Analyse nicht-zertifizierter IT-Sicherheitselemente

Untersuchung einer Java-Karte mit FIDO U2F Applet

Prüfbericht

Autor: TÜV Informationstechnik GmbH

Version: 1.0

Datum: 14.09.2016

Änderungshistorie

Version	Datum	Name	Beschreibung
0.3	2016-08-16	TÜViT	Zwischenstand für BSI zur Kommentierung
1.0	2016-09-14	TÜViT	Finaler Report

Bundesamt für Sicherheit in der Informationstechnik
Postfach 20 03 63
53133 Bonn
Tel.: +49 22899 [redacted]
E-Mail: [redacted]@bsi.bund.de
Internet: https://www.bsi.bund.de

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung und Motivation

Sicherheitselemente und Authentifizierungstoken, wie zum Beispiel USB-Token nach FIDO-Standard [FIDO15] oder SmartCards finden in den unterschiedlichsten Bereichen Verwendung. Sowohl im hoheitlichen Kontext als auch im privatwirtschaftlichen Umfeld sind sie ein lohnendes Angriffsziel, wenn sie eine hohe Verbreitung haben und zum Schutz sensibler Daten eingesetzt werden.
Das Auslesen von sensiblen Daten durch Dritte sollte durch die Verwendung von Sicherheitselementen geschützt sein. Eine konkrete Einschätzung des praktischen Sicherheitsniveaus existiert jedoch in der Regel nur für CC-zertifizierte Produkte. In den o.g. Bereichen kommen jedoch häufig auch nicht-zertifizierte „Massenprodukte“ zum Einsatz, deren tatsächliches Sicherheitsniveau unbekannt ist. Insbesondere für den Einsatz im hoheitlichen Bereich, aber auch im Bereich von kritischen Infrastrukturen ist die Verwendung derartiger Sicherheitstoken bedenklich.
In der vorliegenden Studie wird ein nicht-zertifiziertes Sicherheitselement exemplarisch untersucht. Hierbei umfasst das Wort Sicherheitselement den Chip, das OS und das Applet. Die Untersuchung dient primär der Erprobung von Seitenkanalangriffstechniken auf einen Sicherheitscontroller, der in einem nicht-zertifizierten Modus betrieben wird. Hieraus sollen Schlüsse auf die grundsätzliche Anwendbarkeit von Seitenkanalangriffen auf marktübliche Sicherheitscontroller, insbesondere bei nicht-zertifizierter Instanziierung, gezogen werden. Die Ergebnisse der Studie sollen auch dazu dienen eine Sicherheitsbewertung der untersuchten FIDO-Anwendung vornehmen zu können und eine Empfehlung für oder gegen den Einsatz in sicherheitskritischen Anwendungen treffen zu können.
Als Basis unserer Sicherheitsuntersuchung dient die offizielle FIDO U2F Spezifikation vom 14 Mai, 2015.

2 Softwaretechnische Präparation

Dieser Abschnitt gliedert sich in drei Teile. Zunächst wird beschrieben, welche Voraussetzungen gegeben sein müssen und wie das Vorgehen ist, um das zu untersuchende FIDO-Applet auf die gegebene Plattform zu laden. Daraufhin wird beschrieben, wie man mit diesem Applet kommunizieren kann. Der letzte Schritt befasst sich damit, wie eigene Applets für die Plattform erstellt und auf der Plattform ausgeführt werden können. Verwendete Skripte hierfür werden jeweils im Anhang gelistet und referenziert. Die verwendete Software wird gegebenenfalls angegeben oder (falls es sich um eigene Software der TÜV Informationstechnik GmbH handelt) deren Funktionalität beschrieben.

2.1 Laden und Installation des FIDO-Applets auf die vorgegebene Plattform

Das untersuchte FIDO Applet ist eine Open-Source-Implementierung des FIDO-Standards und kann unter [redacted] heruntergeladen werden. Das Applet wurde von der französischen Firma Ledger entwickelt, welche eigene Java-Karten vertreibt. Auf diesen Java-Karten ist der Card-Appstore *fidesmo* (https://www.fidesmo.com) vorinstalliert, über den das FIDO-Applet auf diesen vorbereiteten Karten installiert werden kann.
Die uns zur Verfügung gestellten Java-Karten stammten leider nicht von Ledger und haben daher keinen Zugriff auf den *fidesmo*-Store. Daher musste das Applet von uns selbst kompiliert und auf der Java-Karte installiert werden. Für diesen Prozess wurde die Software JCworkBench (Version 2.7.2) von der Firma Riscure genutzt. Die Kompilierung des Applets war ohne Modifikationen möglich. Da die Java-Karte Globalplatform-kompatibel ist, konnte das Applet mit den Standardbefehlen über den Cardmanager der Java-Karte geladen werden. Die verwendeten Transportschlüssel entsprachen den Standardschlüsseln nicht personalisierter Java-Karten.
Applets werden standardmäßig über den Cardmanager geladen und danach über den Cardmanager in das Java-Card OS installiert. Das erste Problem trat bei der Installation des Applets auf. Die Source-Code-Analyse ergab, dass das Applet die folgenden Installationsparameter erwartet:

- privaten ECC-256 Schlüssel,
- ein von einer CA ausgestelltes Zertifikat, welches den dazugehörigen öffentlichen Schlüssel signiert.

Ein Beispieldatensatz kann in der Dokumentation des FIDO-Standards unter „*FIDO U2F Raw Message Formats*“ gefunden werden. Nach der Anpassung des Installationsscripts ließ sich das Applet installieren und anschließend über den Cardmanager auswählen. Unter Zuhilfenahme der Beispieldaten wurden auch Beispielskripte zum Aufruf der beiden FIDO-Funktionen „*Registration*“ und „*Authentication*“ erstellt und getestet.

2.2 Messumgebung für das FIDO-Applet

Als Messumgebung wurde ein proprietäres von TÜViT entwickeltes Programm namens DPA_FI genutzt. Die Messskripte basieren auf den JCworkBench-Skripten, welche die Funktionen „*Registration*“ und „*Authentication*“ aufrufen (siehe voriges Kapitel), und liegen dem Bericht bei. Das Programm ist verantwortlich für die Kommunikation mit dem FIDO-U2F-Applet und dem Oszilloskop, sowie für die Aufnahme und Speicherung der Messdaten. Für die Analyse der Messdaten wurden proprietäre Analyseprogramme der TÜViT genutzt und, sofern notwendig, angepasst. Die Analysemethoden werden in Kapitel 5 beschrieben.

2.3 Erstellen eigener Applets für die vorgegebene Plattform

Basierend auf der Protokollanalyse und der Definition der Assets (siehe Kapitel 4) wurden 3 eigene Applets für die Sicherheitsanalyse entwickelt. Hier sei erwähnt, dass das FIDO-Applet keine eigenen kryptographischen Funktionen implementiert sondern die vom [redacted] implementierten Funktionen nutzt (siehe Kapitel 4.2).

1. Das erste Applet instanziiert das vom Betriebssystem implementierte AES-Modul und ermöglicht es uns diesen mit selbst gewählten Schlüsseln und Plaintexten aufzurufen.
2. Das zweite Applet instanziiert das vom Betriebssystem implementierte ECC-Modul und ermöglicht es uns die Funktionen zur Schlüsselgenerierung und Signaturerzeugung aufzurufen. Hier sei erwähnt, dass es keine Möglichkeit gibt, implementierte Gegenmaßnahmen auszuschalten. Weiterhin ist es nicht möglich, selbst gewählte *ephemeral keys* für die ECDSA Signaturerzeugung zu übergeben. Stattdessen werden diese in jedem Aufruf der ECDSA-Signaturerzeugung vom Betriebssystem zufällig gewählt.
3. Das dritte Applet implementiert eine Kopie der deinterleave Funktion, die einen privaten ECC Schlüssel aus dem entschlüsselten *key handle* extrahiert.

Die Applets wurden mit Hilfe von JCworkBench programmiert, kompiliert und auf der Java-Karte installiert. Die Source-Codes der Applets liegen dem Bericht bei.

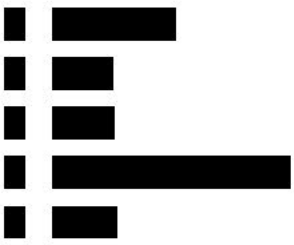
3 Charakterisierung der Hardware

3.1 Identifizierung des zugrundeliegenden Sicherheits-ICs durch Oberflächenaufnahmen und Recherche

Die uns zur Verfügung gestellten Java-Karten wurden an das zu TÜV NORD gehörende Chemielabor gegeben. Dort wurden sie mit Salpetersäure aufgeätzt um die Chipoberfläche freizulegen. Hier sei zu erwähnen, dass bei diesem Prozess der Chip leicht zerstört werden kann und 2 von 3 Java-Karten zerstört wurden. Des Weiteren können Säurerückstände dazu führen, dass Chips nach einer gewissen Zeit plötzlich nicht mehr funktionieren. Abbildung 1 zeigt eine unserer aufgeätzten Java-Karten.

Abbildung 1: Freigelegte Chipoberfläche einer Java-Karte

Auf Abbildung 1 kann man deutlich verschiedene Bereiche auf der Chipoberfläche erkennen. Im Folgenden haben wir versucht, diese Bereiche zu benennen. Die Identifikation der verschiedenen Bereiche basiert auf unserer Erfahrung und und hat keinen Anspruch auf 100%ige Richtigkeit.



[redacted]

4 Protokollanalyse und Source-Code-Analyse des Applet

4.1 Protokollanalyse

Das FIDO-Protokoll unterteilt sich grundsätzlich in zwei Phasen, Registrierung und Authentifizierung, die im Folgenden vorgestellt und analysiert werden. Dabei werden wir die allgemeine Funktionsweise erklären und dann auf die Sicherheitsaspekte eingehen, das heißt welche Sicherheitsziele wurden definiert und wie werden diese von den Funktionen umgesetzt bzw. erfüllt. Abschließend werden wir noch einige Defizite des FIDO-Protokolls diskutieren.

4.1.1 Allgemein

Jedes FIDO-U2F-Token besitzt ein ECC-256-SchlüsselPaar, welches während der Produktion/Personalisierung auf das Token geladen wird. Der öffentliche Schlüssel wird zusätzlich mit dem privaten Schlüssel des Herstellers (oder einer vertrauenswürdigen CA) signiert und das Zertifikat auf dem Token gespeichert.

4.1.2 Registrierung

Das Registrierungskommando erlaubt es einem Nutzer sich bei einer Partei (meistens eine Webseite) sich für die Two-factor-authentication (U2F) zu registrieren. Genaugenommen registriert sich die Webseite bei dem FIDO Token (hier unser Applet) und schaltet sich so für U2F mit dem Besitzer des Applets frei. Die Kommunikation zwischen Webseite und FIDO-Token läuft über eine Vermittlungsstelle, was meistens der Browser des Benutzers ist. Abbildung 2 zeigt den typischen Ablauf einer Registrierung.

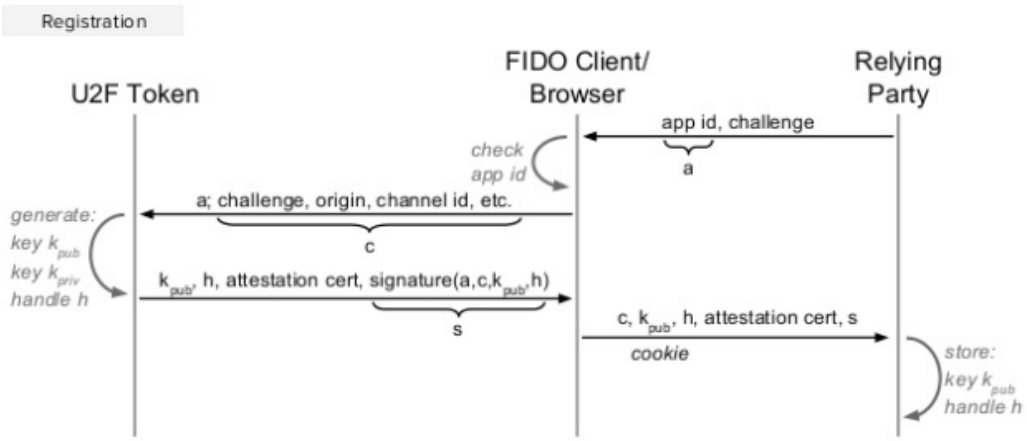


Abbildung 2: Ablauf eines Registrierungskommandos nach FIDO [FIDO15]

Im ersten Schritt generiert die Relying Party (Webseite) eine *challenge* und eine *app id*. Die *challenge* ist ein JSON Daten-Objekt, welches eine zufällige (websafe-base64 encodierte) Nonce beinhaltet. Zusätzlich ist in diesem Objekt noch die Web Origin kodiert (Domäne der zu registrierenden Seite) sowie optional eine

Channel ID public key, falls Channel ID unterstützt wird. Auf die Funktion Channel ID wird nicht weiter eingegangen, da sie irrelevant für unsere Sicherheitsuntersuchung ist. Die *app id* ist eine eindeutige Bezeichnung der relying Party, meistens die URL der zu registrierenden Webseite.

Im zweiten Schritt wird die *app id* und die *challenge* der Webseite vom Browser des Nutzers überprüft, insbesondere die Origin. Dies sollte aber einfach möglich sein, da die Kommunikation im Normalfall über eine gesicherte TLS-Verbindung stattfindet und die Webseite somit gegenüber dem Browser bereits authentifiziert ist. Diese TLS-Verbindung verhindert auch eine Manipulation der Daten während der Registrierung. Nach der Überprüfung werden die *app id* sowie die *challenge* vom Browser gehasht (SHA-256) und an das U2F Token weitergeleitet.

Das U2F-Token generiert nun ein neues ECC-256 Schlüsselpaar für die anfragende Webseite und gibt die folgenden Daten zurück:

1. Neu generierter öffentlicher Schlüssel.
2. *key handle* welches dem Token erlaubt, das generierte Schlüsselpaar zu identifizieren. Der FIDO-Standard lässt offen, wie dieser Wert berechnet wird. Es wird jedoch vorgeschlagen, den neu generierten privaten Schlüssel der Webseite vom Token verschlüsseln zu lassen und diesen Wert als *key handle* zu nutzen. Dies hat den Vorteil, dass der Token die generierten privaten Schlüssel der Webseiten nicht speichern muss, sondern diese während der Authentifizierung von den Webseiten geschickt bekommt. Diese können dann vom Token wieder entschlüsselt und für die Authentifizierung genutzt werden. Somit ist es theoretisch möglich, eine unendliche Anzahl an Webseiten auf dem Token zu registrieren.
   Das von uns analysierte Applet verschlüsselt den neu generierten privaten ECC-Schlüssel mit AES-256 und benutzt diesen Wert als *key handle*.
3. Zertifikat für den public key des Tokens, ausgestellt vom Hersteller oder einer CA.
4. Signatur über *app id, challenge*, generierter öffentlicher Schlüssel für die Webseite und *key handle*, erzeugt mit dem privaten Schlüssel des Tokens

Authentifizierung

Das Authentifizierungskommando erlaubt es einem Nutzer, sich gegenüber einer zuvor registrierten Webseite zu authentifizieren. Abbildung 3 zeigt den Protokollablauf.

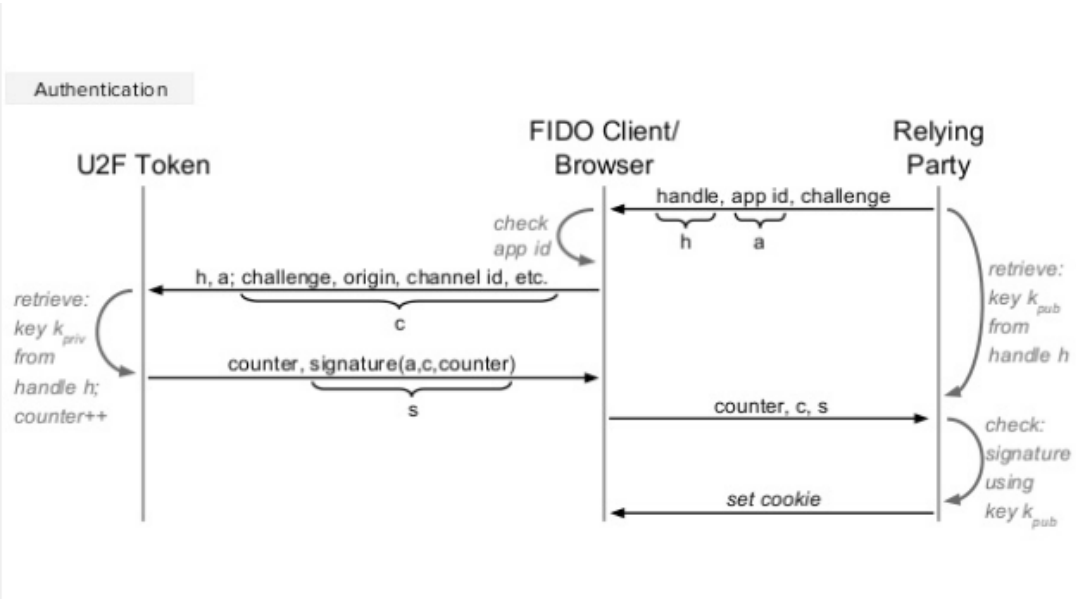


Abbildung 3: Ablauf eines Authentifizierungskommandos nach FIDO [FIDO15]

Im ersten Schritt schickt die Relying Party ihre *app id*, eine *challenge* (mit einer zufälligen Nonce), und das erhaltene *key handle* aus der Registrierungsphase. Der Browser dient wieder als Vermittlungsstelle zwischen Webseite und FIDO-Token und ist verantwortlich für die Überprüfung der *app id*. Wie bereits erwähnt muss die Verbindung zwischen Browser und Webseite mit TLS gesichert sein, damit der Browser die *app id* verifizieren kann. Nach der Verifikation werden die Daten an das Token weitergeleitet.

Der FIDO-Token kann nun aus dem *key handle* den privaten ECC-Schlüssel der Webseite entschlüsseln und mit diesem eine Signatur über die empfangene *app id, challenge* und einen *counter* erstellen. Diese Werte werden an die Webseite übermittelt, die nun mit dem in der Registrierung zugewiesenen öffentlichen ECC-Schlüssel die Signatur verifizieren kann.

4.2 Source-Code-Analyse

Unsere Analyse fokussiert sich auf die Teile des Source-Codes, welche Schlüsselmaterial verarbeiten oder kryptographische Funktionen aufrufen. Jegliches Schlüsselmaterial, d.h. das ECC-Schlüsselpaar des Tokens, der AES-Schlüssel des Tokens, sowie die generierten ECC-Schlüssel für sich registrierende Webseiten, werden in den von der Plattform zur Verfügung gestellten Schlüsselcontainern (*javacard.security*) gespeichert. Diese schützen die Schlüssel vor unbefugtem Zugriff, gegen Manipulation und gegen Seitenkanalattacken.

Das Applet implementiert selbst keine kryptographischen Funktionen. Alle ECC- und AES-Operationen werden über die von der Plattform zur Verfügung gestellten Funktionen ausgeführt. Hier ist davon auszugehen, dass die kryptographischen Funktionen der Plattform gegen Fehlerinjektion und Seitenkanalangriffe gesichert sind.

Die einzige Funktion, in der Schlüsselmaterial direkt vom Applet verarbeitet wird, ist die *deinterleave*-Funktion. Nach der Entschlüsselung des *key handles* während einer Authentifizierung wird durch diese Funktion die Verwürfelung rückgängig gemacht und der private ECC-Schlüssel der Webseite extrahiert. Der Sourcecode der *deinterleave*-Funktion ist im Folgenden dargestellt.

```
private static void deinterleave(byte[] src, short srcOffset, byte[] array1, short array1Offset, byte[] array2,
short
                                 array2Offset, short length)
  {
     for (short i=0; i<length; i++) {
        short a = (short)(src[(short)(srcOffset + 2 * i)] & 0xff);
        short b = (short)(src[(short)(srcOffset + 2 * i + 1)] & 0xff);
        array1[(short)(array1Offset + i)] = (byte)((short)(a & 0xf0) | (short)(b >> 4));
        array2[(short)(array2Offset + i)] = (byte)(((short)(a & 0x0f) << 4) | (short)(b & 0x0f));
     }
  }
```

Tabelle 1: Sourcecode der deinterleave-Funktion

In der dargestellten Funktion ist src das entschlüsselte *key handle. array1* wird der extrahierten *app id* und *array2* dem entschlüsselten privaten ECC-Schlüssel der Webseite zugewiesen. Wie zu sehen ist, wird jedes Byte des entschlüsselten *key handles* mehrmals gelesen und geschrieben (teilweise als Nibble). Diese Funktion eignet sich also theoretisch sehr gut für einen Seitenkanalangriff (siehe Kapitel 5.2).

4.3 Identifizierte Assets

Folgende Assets werden im Dokument „*FIDO Security Reference*“ definiert:

1. Cryptographic Authentication Key. Typically keys in FIDO are unique for each tuple of (relying party, user account, authenticator).

2. Cryptographic Authentication Key Reference. This is the cryptographic material stored at the relying party and used to uniquely verify the Cryptographic Authentication Key, typically the public portion of an asymmetric key pair.
3. Authenticator Attestation Key (as stored in each authenticator). This should only be usable to attest a Cryptographic Authentication Key and the type and manufacturing batch of an Authenticator. Attestation keys and certificates are shared by a large number of authenticators in a device class from a given vendor in order to prevent their becoming a linkable identifier across relying parties. Authenticator attestation certificates may be self-signed, or signed by an authority key controlled by the vendor.
4. Authenticator Attestation Authority Key. An authenticator vendor may elect to sign authenticator attestation certificates with a per-vendor certificate authority key.
5. Authenticator Attestation Authority Certificate. Contained in the initial/default trust store as part of the FIDO Server and contained in the active trust store maintained by each relying party.
6. Active Trust Store. Contains all trusted attestation master certificates for a given FIDO server.
7. All data items suitable for uniquely identifying the authenticator across relying parties. An attack on those would break the non-linkability security goal.
8. Private key of Relying Party TLS server certificate.
9. TLS root certificate trust store for the user's browser/app.

Von den hier aufgeführten Assets sind nur die schwarz hinterlegten für unsere Sicherheitsuntersuchung relevant. Zusätzlich zu den bereits erwähnten Assets hat unsere Source-Code-Analyse noch folgendes Asset identifiziert:

10. Symmetric Encryption Key. This key is used to encrypt the ECC private key of an authenticator to be used as key handle.

Im Folgenden werden wir uns auf die identifizierten relevanten Assets konzentrieren.

4.4 Angriffsszenarien

Basierend auf unseren bisherigen Untersuchungen haben wir zwei Angriffsszenarien erarbeitet, die aus unserer Sicht am realistischsten und vielversprechendsten (aus Sicht eines Angreifers) sind. Einen Angriff auf den Authenticator Attestation Key halten wir für nicht zielführend. Dieser wird nur für die Signierung eines neu generierten ECC-Schlüsselpaars während einer Registrierung verwendet. Hätte ein Angreifer diesen Schlüssel könnte er theoretisch das Schlüsselpaar austauschen, durch sein eigenes ersetzen und eine korrekte Signatur erstellen. Das Problem hier ist, dass er keinen gültigen privaten Schlüssel einschleusen kann, da dieser nochmals verschlüsselt wurde und als key handle genutzt wird. Somit würde der entschlüsselte private Schlüssel nicht zum öffentlichen Schlüssel passen und der Angriff auffallen.

Die nachfolgenden Abschnitte beschreiben die aus unserer Sicht vielversprechendsten Angriffsszenarien:

4.4.1 Angriff auf den Cryptographic Authentication Key einer Webseite

Bei diesem Angriff versucht ein Angreifer den privaten ECC-Schlüssel einer bestimmten Webseite zu erhalten. Für diesen Angriff muss der Angreifer (kurzzeitig) in den Besitz eines FIDO-Tokens kommen. Zusätzlich muss er einmal eine Authentifizierung des Users mit der Zielwebseite mitschneiden, um *app id*, *key handle* und *challenge* zu erhalten. Hat er diese Werte, kann er nun selbst die Authentifizierungsfunktion des FIDO-Tokens aufrufen. Wie in Kapitel beschrieben wird während der Authentifizierung eine Signatur über die *app id*, die *challenge* und einen *counter* berechnet, die den User gegenüber der Webseite authentifiziert. Diese Funktion wird nun per Seitenkanalanalyse angegriffen, um den privaten ECC-Schlüssel der angegriffenen Webseite zu erhalten. Hier sei erwähnt, dass der Angreifer Kenntnis über den gesamten Input und Kontrolle über einen Teil des Inputs (*challenge*) hat. Hat ein Angreifer diesen Schlüssel extrahiert und ist im Besitz der Logindaten des Users, kann er sich nun bei der Webseite anmelden und authentifizieren.

Eine zweite Möglichkeit den privaten ECC-Schlüssel anzugreifen ist ein Template Angriff auf die *unwrap* Funktion. Diese Funktion entschlüsselt das mit AES verschlüsselte *key handle* und extrahiert danach den privaten ECC Schlüssel aus dem erhaltenen Plaintext.

4.4.2 Angriff auf den Symmetric Encryption Key

Bei diesem Angriff versucht der Angreifer den privaten symmetrischen Schlüssel des FIDO Tokens zu erhalten, der für die Entschlüsselung der *key handles* benutzt wird. Der Angreifer hat zwei Möglichkeiten den Angriff durchzuführen. Er kann die Berechnung eines *key handles* (Verschlüsselung) während des Registrierungsprozesses angreifen. In diesem Fall hat ein Angreifer Kenntnis über die Ausgabe der Funktion. Die zweite Möglichkeit ist die Entschlüsselung des *key handles* (= privater ECC-Schlüssel) anzugreifen. In diesem Fall kennt und kontrolliert der Angreifer die Eingabe der Funktion. In beiden Fällen wird die symmetrische kryptographische Funktion des FIDO-Tokens angegriffen. In den uns vorliegenden Java-Karten wird wie bereits erwähnt ein AES-256 verwendet.

Schafft es ein Angreifer in den Besitz des AES-Schlüssels zu kommen, kann er alle vom Token erstellten *key handles* entschlüsseln und erhält somit alle privaten ECC-Schlüssel der beim Token registrierten Webseiten und kann sich nun bei allen Webseiten authentifizieren (vorausgesetzt der Angreifer ist im Besitz der Logindaten des Nutzers). Überdies kann ein Angreifer mit dem AES-Schlüssel auch die privaten ECC-Schlüssel aller zukünftig registrierten Webseiten entschlüsseln.

Dieser Angriff ist somit bedeutend mächtiger als der erste von uns beschriebene Angriff. Zudem zeigt dieser Angriff auch eine Schwachstelle des FIDO-Standards auf, da dieser AES-Schlüssel einen „Single Point of Failure“ darstellt.

5 Dokumentation der Angriffe und Bewertungen

5.1 AES-Leakage-Analyse

Für die Analyse der AES-Verschlüsselung wurden eigens hierfür implementierte Applets verwendet (vgl. Abschnitt 2.3), um die Kommunikation mit der Zielplattform auf ein Minimum zu beschränken. Das erste Applet ermöglicht uns einen beliebigen AES-Schlüssel zu setzen und somit den auf dem Token gespeicherten Schlüssel zu überschreiben. Anschließend kann mit Hilfe eines weiteren Applets ein Plaintext an den Token übergeben werden, der mit dem zuvor gesetzten Schlüssel verschlüsselt wird. Der Vorteil dieses Konstrukts ist in erster Linie die Isolation der AES-Verschlüsselung gegenüber den weiteren im Protokoll ausgeführten Operation, wie z.B. die Generierung von ECC-Schlüsseln und die Erzeugung einer Signatur, und dem damit verbundenen Rechen- und Kommunikations-Overhead. Um dennoch ein realistisches Szenario zu erschaffen, wurden die Orignial-AES-Parameter übernommen, d.h. das Applet verwendet ein AES-256 im CBC-Modus, wobei jeweils vier Blöcke pro Durchlauf verschlüsselt werden.

Nachfolgend werden die notwendigen Schritte beschrieben, die bei dieser Leakage-Analyse durchgeführt wurden.

5.1.1 Lokalisierung der AES-Verschlüsselung

Im ersten Schritt wurde ein Vergleich durchgeführt, um die AES-Verschlüsselung zu lokalisieren. Hierzu wurden zwei AES-256-Verschlüsselungen im CBC-Modus aufgerufen. Bei der ersten Verschlüsselung wurde ein Block verschlüsselt, d.h. der AES wurde genau ein Mal aufgerufen. Im zweiten Fall fand, wie bei der FIDO-Implementierung, eine Verschlüsselung von vier Blöcken statt. Die daraus resultierenden Stromprofile sind in Abbildung 4 und Abbildung 5 dargestellt.

Abbildung 5: Stromprofil der AES-Verschlüsselung im CBC-Modus; Anzahl der verschlüsselten Blöcke: 4

Der markierte Bereich ist in Abbildung 5 deutlich länger als in Abbildung 4. In der vergrößerten Darstellung (siehe Abbildung 6 und Abbildung 7) erkennt man, dass sich die beiden Messungen in genau drei Blöcken unterscheiden. Wir gehen daher davon aus, dass die AES-Verschlüsselung in diesem Bereich stattfindet.

Abbildung 6: Vergrößerte Darstellung des in Abbildung 4 markierten Bereichs

Abbildung 7: Vergrößerte Darstellung des in Abbildung 5 markierten Bereichs

5.1.2 EM-Scan

Aufgrund fehlender Layout-Informationen stellte sich die Positionierung der EM-Sonde als relativ schwierig heraus. Ein Ansatz, um eine geeignete Messposition festzulegen, ist die Durchführung eines EM-Scans. Hierbei wird nach und nach die Chipoberfläche mit einer EM-Sonde abgetastet, während an jeder Position eine AES-Verschlüsselung berechnet wird. Anschließend wird das Verhältnis von Signal zu Rauschen ermittelt und eine Position gewählt, an der dieses Verhältnis maximal ist.

Abbildung 8 zeigt das Ergebnis des Scans, wobei sich die Suche lediglich innerhalb der synthetisierten Logik beschränkte. Ein blauer Punkt stellt hierbei das schlechteste Verhältnis dar, während eine Verschiebung ins (dunkel-)rote ein hohes Signal-zu-Rauschen-Verhältnis markiert.

Abbildung 8: Ergebnis des EM-Scans während der Ausführung einer AES-Verschlüsselung

Eine Beispielmessung, die laut EM-Scan an einer vielversprechenden Position aufgenommen wurde, ist in Abbildung 9 dargestellt.

Abbildung 9: EM-Messung an einer Position mit hohem Signal-zu-Rauschen-Verhältnis

Eine AES-256-Verschlüsselung wird in 14 Runden ausgeführt. Versucht man diese in der EM-Messung aus Abbildung 9 wiederzufinden, stößt man entweder auf zu viele oder zu wenige Runden. Auch erscheinen die aufgenommenen Muster nach unseren bisherigen Erfahrungen „zu glatt". Nichtsdestotrotz wurden 200.000 Messungen aufgenommen. Die einzelnen Patterns wurden zerschnitten, synchronisiert und analysiert. Ein erfolgreicher Angriff konnte allerdings nicht durchgeführt werden.

Bei einer erneuten (diesmal händisch-geführten) Abtastung der Chipoberfläche fiel ein Bereich auf, der bei dem EM-Scan scheinbar vernachlässigt wurde. Im Bereich von Sample 260.000 (vgl. Abbildung 9) war an einigen wenigen Positionen ein Pattern erkennbar, der auf eine mögliche AES-Verschlüsselung hindeutete. Die Messposition ist in Abbildung 10 zu sehen.

Abbildung 10: Position der EM-Sonde während der Messung

Abbildung 11 zeigt eine Messung an der ausgewählten Position.

Abbildung 11: Beispielmessung einer möglichen AES-Verschlüsselung

Abbildung 12: Vergrößerte Darstellung des Endbereichs der vermuteten AES-Verschlüsselung; mögliche Sbox-Aufrufe rot-umrandet

5.1.3 Synchronisation

Da das aufgenommene Pattern sich sehr stark im zeitlichen Verlauf von Messung zu Messung verschoben hatte, war eine Synchronisation notwendig. Diese wurde in mehreren Schritten durchgeführt:

Im ersten Schritt diente ein auffälliges Muster innerhalb der Messung dafür, dass eine grobe Synchronisation sichergestellt wird. Nachdem wir uns entschieden haben, die letzte Runde der AES-Verschlüsselung anzugreifen, welche auf dem ersten Blick weniger Störsignale beinhaltete, fand eine zweite Synchronisation am Ende des Patterns statt. Zum Schluss wurde versucht, die einzelnen (vermuteten) Sbox-Aufrufe einzeln auszurichten. Die Messungen wurden in Teilbereiche unterteilt, auseinandergeschnitten, synchronisiert und letztendlich wieder zusammengefügt. Die daraus resultierenden Messungen wurden anschließend analysiert.

5.1.4 Kollisionsanalyse

Die Kollisionsanalyse basiert auf der Annahme, dass eine Sbox-Implementierung, die mehrmals verwendet wird, unter identischen Voraussetzungen, ein ähnliches Leakage verursacht. Unter dieser Annahme wäre z.B. die elektromagnetische Abstrahlung zweier Sbox-Berechnungen ähnlich, falls diese einen identischen Input vorweisen. D.h., bei zwei Plaintexten *p0* und *p1* und zwei Schlüsselbytes *k0* und *k1* existiert eine Ähnlichkeit genau dann, wenn

$$p0 \text{ xor } k0 = p1 \text{ xor } k1.$$

Diese Eigenschaft kann von einem Angreifer ausgenutzt werden, um die Differenz *delta* zweier Schlüsselbytes zu berechnen. Dabei werden Hypothesen für diese Differenz erstellt und zu jedem Plaintext *p0* ein entsprechendes

$$p1 = p0 \text{ xor } delta$$

berechnet. Es existiert allerdings nur ein *delta* für die die erste Gleichung erfüllt ist. Falls es möglich ist, das entsprechende *delta* durch eine Ähnlichkeitsanalyse zu erkennen, ist der Angriff erfolgreich und die Differenz zwischen den beiden Schlüsselbytes ist gefunden. Nach der Berechnung aller Differenzen muss lediglich ein Schlüsselbyte bestimmt werden, um alle anderen Bytes zu berechnen. Es bleibt daher eine Restkomplexität von einem Byte übrig. Details zu diesem Angriff sind zu finden unter [MoMiEi10].

Die folgende Abbildung zeigt einige ausgewählte Ergebnisse unserer Analyse.

Abbildung 13: Teilergebnisse aus der Kollisionsanalyse

5.2 Deinterleave-Leakage-Analyse

5.2.1 Lokalisierung der Deinterleage-Funktion

Wie bereits in Kapitel 4.2 beschrieben scheint die *deinterleave*-Funktion ein gutes Angriffsziel für eine Template-Analyse zu sein. Für die Analyse der Funktion wurde ein eigenes Applet geschrieben, welches den direkten Aufruf der *deinterleave*-Funktion erlaubt. Für unseren Test übergeben wir ein 12 Byte *key handle* (statt 64 Byte) und bekommen einen 6 Byte ECC privaten Schlüssel zurück (statt 32 Byte). Um das 12 Byte *key handle* zu entwürfeln, muss die for-Schleife der *interleave*-Funktion 6 mal durchlaufen werden. In jedem Durchlauf werden 4 Operationen ausgeführt. Somit erwarten wir, im Seitenkanalprofil 24 Operationen identifizieren zu können. Abbildung 15 zeigt das Stromprofil einer Messung.

Abbildung 15: Stromprofil der deinterleave-Funktion

Abbildung 16 zeigt das entsprechende EM-Profil. Auch hier wurde vor Aufnahme der Messungen der gesamte Logikbereich des Chips abgescannt. Das Ergebnis gleicht exakt unserem EM-Scan aus Abbildung 8.

Abbildung 16: EM-Profil der deinterleave-Funktion

Abbildung 17: Vergrößertes EM-Profil

5.2.2 Korrelationsanalyse

Als erstes wurde eine Korrelationsanalyse [BrClOl04] auf 10.000 Messungen durchgeführt, zum einen um zu untersuchen, ob es Seitenkanalinformationen gibt, und falls ja, um die zeitlichen Positionen zu bestimmen. Des Weiteren können wir durch diese Analysen bestätigen, dass die 24 Blöcke wirklich den 24 identifizierten Operationen entsprechen. Falls dem so ist, sollte sich dies in den zeitlichen Positionen des auftretenden Leakages widerspiegeln.

Als Leakagemodell wurde, wie bei Softwareimplementierungen üblich, das Hamminggewicht des anzugreifenden Wertes gewählt. Abbildung 18 zeigt die Analyseergebnisse für die ersten zwei Bytes des *key handles* im Stromprofil.

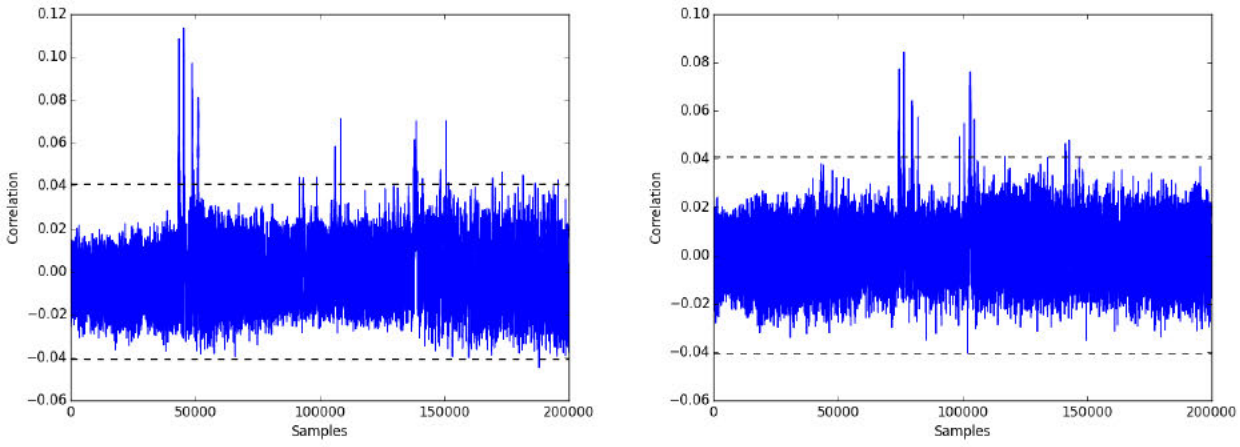


Abbildung 18: Ergebnisse der Korrelationsanalyse des Stromprofils; Byte 1 und 2 des key handles

Wie man sehen kann erhalten wir eindeutige Korrelationspeaks. Des Weiteren liegen die Peaks für Byte 1 und 2 ungefähr 25.000 Samples voneinander entfernt, was genau dem Abstand der einzelnen Blöcke entspricht. Somit sehen wir unsere Vermutungen bestätigt, dass jeder Block im Seitenkanalprofil einer

Operation innerhalb der for-Schleife entspricht. Allerdings ist das Leakage mit ca. 0,1 nicht sehr ausgeprägt. Abbildung 19 zeigt die Analyseergebnisse für das EM-Profil.

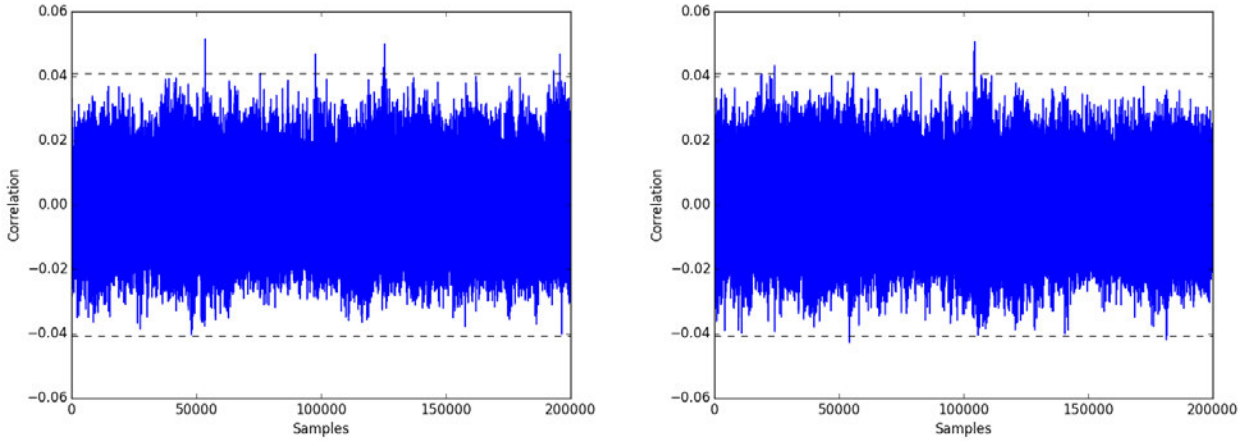
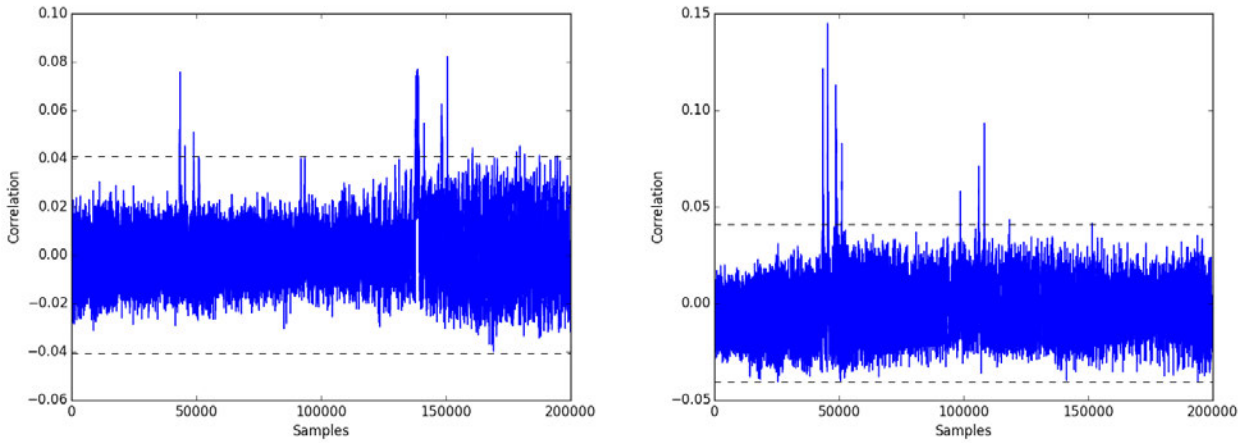


Abbildung 19: Ergebnisse der Korrelationsanalyse des EM-Profils; Byte 1 und 2 des key handles

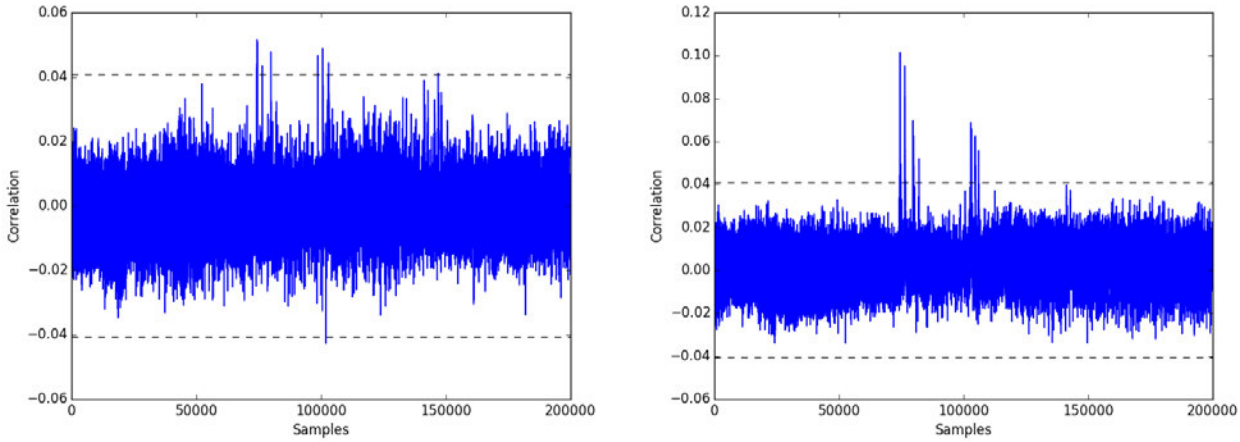
Auch hier erkennen wir Korrelationspeaks, allerdings sind diese deutlich kleiner als in der Analyse des Stromprofils. Eine Analyse mehrerer verschiedener Positionen, sowie das Komprimieren der Messungen um kleine Timingschwankungen auszugleichen, verbesserten das Ergebnis nicht.

Da von der Funktion auch auf Nibble-Ebene gearbeitet wird, führten wir noch eine Korrelationsanalyse auf das Hamminggewichts der einzelnen Nibbles der ersten 2 Eingangsbytes durch. Abbildung 20 zeigt die Ergebnisse.



Byte 1, höherwertiges Nibble

Byte 1, niederwertiges Nibble



Byte 2, höherwertiges Nibble Byte 2, niederwertiges Nibble

Abbildung 20: Korrelationsanalyse auf das Hamminggewicht einzelner Nibble

Wie zu erwarten sehen wir eindeutige Korrelationspeaks, wobei die niederwertigen Nibble der beiden Bytes ein höheres Leakage zeigen.

Unsere Ergebnisse zeigen also, dass Daten, die von der Karte verarbeitet werden, mit einer Korrelationsanalyse angegriffen werden können, wenn es das Angriffsszenario erlaubt. Zur Info: dieser Umstand wird normalerweise von der Security Guidance eines zertifizierten Produkts abgedeckt.

In unserem Fall jedoch werden immer die gleichen Daten (zu einer Webseite gehört ein fester privater ECC-Schlüssel) verarbeitet, und diese werden auch nicht mit bekannten sich ändernden Daten kombiniert, wie es Voraussetzung für einen Korrelationsangriff ist. Deshalb bleibt uns hier nur die Möglichkeit, einen Templateangriff [ChRaRo02] auf die Eingangsdaten oder die Ausgangsdaten durchzuführen.

5.2.3 Templateanalyse

Da das Hamminggewicht des niederwertigen Nibbles des ersten Bytes die höchsten Korrelationswerte aufzeigte, wählten wir diesen Wert als Ziel für unseren Templateangriff. Für die Templateanalyse nutzten wir 8000 Messungen, um 5 Templates zu bauen. In der Matchingphase wurden die restlichen 2000 Messungen dann den Templates zugewiesen. Anschließend wurde aus den Ergebnissen die Erfolgswahrscheinlichkeit für einen Angreifer berechnet. Da in einer Templateanalyse nur eine begrenzte Anzahl an Punkten genutzt werden kann (aufgrund numerischer Einschränkungen), wurde das sogenannte SOST-Verfahren (Sum Of Squared pairwise T-differences) eingesetzt, um relevante Punkte zu identifizieren. Abbildung 21 zeigt ein Stromprofil, den berechneten SOST-Graphen und die ausgewählten Punkte für die Templateanalyse.

Abbildung 21: Stromprofil (oben) und SOST-Graph (unten)

Hohe SOST-Werte bedeuten eine hohe Signifikanz. Dementsprechend wurden 12 Punkte mit den höchsten SOST-Werten für die Templateanalyse ausgewählt.

Die Erfolgswahrscheinlichkeit das Hamminggewicht eines Wertes korrekt zu bestimmen beträgt 25% (eine zufällige Zuordnung entspricht einer Wahrscheinlichkeit von 20%). Die Erfolgswahrscheinlichkeit ist somit leicht erhöht, aber nicht ausreichend für einen kritischen Angriff. Außerdem wird bei diesem Angriff nur das Hamminggewicht bestimmt, nicht der Wert selbst. Es ist fraglich ob das Hamminggewicht genügend Informationen liefert, um den privaten ECC-Schlüssel zu berechnen (uns ist kein derartiger Angriff bekannt).

Aus diesem Grund führten wir einen weiteren Templateangriff durch, diesmal auf den Wert des Nibbles. Somit brauchen wir 16 Templates. Auch hier erhalten wir eine leicht erhöhte Erfolgswahrscheinlichkeit (8% gegenüber 6% bei einer zufälligen Zuordnung), die jedoch nicht als kritisch eingestuft werden kann.

Alle Templateanalysen wurden zusätzlich noch mit einem anderen Punktauswahlverfahren wiederholt, der sog. Hauptkomponentenanalyse. Allerdings konnte durch diese Methode keine Verbesserung der Erfolgswahrscheinlichkeit erzielt werden.

5.3 ECC-Leakage-Analyse

Wie in Abbildung 3 dargestellt generiert das FIDO-Applet eine Signatur, um sich damit bei einem Host zu authentisieren. Diese Signatur ist als ECDSA implementiert. Bei einer Signatur-Generierung wird ein flüchtiger Schlüssel generiert, der im weiteren Verlauf für eine Skalarmultiplikation mit dem Basispunkt und nach einer Invertierung für eine modulare Multiplikation verwendet wird. Wenn man von diesem flüchtigen Schlüssel eine bestimmte Anzahl an Bits sicher bestimmen kann, so kann der geheime Schlüssel berechnet werden.

Es ist also notwendig, dass bei der ausgeführten Skalarmultiplikation und darauffolgender Inversion des flüchtigen Schlüssels der flüchtige Schlüssel nicht leckt. Wir konzentrieren uns im Folgenden auf eine Analyse der Implementierung der Skalarmultiplikation.

Da nichts über die Implementierung bekannt ist, müssen ein paar Annahmen getroffen werden, um den Untersuchungsraum etwas einzugrenzen. Es ist davon auszugehen, dass die Ausführung der Skalarmultiplikation nicht von der Protokollfunktion (d.h. ECCDSA, ECDH, etc.) abhängig ist, sondern alle diese Funktionen intern auf die gleiche Implementierung für die Skalarmultiplikation zurückgreifen.

Da der flüchtige Schlüssel für die ECC-Signaturgenerierung intern generiert wird und mit den API-Funktionen nicht ausgegeben werden kann, ist eine Analyse anhand dieser Funktion äußerst schwierig. Da wir davon ausgehen, dass alle Operationen auf die gleiche Implementierung der Skalarmultiplikation zurückgreifen, verwenden wir für unsere Analyse eine Funktion, bei der wir den Skalar (bis auf eine mögliche interne Randomisierung) vorgeben können. Hierfür haben wir ein Applet programmiert, was es uns erlaubt mithilfe der JavaCard-Funktion generateSharedSecret eine Skalarmultiplikation mit einem von uns übergebenen Skalar zu berechnen. Dieser Funktion übergeben wir zufällig gewählte Skalare, werden dann eine Korrelationsanalyse auf die ersten Bits dieser Skalare durchführen und dann anhand eines Template-Angriffs überprüfen, ob wir am EM- oder Stromprofil ein verarbeitetes Bit vorhersagen können.

Unser Applet benutzt folgende JavaCard-API-Operationen: Es kopiert den in der APDU übergebenen Skalar in einen Array und initialisiert damit ein Objekt vom Typ ECPrivateKey. Dieses Objekt ist mit einem anderen Objekt vom Typ KeyPair verlinkt. Durch Aufrufen der Funktion init() initialisieren wir das KeyPair-Objekt mit dem privaten Schlüssel. Zuletzt rufen wir die Funktion generateSharedSecret auf, um aus dem privaten Schlüssel ein Schlüsselpaar zu generieren. Danach kopiert die Operation den generierten öffentlichen Schlüssel in den IO-Buffer und sendet ihn an das Terminal zurück. Das Stromprofil dieser Funktionsfolge ist in Abbildung 22 dargestellt.

Abbildung 22: Stromprofil der generateSharedSecret-API-Funktion

[redacted]

Abbildung 23: Vergleich der generateSharedSecret Funktionen auf einer 256 und eine 112-bit Kurve.

Abbildung 24: Ausschnitt aus den synchronisierten ECC Traces

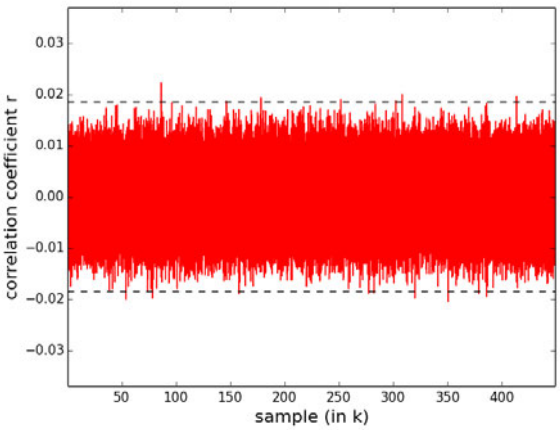
Das Problem mit den ECC-Messungen ist, dass sie zwar auf eine Art regulär sind in dem sich eine bestimmte Struktur wiederholt, diese allerdings sehr unterschiedlich sind (siehe Abbildung 24 und noch deutlicher

Abbildung 25). An der Struktur rechts in Abbildung 25 ist deutlich zu erkennen, dass sich die Traces in ihrer Ausführungslänge unterscheiden. Dies macht es ohne Informationen über die interne Implementierung extrem schwer die Messungen zu synchronisieren, weil es zu viele Punkte gibt, auf die wir synchronisieren könnten.

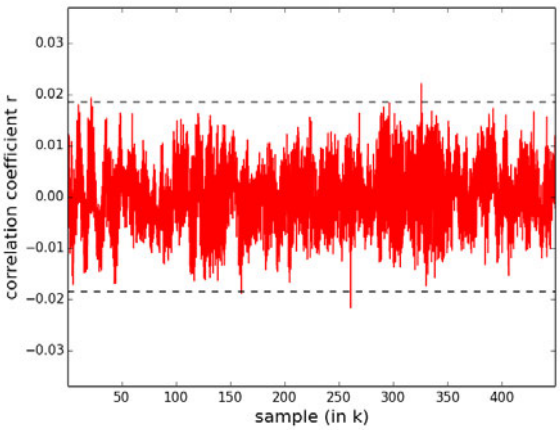
Abbildung 25: Zoom in die synchronisierten Traces aus Abbildung 24

5.3.1 Korrelationsanalyse

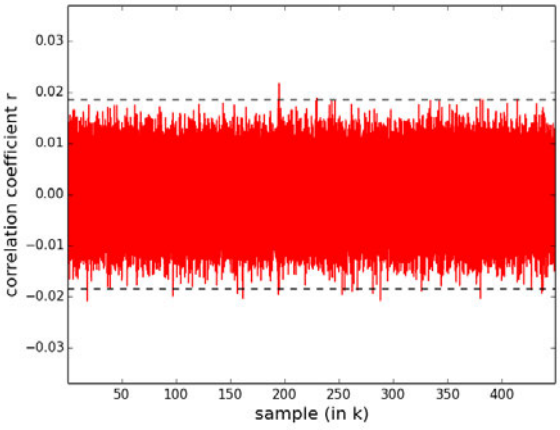
Trotz der geringen Erfolgsaussichten, haben wir auf den Messungen eine Korrelationsanalyse durchgeführt. Da wir nicht wissen, ob der implementierte Skalarmultiplikationsalgorithmus zuerst die höchstwertigsten oder zuerst die niederwertigsten Bits des Skalars verarbeitet, korrelieren wir auf einzelne Bits und zwar die drei höchstwertigsten und die drei niederwertigsten Bits des Skalars. Die Ergebnisse für die Strom- und EM-Messungen sind in Tabelle 2 dargestellt.



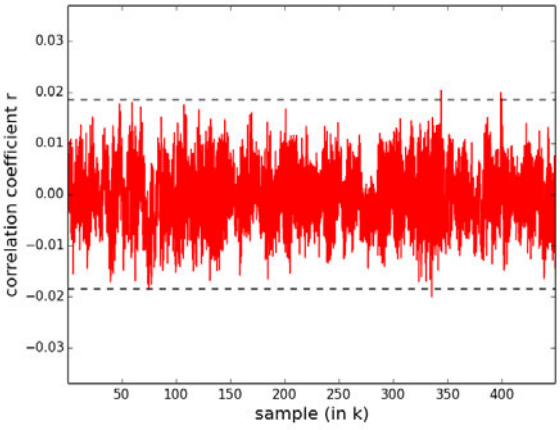
Bit 0 EM



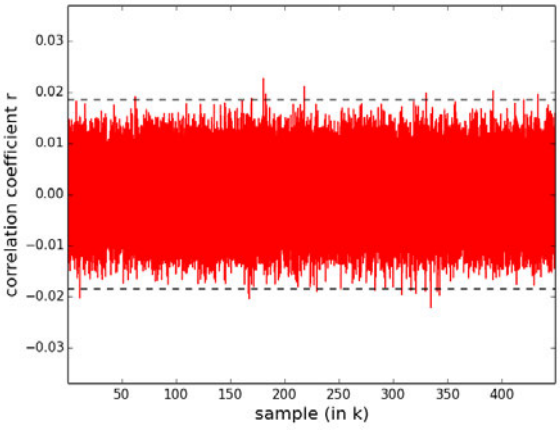
Bit 0 PWR



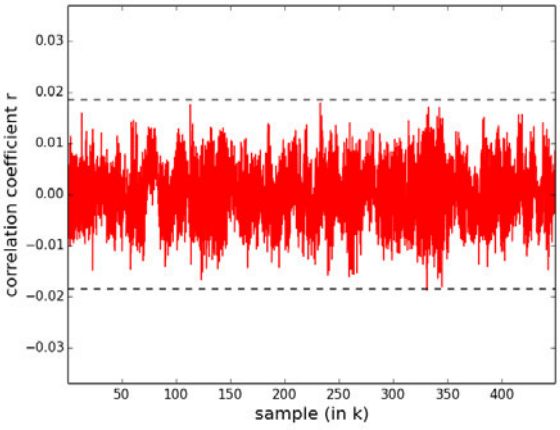
Bit 1 EM



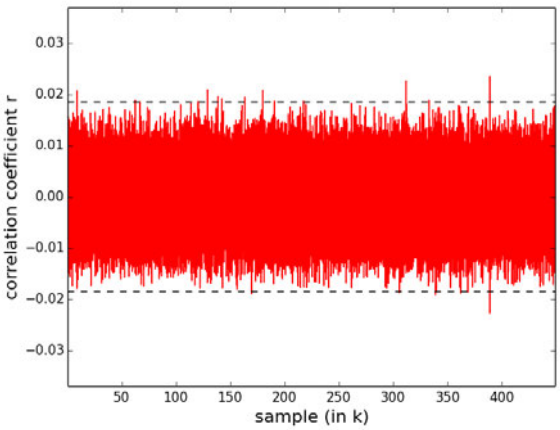
Bit 1 PWR



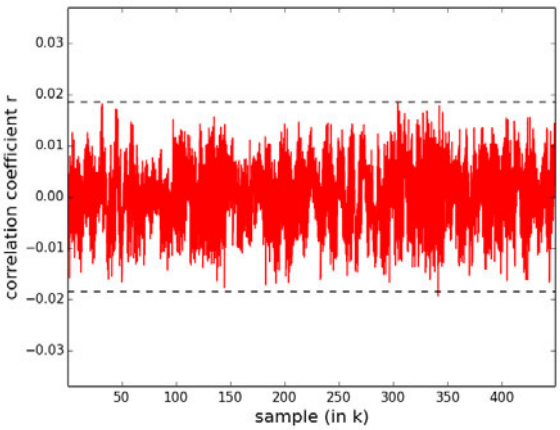
Bit 2 EM



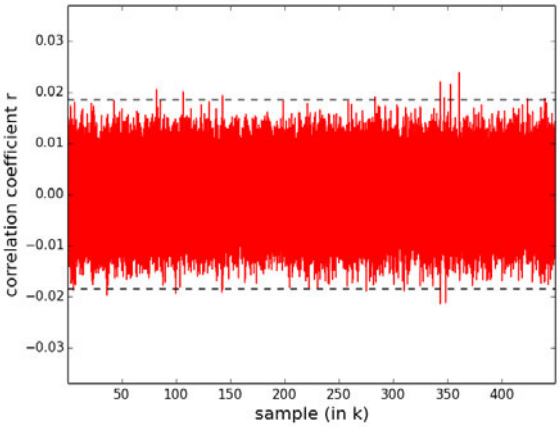
Bit 2 PWR



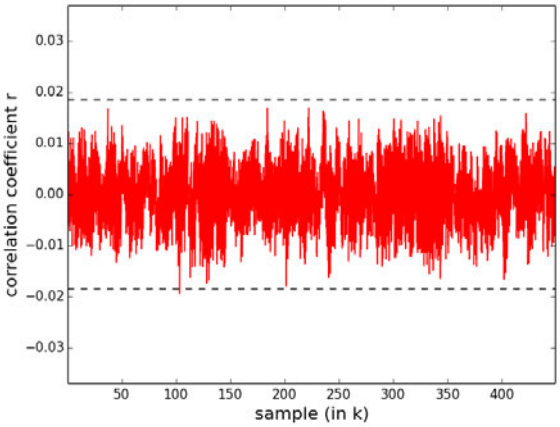
Bit 253 EM



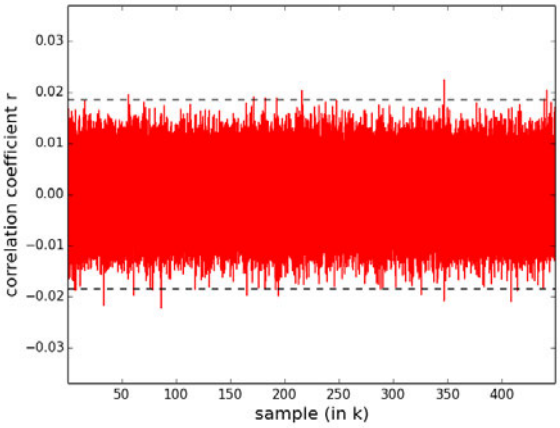
Bit 253 PWR



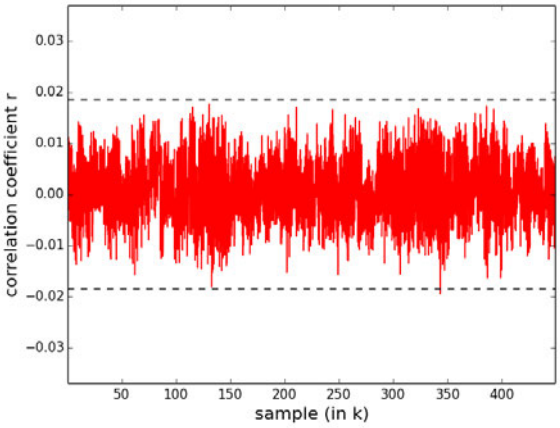
Bit 254 EM



Bit 254 PWR



Bit 255 EM



Bit 255 PWR

Tabelle 2: Ergebnisse der Korrelationsanalyse auf die Strom- und EM-Messungen. Betrachtet wurden die Anfänge der identifizierten regulären Struktur.

Die Ergebnisse deuten nicht darauf hin, dass die Messungen Rückschlüsse über die verarbeiteten Skalarbits zulassen. Es ist also nicht zu erwarten, dass eine Templateanalyse erfolgreich ist. Dies wurde durch die Ergebnisse unserer Templateanalyse bestätigt. Diese Ergebnisse haben wir nicht separat in diesem Bericht aufgeführt.

6 Zusammenfassung und Schlussfolgerung

Dieser Bericht fasst die Ergebnisse unserer Sicherheitsuntersuchung einer Java-Karte mit einem FIDO-U2F-Applet zusammen. Als erstes wurde beschrieben, wie das Open-Source-Applet kompiliert und auf die Karte geladen werden kann. Dabei wurde festgestellt, dass das Applet während der Installation zusätzliche Parameter sowie die Ausführung eines proprietären Kommandos erwartet, um personalisiert zu werden. Um die Sicherheitsuntersuchung zu vereinfachen und zu beschleunigen, wurden eigene Applets entwickelt, die es uns ermöglichten, kritische Funktionen mit selbst gewählten Parameters aufzurufen und zu analysieren. Die Quellcodes der Applets liegen dem Bericht bei.

Im nächsten Schritt wurden die Java-Karten mit Salpetersäure geöffnet. Dies ermöglichte uns die Identifizierung der verschiedenen Bereiche auf dem Chip. [redacted]

Im nächsten Schritt wurde das FIDO-Protokoll vorgestellt und analysiert. Dabei wurde untersucht, welche Assets existieren und wie diese vom Applet verarbeitet werden. Basierend aus dieser Analyse wurden anschließend Angriffsszenarien definiert, sowie deren Auswirkungen diskutiert.

Basierend auf unseren vorhergehenden theoretischen Untersuchungen wurden nun die praktischen Seitenkanal-Angriffe durchgeführt. In unseren Analysen wurde der Stromverbrauch sowie die elektromagnetische Abstrahlung betrachtet. Es wurden verschiedenste Analysemethoden verwendet, um AES, ECC und die Deinterleave-Funktion anzugreifen.

Die Untersuchung der Deinterleave-Funktion hat gezeigt, dass die Java-Karte mit differenziellen Analysemethoden angegriffen werden kann. Bei der Programmierung eigener Applets ist also, trotz Verwendung eines Sicherheitscontrollers, durchaus Vorsicht geboten, wenn Operationen auf geheimen Daten durchgeführt werden. In Falle des FIDO-Applets werden die geheimen Daten jedoch in einer Art und Weise verarbeitet, die nur ein Templateangriff zulässt. Trotz verschiedener Methoden war dieser Angriff nicht erfolgreich.

Die Untersuchung der ECC Implementierung haben ergeben, dass die Ausführung einer Skalarmultiplikation keinen Aufschluss über einzelne verarbeitete Bits zulässt. Dies war zu erwarten, da eine Analyse schwierig ist, wenn z.B. Skalarblinding implementiert ist. Desweiteren sind die Traces sehr lang und eine Synchronisation ohne jegliche Informationen über die Implementierung ist äußerst schwierig. Ein Angriff auf die Skalarmultiplikation war nicht erfolgreich.

[redacted]

Schlussfolgernd kann gesagt werden, dass eine Implementierung auf einem [redacted]) Sicherheitscontroller keine Garantie für die Sicherheit des Gesamtprodukts bietet. [redacted]

Anhang

Literaturverzeichnis

[BrClOl04] Brier, Eric, Christophe Clavier, and Francis Olivier, *Correlation power analysis with a leakage model*, International Workshop on Cryptographic Hardware and Embedded Systems. Springer Berlin Heidelberg, 2004.

[ChRaRo02] Chari, Suresh, Josyula R. Rao, and Pankaj Rohatgi, *Template attacks*, International Workshop on Cryptographic Hardware and Embedded Systems. Springer Berlin Heidelberg, 2002.

[FIDO15] FIDO Alliance, *U2F Specifications*, https://fidoalliance.org/specifications/overview/, 2015-05-14

[MoMiEi10] Moradi, Amir, Oliver Mischke, and Thomas Eisenbarth, *Correlation-enhanced power analysis collision attack*, International Workshop on Cryptographic Hardware and Embedded Systems. Springer Berlin Heidelberg, 2010.

Stichwort- und Abkürzungsverzeichnis