MnemonicResolverHandle, u32, *const c_char, u64, u32) -> PlatformWalletFFIResult` - `platform_wallet_manager_shielded_identity_create_from_pool(...)` (11 params) - `platform_wallet_manager_shielded_shield_preflight(...)` - `platform_wallet_manager_shielded_shield(...)` - `platform_wallet_manager_shielded_shield_to_recipient(...)`

Each becomes a `[DllImport("libdash_platform_wallet")]` stub in C# — the same shape the Swift and Kotlin consumers already use.

### 3.2 A new BTCPay plugin, not an edit to the existing Dash plugin

The transparent-L1 Dash plugin ( `AltcoinsPlugin.Dash.cs` , NBitcoin/NBXplorer) stays as-is. We add a separate payment-method plugin `AltcoinsPlugin.DashShielded.cs` that: - registers a new `BTCPayNetwork` for shielded Dash (distinct from the transparent DASH network), - owns the native library lifecycle (load `libdash_platform_wallet.{so,dylib,dll}` via BTCPay's existing `Plugins/Dotnet/LibraryModel/NativeLibrary.cs` + `Loader/AssemblyLoadContextBuilder` infra — present in-repo, currently unused by any coin plugin), - drives sync → estimate-fee → transfer on each invoice, and reports settled state back to the BTCPay invoice lifecycle.

### 3.3 Native cross-build packaging (the genuine risk area)

The one piece I cannot fully scope from in-repo evidence is the cross-platform packaging of the `cdylib` into the .NET app on Windows/Linux/macOS. BTCPay's `NativeLibrary` loader exists but is only exercised by the generic plugin loader, not by any coin. The plan is to mirror the existing `DashSDKFFI.xcframework` (iOS) and `libdash_sdk_jni.so` (Android) build steps, but as NuGet-packed native assets for the three desktop .NET runtimes. This is the single highest-risk work item and is the natural place for schedule slippage.

---

## 4. Deliverables

1. `BTCPayServer/Plugins/Altcoins/AltcoinsPlugin.DashShielded.cs` — payment-method registration.
2. A C# P/Invoke wrapper class mirroring the `extern "C"` surface in §3.1.
3. A native-library packaging pipeline producing `libdash_platform_wallet` artifacts for Windows (x64), Linux (x64), macOS (x64 + arm64), consumed by BTCPay's existing `NativeLibrary` / `AssemblyLoadContextBuilder` loader.
4. Regtest/integration tests under `BTCPayServer.Tests` using a local Dash Evolution Platform dev network (protocol v12+).
5. A `Changelog.md` entry and a PR description per `.agents/skills/btcpayserver-*` conventions (in the event this is later upstreamed to `btcpayserver/btcpayserver` ).

---

## 5. Scope limits (what is NOT in scope)

- **No porting of Halo2 / Orchard / grovestark to C#.** The Rust engine stays the source of truth.
- **No changes to the transparent-L1 Dash plugin.** Existing DASH behavior is preserved.
- **No unattended shielded spends from protected (HD-locked) wallets.** `dash-evo-tool`'s own coordinator already forbids this; we inherit the same constraint — shielded spends require the mnemonic resolver handle, which the merchant supplies at runtime.
- **No assumption of upstream BTCPay acceptance.** This proposal funds the private `sansbankdao/btcpayserver` integration. Upstreaming is a separate, later decision.

---

## 6. Honest risk register

| # | Risk | Mitigation |
|---|---|---|
| R1 | Native cross-build packaging into .NET has no existing coin-plugin precedent in BTCPay | Time-box a spike first; fall back to single-platform (Linux x64) MVP if cross-build slips |
| R2 | Halo2 prover warm-up (~30s) on first invoice may hurt UX | Pre-warm at plugin load; surface “prover warming up” state in the BTCPay UI |
| R3 | No evidence BTCPay maintainers want a Dash-shielded plugin (recent discussions skew Bitcoin-only) | Open a `github.com/btcpayserver/btcpayserver` discussion thread before any upstream attempt; this proposal does not depend on upstream acceptance |
| R4 | Shielded sync latency vs BTCPay’s invoice-settlement expectations | Reuse `sync_now` + per-invoice note scan; benchmark before claiming real-time |

---

## 7. Open questions for the funder

1. Target deliverable horizon — is this a **3-month**, **6-month**, or **12-month** engagement?
2. Is the expected output a **private Sansbank integration only**, or is upstreaming to `btcpayserver/btcpayserver` a funded requirement? (They have different review/licensing costs.)

3. Which Dash Evolution Platform network(s) must be supported — **mainnet** (v12 once live), **testnet**, or **regtest only** for the MVP?
4. Is a native cross-build for all three OSes required, or is a **Linux-only** MVP acceptable for the first milestone (BTCPay is most commonly self-hosted on Linux)?

---

## 8. Provenance of facts in this proposal

- BTCPay repo, plugin location, license, stars, discussion count: `gh api repos/btcpayserver/btcpayserver`, `gh api graphql` over discussions (this session).
- Transparent Dash plugin: `BTCPayServer/Plugins/Altcoins/AltcoinsPlugin.Dash.cs` (present, 1622 B, in this clone).
- Native-library loader infra: `BTCPayServer/Plugins/Dotnet/LibraryModel/NativeLibrary.cs`, `BTCPayServer/Plugins/Dotnet/Loader/AssemblyLoadContextBuilder.cs` (in this clone).
- Shielded FFI signatures: `packages/rs-platform-wallet-ffi/src/shielded_sync.rs`, `shielded_send.rs`, `shielded_types.rs`, `shielded_persistence.rs` (`dashpay/platform`).
- FFI consumers: `DashSDKFFI.xcframework` (Swift/iOS, `dashpay/dashwallet-ios`), `libdash_sdk_jni.so` (Kotlin/Android, `rs-unified-sdk-jni`).
- Orchard ZK stack: `dashpay/orchard` (`halo2_proofs = "0.3"`, `halo2_gadgets`, `sinsemilla`, `reddsa`).
- Funder identity: `x.com/thedesertlynx` page `<title>` = “Joel Valenzuela (@TheDesertLynx) / X”.