目次

本誌は過去に発行した記事を一部加筆・修正の上で再録したものです。記事の内容は特に言及がない限り全て執筆時点のものであり、最新の情報とは異なる場合があります。

各記事の初出は以下の通りです。

記事	初出巻数	初回発行日・イベント
公平なランサムウェアプロトコル	vol.4	2017-08-13 C92
Secure Grouping Protocol using Mental Poker	vol.3	2016-12-31 C91
BLE の通信を覗いてみよう	vol.8	2021-12-31 C99A
BLE の通信を覗いてみよう - S 社編	vol.11	2023-08-13 C102
とある RISC-V の Voltage Fault Injection（C100 版）	vol.9	2022-08-13 C100
ファミコンミニと Linux	vol.3	2016-12-31 C91

公平なランサムウェアプロトコル

1 はじめに

ランサムウェア（*Ransomware*）とは悪意のあるプログラムの 1 つです。感染すると被害者のコンピューターに保存されているデータを暗号化し、それを復号する対価として Bitcoin などのお金を要求するプログラムです。最近では KADOKAWA・ニコニコ動画がランサムウェアによる攻撃を受けました[*1]。ところが単純な疑問として、ランサムウェアの指示通りに Bitcoin などを送金したとして、はたして暗号化されたデータをきちんと復号してもらえるのでしょうか？ つまり従来のランサムウェアはお金もデータも失ってしまう可能性があるといえます。

ランサムウェアの作者であり Bitcoin を得たいアリスと、アリスのランサムウェアに感染してデータを暗号化されてしまったボブがいるとします。ボブはアリスへ Bitcoin を支払ってでもデータを復号したいですが、この 2 人の間には次のような不正が考えられます。

アリスの不正 アリスはボブから Bitcoin を受け取ったにも関わらず、暗号文を復号するための鍵を渡さない
ボブの不正 ボブはアリスが暗号文を復号するための鍵を渡したにも関わらず、Bitcoin を送金しない

一般的に、ランサムウェアの交渉はランサムウェアの作者であるアリスの方が優位であるため、アリスの不正が発生しがちです。本稿ではこのようなアリスとボブの不正を高い確率で防止するための**公平なランサムウェアプロトコル**について考えます。また、このプロトコルは Bitcoin の性質と暗号技術を巧みに利用しているため、信頼できる第三者を必要としません。ランサムウェアの作者であるアリスは警察などを利用することはできませんし、被害者ボブがたとえ警察などに相談したとしても警察が違法組織との取引を仲介することはないでしょう。ランサムウェアのような違法なビジネスにおいて、信頼できる第三者を利用しなくてよいというのは非常に重要なことだと考えられます。なお、本稿は Qiita で公開した記事 [1] を大幅に加筆・修正したものです。

1.1 本稿の構成

本稿の構成について説明します。まず 2 節でプロトコルの構成に必要な暗号技術について詳細に説明し、次に 3 節でプロトコルの前提となるアリスとボブの状況を整理します。4 節では筆者の提案する公平なランサムウェアプロトコルの詳細について説明し、最後に 5 節で全体的なまとめを述べて、最後に参考文献を紹介します。

2 暗号技術

この節ではプロトコルを構成するための暗号技術について詳細に解説します。

2.1 対称鍵暗号

対称鍵暗号とは暗号化と復号で同じ鍵を利用する暗号技術です。対称鍵暗号は**平文**と呼ばれる任意長のデータを**対称鍵**というデータによって暗号化します。対称鍵 k を用いてデータ x を暗号化するとき、本稿では Enc を暗号化関

[*1] 本誌を執筆する際に加筆を行いました。

数として**暗号文**を $\mathrm{Enc}_k(x)$ のように表記します。

また、復号関数 Dec を利用して暗号文を復号することができるものとします。ただし、復号の際に利用する鍵が暗号化の際に利用した鍵と等しくなければ復号することはできません。つまり次のような式（1）が成り立ちます。

$$x = \mathrm{Dec}_k(\mathrm{Enc}_k(x)) \tag{1}$$

また一般的な対称鍵暗号は複数の鍵 k_1, k_2 の暗号化と復号の順序を入れかえた場合、次のように正しく動作しません。

$$\mathrm{Dec}_{k_2}(\mathrm{Dec}_{k_1}(\mathrm{Enc}_{k_2}(\mathrm{Enc}_{k_1}(x)))) \neq x$$

対称鍵暗号の特徴は後述する RSA 暗号のような公開鍵暗号と比べて高速なことです。対称鍵暗号の最も有名な実装として AES があります。他にも最近は ChaCha20 など色々な実装が提案されています。本稿のプロトコルでは AES や ChaCha20 などといったどの実装を選択しても構いませんが、統一的に対称鍵暗号の暗号化関数を Enc とし、また復号関数を Dec で表記します。

2.2 ハッシュ関数

ハッシュ関数とは任意長の入力データに対して、固定長のデータを出力する関数です。ハッシュ関数を H としてデータ x の返り値 $H(x)$ を**ハッシュ値**と呼びます。また、あるハッシュ値 $h := H(m)$ の入力 m のことをハッシュ値 h の**原像**と言います。ハッシュ関数には次の性質を満たします。

原像計算困難性 与えられたハッシュ値 h に対して、$h = H(m)$ となる m を見つけることが困難である。
第二原像計算困難性 m_1 が与えられたとき、$H(m_1) = H(m_2)$ となる m_2 $(\neq m_1)$ を見つけることが困難である。
衝突困難性 $H(m_1) = H(m_2)$ となる相違な m_1, m_2 を見つけることが困難である。

ハッシュ関数の具体的な実装として有名なものに SHA-1[*2]や SHA-2、MD5 などがあります。このプロトコルにおいては後述する Bitcoin のスクリプト上で実行できる必要があるため、ハッシュ関数の実装は SHA-1、SHA-256 あるいは RIPEMD-160 でなければなりません。本稿ではハッシュ関数を H と書き、これは SHA-1、SHA-256 または RIPEMD-160 のいずれかであるものとします。

2.3 RSA 暗号

RSA **暗号** [3] とは**公開鍵暗号**の 1 つです。公開鍵暗号とは 2.1 節の対称鍵暗号とは異なり、暗号化と復号にそれぞれ別の鍵を利用する暗号化方式です。暗号化のために利用する鍵を**公開鍵**と言い、復号に利用する鍵を**秘密鍵**と言います。RSA 暗号では公開鍵を (e, N) として、整数の平文 x を暗号化する操作は次のようになります。

$$x^e \bmod N$$

このとき、N は互いに異なる巨大な素数 p, q の積で $N = p \cdot q$ となります。また e は 65537 が慣例として用いられます。

RSA 暗号の特徴として、N の素因数 p, q を知る者にとっては次のような式を満す秘密鍵 d を容易に求めることができます。

[*2] Google による “SHAttered”[2] では 2 つの異なる PDF ファイルが同じ SHA-1 のハッシュ値を持つ例が示されました。潤沢なリソースを前提とするものの、第二原像計算困難性を突破できることを前提に SHA-1 を利用するかどうかを決める必要があります。

Secure Grouping Protocol using Mental Poker

1 はじめに

離れた所にいるプレイヤーがポーカーなどを公平な第三者なしにプレイするための方法として、*Mental Poker* というものがあります。Mental Poker はシャッフル・ドロー・カードの公開という 3 つの操作を公平な第三者なしに提供する暗号技術です。一方で Mental Poker では「人狼」のようなゲームはプレイできないと考えられていました。

人狼は「村人」と「人狼」にプレイヤーをグループ分けして行うゲームです。村人になったプレイヤーは他のプレイヤーが村人か人狼のどちらに所属しているのか判定できてはならない一方で、人狼になったプレイヤーはどのプレイヤーが人狼であってどのプレイヤーが村人であるかを識別できます。このような柔軟なグループ分けを公平な第三者なしで実行する方法がなかったため、人狼は従来 Mental Poker の道具だけでは実行できないと考えられていました。

2016 年に**秘匿グループ分けプロトコル**（*Secure Grouping Protocol*）[13] が提案されました。このプロトコルを用いることで、人狼のように複雑なグループ分けが必要なゲームを公平な第三者なしで実現できるのではないだろうかと研究が進められています。

本稿ではまず 2 節と 3 節で Mental Poker について説明し、次に 4 節で置換群の基礎を解説し、この置換群の性質を利用して 5 節で秘匿グループ分けプロトコルについて説明します。そして最後に 6 節で Mental Poker と秘匿グループ分けプロトコルの関係について述べます。

2 直感的な Mental Poker

Mental Poker は電話やインターネット越しに公平なカードゲームをするために考案されました。電話越しなのでまずシャッフルが信用できません。たとえば山札をプレイヤーが順番に回してシャッフルしていく場合、最後にシャッフルするプレイヤーは実際にシャッフルせず、自分にとって都合のよい順番に並び換える可能性があります。さらに、山札からカードを引く操作（ドロー）について考えても、悪意あるプレイヤーは自分にとって有利なカードが引けるまで引いては山札に戻すかもしれません。またポーカーのルールによってはプレイヤーの 1 人がカードを 1 枚引いてそれを全員に公開する、という操作があります。このプレイヤーが誠実ではない場合、表にしたカードを自分に都合のよいものに操作する可能性があります。どのようにしたらこのような問題を防ぐことができるでしょうか。

2.1 箱と南京錠を用いたプロトコル

電話やインターネットにおける説明はやや数学的で大変なので、まずは物理的な箱と南京錠、そしてカードを用いて離れた場所にいるアリスとボブがシャッフル・ドロー・公開という 3 つの操作を不正なく行う方法について考えます。

ここで言う “箱” というのは、外側からは何が入っているのかが全く分からない入れ物のことです。また箱には任意の数の南京錠を取り付けることができます。箱に自分の取り付けた南京錠しかない場合、自分の持っている鍵を使って南京錠を取り外し中身を見ることができますが、1 つでも他人の南京錠が取り付けられている箱を開けることはできません。

このような箱と南京錠を用いて、まずはシャッフルについて説明します。アリスはカードを 1 枚ずつ箱の中に入

れ、それに南京錠をかけた後で箱をシャッフルします。この時アリスは箱の順序を恣意的に操作する可能性がありますが、そうしたとしても問題はありません。そして、アリスは全ての箱を宅配便などを使ってボブに送ります。ボブに送られた箱には全てアリスの南京錠が取り付けられているので、ボブは箱を開けることができません。ボブはこの箱のひとつひとつにボブの南京錠を取り付けて、シャッフルします。すると箱にはアリスの鍵とボブの鍵が取り付けられたことになります。このようにすることでアリスがたとえ恣意的な順番に箱を並べたとしても、ボブにシャッフルされてしまうため意味がありません。また、ボブは箱の中身を見ることができないため、たとえ箱を恣意的な順番に並べたとしても意味がありません。こうして公平なシャッフルを達成します。

次にドローです。アリスが山札から 1 枚のカードを選んでドローすると考えます。なおここで言う山札とは先ほど述べたシャッフルが終わった後の箱の束を指します。今、シャッフルが終わった時点で山札はボブのところにあるので、まずはボブから全ての箱を送ってもらいます。そして、アリスはその中から 1 つを選びその箱をボブへ送ります。この箱にはアリスとボブの南京錠が合計で 2 つ取り付けられています。ボブはこの箱を受け取り、自分が取り付けた南京錠を取り外します。この箱にはアリスの南京錠が取り付けられていますから、ボブが箱の中身を知ることはできません。この箱をボブはアリスへ送ります。そしてアリスはこの箱から自分の南京錠を取り外すことで、箱の中身を得ることができます。こうすることでアリスはボブに知られることなくカードをドローすることに成功します。

最後の公開についてはたとえばアリスが引いたカードをボブに公開する場合、アリスはカードをボブに送るだけです。

そして、最後に不正がおきていないことを検証するため、アリスとボブは自分の南京錠の鍵を全て公開して箱の中身を確認します。

2.2 直感的な Mental Poker の課題

これでシャッフルやドローといった操作ができるわけですが、いくつか課題があります。まず Mental Poker の目的は電話やインターネットを用いることですから、箱や南京錠を用いて達成してもそれは電話などでは使えません。

また、この方法では最後に箱を全て開けて中身を確認して不正がなかったかどうかを判断しています。そうすると各プレイヤーの手札を全て公開する必要もあり、これによってプレイヤーの戦略が明らかになってしまう可能性があります。従って、手札を公開することなく不正を明らかにする必要があります。

さらにこの方法を用いて多人数でポーカーなどのゲームをプレイしたとして、悪意あるプレイヤーはゲームの状況が悪くなるとゲームから退場するかもしれません。この方法では山札のカードに全てのプレイヤーが南京錠を取り付けていますから、誰か 1 人でもいなくなるとドローができなくなり、途端にゲームの続行が不可能になります。

3 数学的な Mental Poker

この Mental Poker を初めて提案した Shamir、Rivest そして Adleman は、世界的に有名な RSA 暗号という公開鍵暗号を考案した 3 人です。彼らが Mental Poker[14] を提案した後も、多くの研究者が別の実現方法を提案しました。[15] では剰余演算の性質を利用した Mental Poker の数学的な実現方法について紹介しました。この他にも色々な方法があり、多くは [16] に載っています。また最近では**楕円曲線暗号**を用いたより効率のよい方法が提案されています。ただ数学的な方法を完全に説明するのは大変なので、ここでは直観的な説明だけをします。

3.1 暗号化と復号

まず、直観的な説明で登場した南京錠を取り付ける操作に対応するため、コンピューターや数学の世界では暗号を使います。鍵を k として平文 m を暗号化する場合、暗号化の関数を Enc として次のように書きます。

$$c := \mathrm{Enc}_k(m)$$

BLE の通信を覗いてみよう

1 注意

本稿では無線通信の傍受を取り扱いますが、すべて自身が所有・管理するデバイス同士が行う通信に対して、実験として傍受を行っています。本稿の内容を実際に試す際には、必ず自己の管理下において実施するようお願いします。他者の通信を傍受し、その存在を明かすこと、またはその内容を利用することは犯罪となる恐れがあります。

2 はじめに

本稿では Bluetooth Low Energy（以降 BLE と略します）の通信の傍受、およびその内容を追っていきます。

BLE は Bluetooth 規格の中でも低消費電力での無線通信を可能とする仕様です。BLE は Bluetooth 4.0 で定義されましたが、それ以前からある仕様（Bluetooth Basic Rate/Enhanced Data Rate。以降 BR/EDR と略します）とは全く別物であり、両者に互換性はありません。BLE は BR/EDR を置き換えるものではなく、メーカーが設計するデバイスの目的に応じてどちらかを選べるようになっています[*1]。基本的に BLE は単発の通信で小さなデータをやりとりすることに向いている一方、BR/EDR はデータを長時間やりとりし続けるストリーミングに向いています。また、BLE は消費電力を抑えるために BR/EDR に比べて通信速度や通信距離が制限されています。

そんな BLE ですが昨今流行の IoT デバイスと呼ばれるものの無線通信規格としてよく使われています。この手のデバイスの主な動作というのは、例えばリモコンとして他のデバイスに命令を送る、スマートフォンと接続してアプリからの命令を受け取る、デバイスに備えたセンサーから得られたデータを定期的に PC 等に送る、といったものが挙げられます。これらはいずれも単発で小さなデータをやりとりしています。また、IoT デバイスの多くは電池で稼働するためできるだけ省電力であることが求められます。

その他には近接検知に使われることもあります。ビーコンと呼ばれる小型デバイスからアドバタイズパケットを発信して、スマートフォンなどでこのパケットを受信できたら「ビーコンの通信範囲内にいる」ということがわかります。これによって、GPS 情報を使わずに人やモノが近くにあるということがわかります。AirTag や Tile などの忘れ物タグはこのような仕組みで動いています。また、接触確認アプリ COCOA[*2]はこれをスマートフォン同士で行い、人と人がどれだけの時間近くにいたかを判定しています。

さて、BLE も無線通信である以上、技術的にはその通信内容は誰にでも見られてしまいます。BLE のプロトコル上で暗号化は定義されているのですが、あくまでオプションです。今回は IoT デバイスを対象に、実際に通信を見ることができるのか試してみます。

[*1] ホスト側は必要に応じて BLE と BR/EDR の両方に対応させることができます。

[*2] 実際にはその基盤である Exposure Notifications System。

3 準備

3.1 ハードウェア調達

まずは通信を傍受するためのデバイスを用意する必要があります。今回は 1 対 1 で通信しているところを傍受するので、「スニッファー」と呼ばれる専用のハードウェアが必要になります。今回は”Bluefruit LE Sniffer” を使用します。“Bluefruit LE Sniffer” は日本ではスイッチサイエンスや千石電商で手に入ります。スイッチサイエンスの通販では 3,916 円で入手できます[*3]。

次に傍受の対象となるデバイスを用意します。今回題材としたのはカーテンを自動で開閉できるようにするデバイスです。タイマーの設定や開閉の制御は専用のアプリをインストールしたスマートフォンから BLE で接続して行います。

最後に、使用したマシンは次の通りです。

機種	Apple MacBook Pro (14 インチ, 2021)
CPU	Apple M1 Max (ARM64)
OS	macOS Monterey (12.1)

3.2 ソフトウェアの準備

ハードウェアが準備できたところで、ソフトウェアの方も準備します。次のものが必要になります。なお、バージョンは今回使ったものを示しており、必須となる最低のバージョンは確認していません。

- Wireshark Version 3.6.0 (v3.6.0-0-g3a34e44d02c9) [*4]
- Silicon Labs CP210x USB to UART Bridge VCP Drivers v6.0.2 [*5]
- Nordic Semiconductor nRF sniffer for Bluetooth LE v4.1.0 [*6]
- Python 3.9.9 (Homebrew からインストール)

CP210x USB to UART ドライバと Wireshark をインストールします。これらは公式サイトからインストーラーをダウンロードして実行し、その指示に従うだけです。なお、Wireshark については Homebrew Casks にあるので `brew install --cask wireshark` でもインストールできます。

Wireshark をインストールしたら、まず起動して Wireshark → About Wireshark を選択します。About ダイアログが出るので、Folders タブに移動し、“Extcap path” を調べておきます。

[*3] https://ssci.to/3347. 価格は 2021 年 12 月 21 日現在。

[*4] https://www.wireshark.org/

[*5] https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers

[*6] https://www.nordicsemi.com/Products/Development-tools/nRF-Sniffer-for-Bluetooth-LE

BLE の通信を覗いてみよう - S 社編

1 注意

本稿では無線通信の傍受を取り扱いますが、すべて自身が所有・管理するデバイス同士が行う通信に対して、実験として傍受を行っています。本稿の内容を実際に試す際には、必ず自己の管理下において実施するようお願いします。他者の通信を傍受し、その存在を明かすこと、またはその内容を利用することは犯罪となる恐れがあります。

2 はじめに

C99A（2021 年冬コミ）にて頒布した “urandom vol.8”[*1]で BLE の通信を傍受する記事を書きましたが[*2]、このときは通信の傍受と通信内容の簡単な分析まで行いました。また、適当なクライアントを用意して通信を再現し、専用のアプリなしでもデバイスを制御することを目指しましたが、通信が上手くいかず断念しました。

今回はそのリベンジということで、前回と同様にカーテンを操作するデバイスをターゲットとして傍受と通信内容の分析を試みます。

なお、BLE とはそもそもなんぞやとか、この記事のモチベーションなんかは前回の記事を参照頂ければと思います。

3 ハードウェアの準備

3.1 使用機材

スマートフォンなど一般的な BLE 対応デバイスでは、他の BLE デバイス同士の通信を傍受することはできません。今回は Adafruit の “Feather nRF52840 Express” を使用しました。秋月電子通商[*3]やスイッチサイエンス[*4]で入手できます。前回使用した “Bluefruit LE Sniffer” と異なり、これはそのままスニッファーとして使えるデバイスではありません。ファームウェアを書き換えることでスニッファーとして使えるようになります。

傍受の対象となるデバイスは前回同様にカーテンを自動で開閉するデバイスです。ただし、前回とは異なるメーカーの製品を使用しています。機能的には概ね同じで、タイマーの設定や開閉制御は専用のアプリをスマートフォンにインストールして行います。また、Wi-Fi に接続されたハブとなるデバイスがあれば外出先からの制御も可能です。

専用のアプリは中古で入手した Pixel 3 にインストールしました。

最後に、今回使用したマシンの詳細は以下の通りです。

機種	Apple MacBook Pro（14 インチ, 2021）
CPU	Apple M1 Max (ARM64)
OS	macOS Ventura (13.4)

[*1] https://urandom.team/books/urandom-vol8/（電子版は執筆時現在未発刊）

[*2] （追記）本誌に再録してある 1 つ前の記事です。

[*3] https://akizukidenshi.com/catalog/g/gM-16358/ 執筆時 3,080 円

[*4] https://www.switch-science.com/products/5400 執筆時 4,697 円

3.2 Feather nRF52840 Express のブートローダー更新

今回は UF2[*5]によるファームウェア書き込みを行いたいのですが、それにはブートローダーのバージョンが 0.4.0 以降でなくてはなりません。そこでまず入手した Feather nRF52840 Express のブートローダーのバージョンを確認します。ホストマシンと接続した状態でボード上にあるリセットスイッチを素早く 2 回押すと、ホストから ***BOOT という名前のリムーバブルディスクとして認識されます。今回は FTHR840BOOT として認識されていました。この中にある INFO_UF2.TXT にブートローダーのバージョンが記載されています。

```
› cat /Volumes/FTHR840BOOT/INFO_UF2.TXT
UF2 Bootloader 0.2.6 lib/nrfx (v1.1.0-1-g096e770) lib/tinyusb (legacy-525-ga1c59649) s140 6.1.1
Model: Adafruit Feather nRF52840 Express
Board-ID: NRF52-Bluefruit-v0
Bootloader: s140 6.1.1
Date: Dec 21 2018
```

バージョン 0.2.6 とのことなので、まずコマンドラインからブートローダーを更新します。もし 0.4.0 以降のブートローダーが最初から入っていればこの手順は飛ばします。

“Adafruit nRF52 Bootloader” の Release[*6]から最新のものを選び、feather_nrf52840_express_bootloader-*.zip を探してダウンロードします（ * にはバージョン等が入ります)。今回は 0.7.0 をダウンロードしました。

ブートローダーの書き込みに必要なツールである adafruit-nrfutil をインストールします。

```
› pip3 install adafruit-nrfutil
```

筆者の環境では pyenv で環境を切り替えているので特に問題はありませんでしたが、システムの Python を使う場合には --user オプションが必要かもしれません。

macOS では接続されたデバイスが /dev/cu.usbmodem* といった名前で見えているので、ls コマンドで特定します。

```
› ls -la /dev/cu.*
crw-rw-rw- 0,3 root 18 6 13:55 /dev/cu.Bluetooth-Incoming-Port
crw-rw-rw- 0,1 root 18 6 13:55 /dev/cu.COTSUBU_ASMR
crw-rw-rw- 0,7 root  2 8 00:48 /dev/cu.usbmodem3101
crw-rw-rw- 0,5 root  1 8 23:39 /dev/cu.usbmodemC0CC0100F9523
```

（別に接続してないのに COTSUBU for ASMR[*7]が見えているのは謎）

usbmodem を含むデバイスが複数ありますが、作成時刻的に cu.usbmodem3101 だと判断しました。これで必要な情報は揃ったのでブートローダーを書き込みます。

```
› adafruit-nrfutil --verbose dfu serial \
    --package ~/Downloads/feather_nrf52840_express_bootloader-0.7.0_s140_6.1.1.zip \
    -p /dev/cu.usbmodem3101 -b 115200 --singlebank --touch 1200
Upgrading target on /dev/cu.usbmodem3101 with DFU package /Users/mayth/Downloads/
     feather_nrf52840_express_bootloader-0.7.0_s140_6.1.1.zip. Flow control is disabled, Single bank, Touch
     1200

(snip)

Activating new firmware

DFU upgrade took 20.372108221054077s
Device programmed.
```

再度リセットボタンを 2 回押して、INFO_UF2.TXT を見てみます。

```
› cat /Volumes/FTHR840BOOT/INFO_UF2.TXT
UF2 Bootloader 0.7.0 lib/nrfx (v2.0.0) lib/tinyusb (0.12.0-145-g9775e7691) lib/uf2 (remotes/origin/configupdate
     -9-gadbb8c7)
```

[*5] UF2 は Microsoft によって開発された、USB マスストレージクラスを通してマイコンにファームウェアを書き込む仕組み・ファイルフォーマットです。

[*6] https://github.com/adafruit/Adafruit_nRF52_Bootloader/releases/

[*7] https://final-inc.com/products/cotsubu-asmr_ag

とある RISC-V の Voltage Fault Injection (C100 版)

1 はじめに

動作中のプロセッサーに供給されている電源電圧をごく短い間変動させ、プロセッサーの挙動がどのように変化するか観察する “Voltage fault injection”（V-FI, VCC glitching）と呼ばれるテスト手法が存在する。時として命令のスキップやデータの変化のような興味深い挙動が観察されることから、V-FI はコンピューターセキュリティにおける関心の対象とされている。

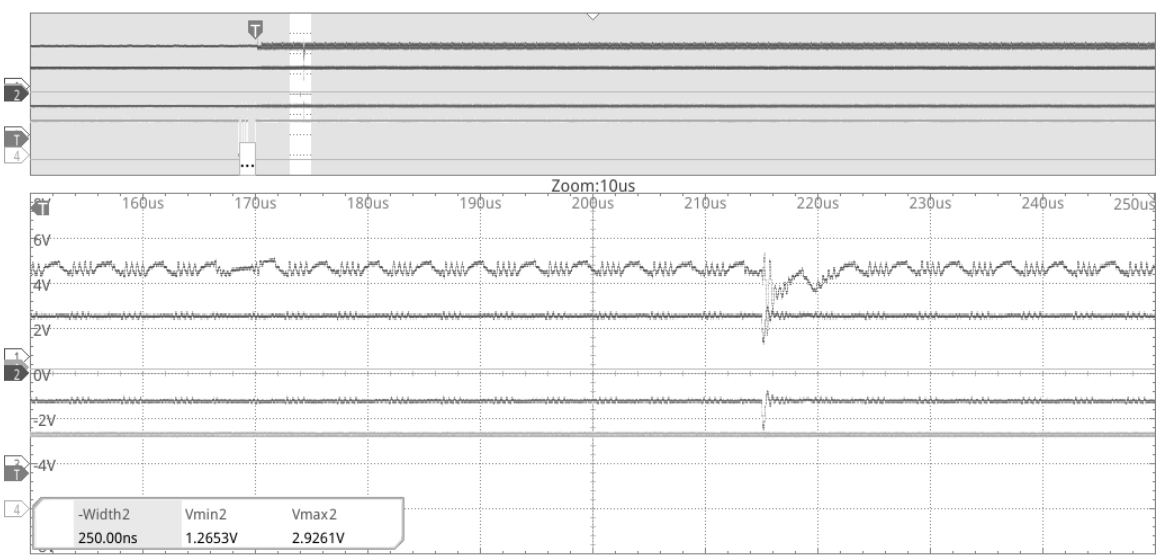

図 1: プロセッサーの電圧を変動させたときのオシロスコープ画像（上 2 本が電流と電圧の測定値）

V-FI は 20 年以上前から研究されているテスト手法であり、V-FI などに対して仕様外挙動が発生しないよう堅牢化されたプロセッサーも一部にあるが、多くのプロセッサーは V-FI に対して何らかの仕様外挙動を示す。最近では、市販の Arm Cortex-M4 プロセッサーで V-FI によりデバッグ機能を有効化できる事例[*1]や、Intel SGX に対する V-FI の研究[*2]などが発表されている。

筆者はソフトウェアセキュリティを専門としているが、ソフトウェアセキュリティの延長線上の課題としてハードウェアセキュリティのテーマも扱っており、いくつかのプロセッサーで V-FI 下の挙動を観察する調査を行っている。この記事では、とある RISC-V プロセッサー上で特定の機械語命令を繰り返し実行している状態で V-FI を行い、命令単位の挙動変化を観察した結果を記す。加えて、同プロセッサー上で動作する簡易な自作アプリケーションに対して

[*1] Informational Notice (IN) IN-133 v1.0 https://infocenter.nordicsemi.com/pdf/in_133_v1.0.pdf

[*2] VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface https://www.usenix.org/system/files/sec21-chen-zitai.pdf

V-FI を行ってアプリケーション機能の挙動変化を観察した事例 1 件とその緩和策についても記す。

この記事は C99 で頒布した同名記事を加筆修正したものである。

2 用語

V-FI
: Voltage Fault Injection の略。

DUT
: Device Under Test の略。テスト対象機器のこと。

glitch
: V-FI で発生させる電圧変動のこと。

3 検証環境

3.1 検証対象

DUT は 32bit RISC-V ISA（RV32I-based）のとあるプロセッサーとした。定格電圧は 3.3V だが V-FI の都合上 3.0V 前後で動作させ、動作周波数は約 100MHz の設定とした。

DUT の動作環境には開発用ボードを用いた。今回の V-FI では電圧が降下する glitch を挿入する。電圧が降下する glitch の V-FI ではノイズ対策のために基板上へ実装されているバイパスコンデンサなどがテストの障壁になるため、予め基板からバイパスコンデンサなどを取り外した。

V-FI の対象として、後述する自作プログラムを DUT に書き込んで動作させた。自作プログラムは OS など無しにそれ単体でプロセッサーを占有して動作するもので、UART を介して入出力を行うよう実装した。

3.2 検証機材

検証機材は次のように接続した。

ファミコンミニと Linux

1 はじめに

身の回りには様々な計算機が溢れていますが、全ての計算機はリバースエンジニアリングの対象たり得ます。2016 年 11 月 10 日に発売されたニンテンドークラシックミニファミリーコンピューター（ファミコンミニ）も例外ではありません。ファミコンミニの中身は U-boot + Linux という組み込み機器にありがちな構成で、ほどほどに未知の部分もあり、簡単な組み込み機器解析の練習題材としてちょうどいいのではないかと思います。カーネル以下のレイヤーが全て GPLv2 ライセンスのソフトウェアで構成されていてソースコードが公開されている点も、練習題材としてのメリットです。

この記事では、組み込み機器解析の一例として、ファミコンミニで自前ビルドの Linux カーネルを起動するまでに筆者（op）が辿った紆余曲折を記します。「部品・完成品ベンダーが提供するドキュメントとか SDK は無いけど、既知の情報と OSS のソースコードを解析してどうにか自前ビルドの Linux を起動する」というものなので、本来の起動手順や用語と比べると正しくない記述が含まれるかもしれませんが、動かした者勝ちという精神でやっていきます。

2 警告

この記事に書かれているファミコンミニの分解等の行為を実践すると、製品保証が無効になったり、ファミコンミニが故障したりする可能性があります。記事内容の実践は、内容を理解できる人が、自己責任で行って下さい。

3 ソースコードの引用とログ出力等の編集について

この記事では、任天堂 Web サイトの OSS ソースコード配布ページ[*1]で配布されている NintendoEntertainmentSystem NESClassicEdition_OSS.zip[*2]からソースコードを引用します。レイアウトの関係上、ソースコードの意味が変わらない範囲で、一部インデントや空白文字を編集して引用しているものがあります。

この記事ではコンソールからのログ出力やソフトウェアの実行結果を引用しますが、長すぎる場合には途中の行を省略しています。省略した行は `.....` で置換しています。ソフトウェアの実行結果中の製品シリアル番号を筆者の意向で隠した部分がありますが、その部分は `<REMOVED>` で置換しています。

4 下調べと事前準備

4.1 ハードウェア構成

まずはハードウェア構成をおさらいします。手元のファミコンミニを分解して出て来た基板の写真（図 1）を示します。

[*1] https://www.nintendo.co.jp/support/oss/

[*2] SHA-1: dbe31c88f2b8b96cb8aa9866e6c59a8ad8fbb628

なお、FEL モードに加えて、FEL モードから更に遷移できる FES モードというモードもあり、こちらは USB を介して Flash へ直接アクセスできる等少し自由度が上がります。最終的には FES モードへの遷移は不要になりましたが、やったことの説明に必要なので、FES モードへの遷移についても書きます。

6.2 U-boot をシェルに落とす

電源投入時のコンソール出力を観察すると、以下のようになっています（リスト 5）。

リスト 5: 通常起動時のコンソール出力

```
[      0.207]

U-Boot 2011.09-rc1 (Aug 30 2016 - 12:07:36) Allwinner Technology

[      0.214]version: 1.1.0
[      0.217]uboot commit : 2f04d11e4dfd9d5022e33833412462859727bdcc

ready
no battery, limit to dc
no key input
dram_para_set start
dram_para_set end
Using default environment

In:    Out:   Err:
Uncompressing Linux... done, booting the kernel.
```

U-boot には各種コマンドを備えたシェルが組み込まれていて、U-boot の設定・ビルドによっては起動シーケンス中に適当なキーを押せばシェルに落ちるのですが、ファミコンミニの U-boot では適当なキーを押していてもシェルに落ちず起動を続行します。なので、ソースコードを調べて穏便にシェルに落とす手が無いかどうか調べます。

とりあえず色々キーを押しながら起動して挙動を観察していると、シェルには落ちないものの、キーを押している場合にのみ `key_press` という文字列が出力されることが分かります。この文字列で U-boot のソースコードを検索すると、sunxi_spl/boot0/libs/common.c の `void set_debugmode_flag(void)` 中に出力処理が見つかります（リスト 6）。

リスト 6: U-boot sunxi_spl/boot0/libs/common.c

```
50 void set_debugmode_flag(void)
51 {
52         char c = 0;
53         int i = 0;
54         for( i = 0 ; i < 3 ; i++)
55         {
56                 __msdelay(10);
57                 if(sunxi_serial_tstc())
58                 {
59                         printf("key_press  \n");
60                         c = sunxi_serial_getc();
61                         printf("0x%x \n",c);
62                         break;
63                 }
64         }
65         if(c  == 's')
66         {
67                 debug_mode = 1;
68                 return ;
69         }
```

これは U-boot の中でも起動初期に走る boot0 の処理ですが、キー入力が s の時に `debug_mode` フラグを立てていることが分かります。適当なキーではなく、s キーを押しながら起動するとどうなるでしょうか？（リスト 7）

リスト 7: s キーを押しながら起動した時のコンソール出力

```
key_press
0x00000073
HELLO! BOOT0 is starting!
```

urandom Selections 1 // 2016-2023

発行者	urandom
表紙デザイン	秋弦めい
発行日	2024 年 08 月 12 日
バージョン	1.00
コミット ID	8abdd5b
連絡先	https://urandom.team/
印刷所	栄光