Česká společnost uživatelů otevřených systémů EurOpen.CZ
Czech Open System Users' Group
www.europen.cz

43. konference

Sborník příspěvků

Penzion Gaudeo – Vranov
29. září–2. října 2013

Programový výbor:

Vašek Matyáš (předseda, Masarykova univerzita v Brně)
Petr Hanáček (Vysoké učení technické v Brně)
Ondřej Krajíček (Y-Soft Corporation, a. s.)
Marek Kumpošt (Masarykova univerzita v Brně, NetSuite Czech Republic, s. r. o.)
Marián Novotný (ESET, spol. s r. o.)
Josef Pojsl (Trusted Network Solutions, a. s.)
Zdeněk Říha (Masarykova univerzita v Brně)
Andrii Stetsko (Y-SoftCorporation, a. s.)
Roman Štěpánek (SODATSW, spol. s r. o.)
Petr Švenda (Masarykova univerzita v Brně)

Sborník příspěvků z 43. konference EurOpen.CZ, 29. září–2. října 2013

Plzeň 2013. První vydání.
Editor: Vladimír Rudolf
Sazba a grafická úprava: Ing. Miloš Brejcha – Vydavatelský servis, Plzeň
e-mail: servis@vydavatelskyservis.cz

Tisk: Typos, tiskařské závody, s. r. o.
Podnikatelská 1160/14, Plzeň

Příspěvky neprošly redakční ani jazykovou úpravou.

ISBN 978-80-86583-26-6

Obsah

Tutoriál: Praktický disassembling malwaru

Václav Lorenc

E-mail: vaclav.lorenc@honeywell.com

V posledních letech se zlepšily možnosti i dostupnost programů pro pořízení obrazu a analýzu RAM, což vedlo ke změně zavedených postupů i usnadnění práce bezpečnostních týmů. Namísto vypínání počítačů se v některých případech pořizuje obraz paměti a provádí se jeho detailní analýza. Tutoriál bude zaměřený právě na artefakty, které se dají nalézt v paměti napadených počítačů, popis výhod, které tato metoda má proti klasické offline analýze či reverse engineeringu malware; to vše na systémech s Windows XP/7/8.

Během praktické části workshopu si účastníci na konkrétních příkladech a obrazech pamětí si vyzkouší praktickou analýzu bezpečnostních incidentů, od pořízení obrazu RAM až po sestavení závěrečného hlášení o nalezených artefaktech či zdrojích infekce. Pro účast na workshopu je třeba mít počítač schopný provozovat virtualizační program VirtualBox.

Virové hrozby na mobilních platformách a jak se jim bránit

Petr Odehnal

E-mail: petr@fpl.cz

Od poloviny devadesátých let začínáme žít v doprovodu chytrých mobilních zařízení a současně s jejich vznikem se objevil i první malware pro ně. Většinou šlo o reálně zcela neškodné proof of concept pokusy, ale to už bohužel neplatí. S rozšířením chytrých telefonů se tyto staly stejně lákavým cílem pro útočníky, jako klasické počítače. První část bude proto věnována vývoji mobilních platforem a jejich (ne)bezpečnosti.

Ve druhé části přijde možná i Mazaný Filip a položí otázku si „Co když zavirování telefonu není nejhorší věc, která se vašemu mobilnímu společníku může stát?“ Věnovat se bude praktickým zkušenostem s platformou Android, kde v posledních měsících dochází ke znatelnému nárůstu počtu různých útoků a ty jsou stále sofistikovanější. Řeč bude o téměř neodstranitelném trojském koni i o chybách systému, které dovolily útočníkovi přidat jeho kód k jakékoli legitimní aplikaci bez nutnosti měnit digitální podpis.

Exploiting missing permission in Android

Juraj Varga, Martin Gazdík

E-mail: juraj.varga@stuba.sk, gazdik.martin.87@gmail.com

Introduction

Operating system Android is currently the most widespread operating system (OS) for mobile phones, tablets and other mobile devices [1]. The first version of this OS was presented to the public in 2007. Since then the number of users (and sold devices) is growing and mobile devices with Android are used in everyday life. With this rise also grows the number of applications users can download. There are also many malicious attempts on compromising security of OS or applications.

Android security architecture

Android is operating system built on Linux architecture. Due to the fact that mobile devices have limited hardware resources, this architecture had to be modified to fit this situation. From the beginning, OS Android was developed as an open mobile platform. It enables applications to use built-in hardware and software along with both local and remote data to grant users desired comfort and functionality. Along with all this the OS must provide means to secure user data, applications and whole device. Android provides various security mechanisms that can be illustrated on the fig. 1.

As was mentioned above, Android is based on Linux kernel. It provides Android with several key security mechanisms:

- **User-based permission model** — Each application contains list of permissions which are needed for its functionality and user can accept or decline them.
- **Process isolation** — Separation of application and assignment of resources and data.

The support of the grant VEGA 1/0173/13 is kindly announced.

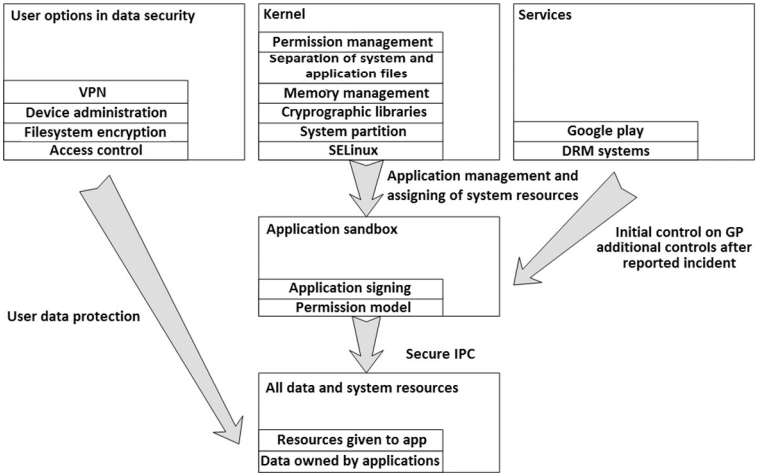


Fig. 1: Security architecture of OS Android

- **Mechanisms for secure IPC** — Mechanisms used for secure communication between applications (between processes).
- **Possibility to remove unused or potentially dangerous parts of kernel** – Linux kernel is modular, apart from basic parts all others are separable and containing components will not be used when separated.

Application sandbox

Each application in Android is given unique user identification UID and each application is run as an independent user in a separate process. Each user is assigned specific system resources and space in file system. This way can be rather simply guaranteed separation of system resources and files of other users, in this case other applications. This approach is different from other operating systems, where multiple applications can run with the same user privileges.

This principle serves as a basis of application sandbox on kernel level. Kernel ensures security between applications and system on process level using standard Linux capabilities, e.g. assigning UIDs and G(roup)IDs to applications. Applications in default settings cannot communicate between each other and have limited access to OS. Since each application has its data and resources separated from other applications, every attack is limited to infected application only. Essentially all software, including OS libraries, application framework and runtime runs in sandbox. There cannot be cases of damaging whole operating

memory because each application has access only to restricted part of it. Although application sandbox is not perfect, to crack it on properly configured device attacker needs to crack security of Linux kernel [3].

Permission model

Since our attack scenario exploits a flaw in permission model, we will inspect this security measure more closely. Operating system Android allows running of applications which support services for users of mobile devices. However, their structure must be from security point of view based on certain common design standards.

Permission model — access to restricted APIs

Applications run in application sandbox and have access to limited system resources. System manages application access to resources and if it is misused or used incorrectly, can severely affect functionality of device or compromise data. These restrictions are implemented by various means. Some possibilities are restricted by lack of corresponding APIs, others by e.g. role separation. Sensitive APIs are used by only trusted applications and protected by permission system. Permission system is platform specific for OS Android, so attacks based on exploiting flaws in permission system are not applicable on other platforms such as Apple iOS or Windows Mobile. Permissions for each application are specified in AndroidManifest.xml — control file where are specified required permissions to access system resources on device — telephony, location services, access to contact list etc [2]. Permissions are divided into four groups based on level of protection (for illustration see fig. 2):

- Normal — permissions of application level, they do not pose a serious risk when they are used by application.
- Dangerous — dangerous permissions which can cause leakage and manipulation with sensitive data or exploit potentially dangerous system resources. They need to be explicitly confirmed by user during installation of application. Here belong for example:
  - Location data from GPS (ACCESS_FINE_LOCATION)
  - Network/data connection (ACCESS_NETWORK_STATE)
  - Telephony (CALL_PHONE)
  - SMS/MMS functions (WRITE_SMS)
  - Access to system configuration (CHANGE_CONFIGURATION)

- Signature — permissions assignable only to applications signed with private key corresponding with certificate of application calling it. They are used by developers to share information among their applications.
- Signature-or-system — special type of permission assignable only to applications installed in system image which are signed with the same certificate as system image.

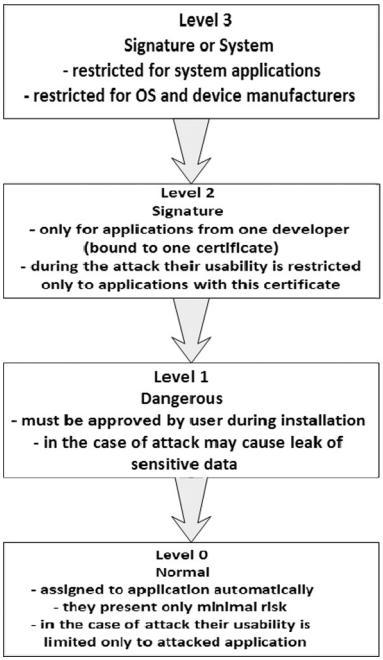


Fig. 2: Android permissions

Aside from above mentioned division, permissions are also divided by the way of accepting them:

- Time-of-use — user must confirm this permission when executing sensitive operation (e.g. access to device location). It is the only way to prevent applications to access device resources.
- Install-time — accepted when installing application, user accepts them as one — he cannot choose which permissions to accept and which to deny.

System resources marked as “dangerous” are accessible only from the OS. Applications must have these requirements specified by permissions in manifest.

During installation these permissions are displayed to user and he can accept or reject them. After accepting, installation continues and these permissions are accepted by system. It is not possible to choose which permissions to accept, they must be accepted as one and that can lead to security incidents (privilege escalation attacks [8]). Permissions are assigned to application for the time of its installation on device and are not additionally required from user. They are removed in the moment the application is deleted from device. They can be revised in application settings and can be restricted by shutting down global functionality, such as Wi-Fi or GPS. In case the application is trying to access feature that is not allowed to, it invokes a security exception and error message in application. Security checks for protected API permissions are done on the lowest possible level to prevent their circumventing. Some device capabilities are not accessible to third-party applications but can be used by pre-installed applications. The complete list of permissions is available on website dedicated to Android development [5].

Cost-Sensitive API

Cost-Sensitive API is a function whose running can cost user something (mostly money).These APIs are also in the list of protected APIs controlled by OS. User must give explicit permission to third-party applications to access them. Here belong for example:

- Telephony functions
- SMS/MMS
- Data and network access
- Manipulation with financial data in application
- Access to NFC (Near Field Communication) technology

Evolution of permission model

As the OS Android evolved, so did the permission system. Each version of OS is represented by API level. These API levels contain (except other things) permissions applicable to this version of OS. Wei et. al. in their paper [10] presented the evolution of permission system. First released API (level 3) contained 103 permissions. Since then, a lot of permissions were added, while some were removed or changed. Currently there are over 160 of them. Majority of these new permissions fall to the dangerous category and were introduced mostly due to the new hardware possibilities of devices such as NFC (Near Field Communication) technology. Since version 4.2 Android provides another security check for SMS — notifies user if some application tries to send SMS to so called premium number (mostly paid services). Sending this kind of message is then in

user's competence. The major problem of this model is that it lacks sufficient permission granularity to secure various combinations of installed applications. Moreover, description of permissions is far too complex for common users [4]. This leads to skipping reading this part during installation, which can lead to installation of potentially dangerous applications.

Proposed attack scenario

Let us suppose that some application has a local database. If this database is stored in internal memory, it is not accessible for other applications due to application sandboxing. Now suppose that this database is large, and the application stores it on the storage media (memory card). Since there is no permission allowing access to files on storage media, any application can access or modify it, which is a clear violation of application separation policy.

To demonstrate the danger of missing storage card permission, we decided to create an application that can access user data on device and send them over the internet to the attacker. Of course, true functionality of this application must remain hidden, so it has to provide legitimate functionality to avoid being detected.

Customized e-mail client

First of all, we needed a way to transport stolen data to our computer. The easiest way was to create a customized e-mail client using Java Mail Library [11]. This custom client has a great advantage, because if fully manageable, with possibility to connect to arbitrary SMTP server. It is possible to configure own SMTP server and connect application to this server. In this work we used SMTP server by Google. The only limitation of this system is data limit for attachments, which is set to 25 MB. However, this capacity is fully sufficient for obtaining various sensitive files located on device. Application that uses this client needs permission to access the internet to work properly and that can be reason for rejecting installation by user. That is why we decided to create two applications — File manager and Communicator. Both applications will require permission to use internet that is obviously not necessary for File manager, but can be requested due to the commercials.

```
<uses-permission Android:name="Android.permission.INTERNET"
    />
```

Fig. 3: Cut-out from manifest file requesting permission to access internet

File manager application

When we had means of sending stolen data, we chose to create a file manager application, which can browse the memory card in infected device. Its functionality is very simple — right after start it shows list of files on screen. User can scroll up and down in the application. After clicking on chosen directory it opens. User can browse directory structure until he finds desired file. When he clicks on this file, application offers a list of installed applications that can correctly open this file.

Since there is not required a permission to access these data, we can implement a malicious functionality (our e-mail client) to this application and thus steal user data. After finding desired file and clicking on it, our mail client is called and sends this file to our SMTP server. This whole action happens in background, without any confirmation — file is normally open and user can not know that he has been just subjected to this attack. The only suspicious thing about this application is required internet permission, but as we mentioned before, it can be used in various advertisement libraries to update commercial banners in application.

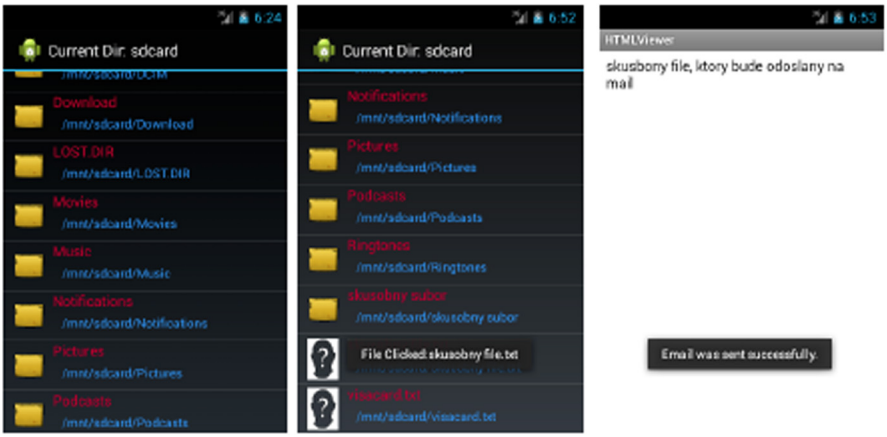


Fig. 4: File manager application — a) running b) after clicking on file c) after sending a file

Messenger application

Since we somehow need to mask the necessity of internet Access, we decided to create another application — a messenger. This application requires internet permission for its correct behavior, so the user has no suspicion of malicious intents. The application consists of five means of communication:

- GTalk
- Facebook chat
- Jabber client
- SMS
- e-mail communication

and embedded file manager. Working with SMS messages requires three additional permissions, but even that should not raise any suspicion, because they are needed for proper work.

Whole application is based on possibility to send e-mails without user's notice. In the first four mentioned parts are all sent and received data written in the database in this form:

- Id
- Receiver
- Message

When the table is filled with given number of items (currently set to 20 items for testing purposes), it is sent to embedded attacker's e-mail. Database capacity is being checked by service running in background. Database has an easy access; it is always stored in /data/data/package.aplication/database directory. It is possible to access this database and send it as an e-mail attachment. Part dealing with e-mails is programmed to send every sent or received mail to our SMTP server. Of course, user can also use integrated file manager to send attachments in his e-mails. This way we can simply obtain not only whole user's communication — but also private contacts, phone numbers and files from memory card — and misuse these data for our malicious purposes. We can also create a backdoor in legitimate application in similar way. The application can for example store some data in a database on memory card data. Other application from the same developer (a game) can access this memory card and send all accessible data to "game server" in order to improve "user experience".

Detection methods

Detection with commercially available applications

After successful creation of these two applications, we decided to test them by seven most-used (most downloaded) antivirus applications found on Google Play Store:

- ESET Mobile Security by ESET
- avast! Mobile Security by Software

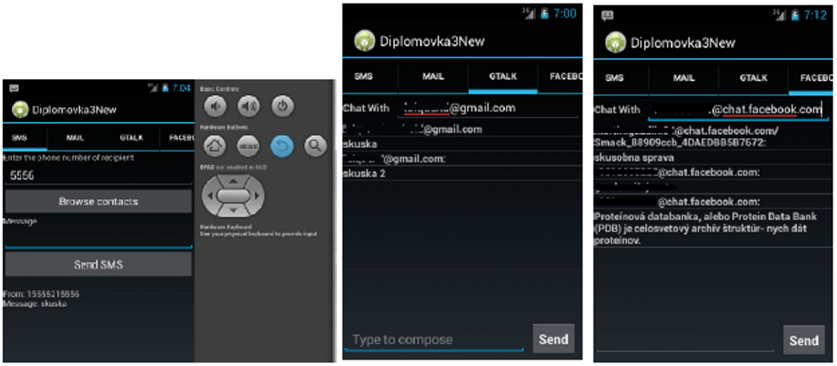


Fig. 5: Messenger application — a) SMS messaging, b) GTalk, c) Facebook chat

- Antivirus Security — FREE by Mobile Technologies
- McAfee Antivirus & Security by McAfee Mobile Security,
- Lookout Security & Antivirus by Lookout Mobile Security
- Mobile Security & Antivirus by Trend Micro
- Norton Security antivirus by NortonMobile

During each test our applications were at first left without interaction, and a deep scan was performed. After that, the active attack was conducted several times in a span of minutes. None of the commercially available applications detected our attempts for attack. During all scans our applications were marked as trusted or the scan finished without warning. Tests were performed on modified CyanogenMod ROM [12] because of the possibility of having a "rooted" device (with administration privileges). This way we wanted to permit applications to scan root parts of the system for the possible threats.

Our approach

After unsuccessful tests of commercial applications we decided to create our own application, which can inform user that something suspicious is happening in tested application. We chose to let user decide whether to flag this application as potentially dangerous and remove it, or leave it alone and trust it. Our design does not require any permission and can be installed from API level 8.

Over past two years there were many approaches on how to detect malicious applications such as [6, 7, 9]. We decided to use a background service which can get upload and download traffic of the application (based on its UID) by embedded functions. These values are stored in *shared preferences* class for

further access. We aim to show user only the applications that use internet so we filter them using Package manager (it scans manifest file and filters required permissions). Information about uploaded/downloaded data will be shown on graph — upload in yellow and download in red. The main idea of detection is very simple: data collecting (or stealing) application should have far higher upload than download. If we manage to find out that application is sending large volumes of data, we have a possibility to remove such application from device. Still, we have to be cautious and use our reason, because even if application does not send a lot of data, it does not mean it is a safe application.

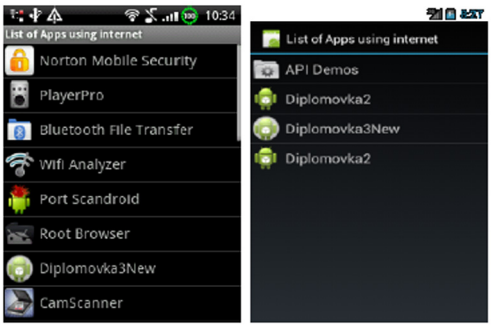


Fig. 6: List of applications with internet usage — a) real device, b) emulator

Results

At first we tested our solution on File manager. This application should not have any upload and minimal download (for commercials). On fig. 7 we can see that upload peaked from zero to 74000 bytes. At this moment we clicked on text file of this size. From this graph we can say that something suspicious happened in application background. Observed application was uninstalled with appropriate button.

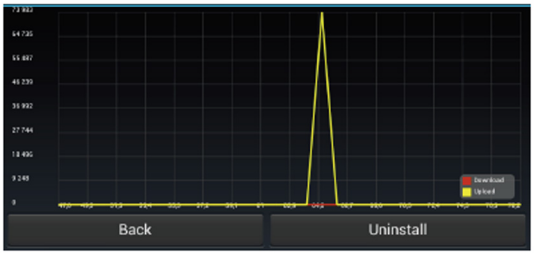


Fig. 7: Observation on File manager

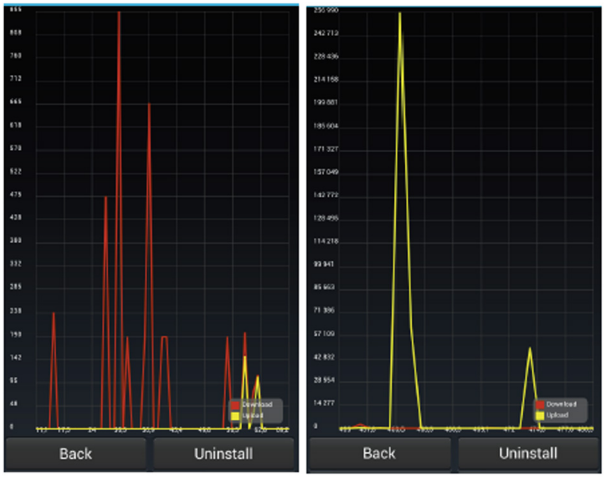


Fig. 8: Observations on Communicator — a) instant messaging, b) sending e-mails

After the first test concluded, we moved on to testing the second application. Here we expected complications in determining whether application sends some data in background. When the database parameters are correctly set — number of items to be send — we could find some variations in communication. As you can see in fig. 8a, during instant messaging with database size of 20 stored messages (database size is 16 kB) the upload does not stand out in the graph. But in the real-life situations, when the size is set to e.g. 1 000 items (so the mail server is not overloaded), size of database to be send would be significantly higher (depending on size of messages). We can assume that database can also be sent periodically (once a week or month), so it will be even easier to detect such behavior.

Finally we tested detection of sending e-mails. Here the amount of data sent was much higher than the size of attachment (as illustrated on fig. 8b). From this observation we can say that apart from our e-mail something else is being sent in the background.

With this method we were fairly successful in detecting that our malicious applications are performing some unwanted in the background. We can be 100 % sure with our result in the case of File manager. However, because of almost even upload and download ration in the Communicator we cannot say this in the second case. Even that there are some clues that the application is sending additional data in background.

Conclusions

We demonstrated how a missing permission in the area of protecting user data compromises storage media on device. Moreover, this type of attack is very

simple and can be performed even by a user with only basic knowledge of programming on OS Android.

The results of this work are three applications — two can successfully obtain a large amount of user data without his knowledge and transfer them to the attacker. The last application can rather effectively detect this attack with some help from the user. The next step in this field of research can be optimization of detecting application so it does not need an input from user to decide whether the application is safe or not. This approach can also be included in current antivirus applications to further increase their chance of finding malicious applications like the two we created.

References

[1] http://9to5mac.com/2013/05/01/idc-ipad-drops-below-android-with-40-share-of-worldwide-tablet-market-apple-still-top-vendor/

[2] http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels

[3] https://source.android.com/tech/security/

[4] FELT, A. P. et. al.: Android Permissions: User Attention, Comprehension, and Behavior, In Symposium on Usable Privacy and Security (SOUPS), 2012.

[5] http://developer.android.com/reference/android/Manifest.permission.html

[6] Zhou, Y., Jiang, X.: Dissecting Android Malware: Characterization and Evolution. In IEEE S & P, 2012.

[7] Zhou, Y., Wang, Z., Zhou, Wu, Jiang, X.: Hey, You, Get off of My Market: Detecting Malicious Apps in Offcial and Alternative Android Markets . In NDSS, 2012.

[8] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A., Shastry, B.: Towards Taming Privilege-Escalation Attacks on Android . In NDSS, 2012.

[9] Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A Study of Android Application Security. In USENIX Security Symposium, 2011.

[10] Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M.: Permission Evolution in the Android Ecosystem, In ACSAC '12 Dec. 3–7, 2012.

[11] http://code.google.com/p/javamail-android/

[12] http://www.cyanogenmod.org/

Techniky získavania citlivých údajov z Apple iOS zariadení

Eugen Antal, František Baranec

E-mail: eugen.antal@stuba.sk, baranecfrantisek@gmail.com

Abstrakt

Dnešné mobilné zariadenia, či už tablety alebo smartfóny fungujú na moderných mobilných operačných systémoch. Počet takýchto zariadení neustále narastá a stávajú sa pre ľudí súčasťou ich každodenného života. S narastajúcim počtom narastá aj ich využiteľnosť a tým pádom aj ukladanie a spracovanie citlivých údajov. Apple iOS zariadenia patria medzi najvýznamnejších predstaviteľov moderných mobilných zariadení. Veľké množstvo citlivých údajov indikuje nutnosť ochrany týchto údajov. Firmy vyvíjajúce mobilné platformy kladú čoraz väčší dôraz na bezpečnosť a ochranu údajov. Operačný systém iOS sa prezentuje ako jedna z najbezpečnejších mobilných platforiem.

Táto práca je venovaná forenznej analýze (resp. získavaniu citlivých informácií) na mobilných zariadeniach od spoločnosti Apple Inc. bežiacich na mobilnom operačnom systéme iOS. V prvom rade skúmame možnosti obídenia všetkých bezpečnostných prvkov platformy pomocou tzv. Jailbreak-u. Popisujeme výhody, nevýhody a typy zraniteľností, ktoré Jailbreak používa. Ďalej sa venujeme rôznym možnostiam získavania informácií buď priamo zo zariadenia alebo zo zálohy. Popisujeme okolnosti a podmienky na hardvérové obídenie hesla iOS zariadení, získavanie AES hardvérových kľúčov či obnovu vymazaných údajov. Na záver sa venujeme analýze niekoľkých vybratých bankových aplikácií slovenských bánk. Zameriavame sa na množstvo ukladaných informácií jednotlivých aplikácií v zariadení a či je možné sa dostať k citlivým údajom.

1 Úvod

Keďže v poslednej dobe rapídne narastá počet smartfónov, je zrejmé, že si uchovávajú veľké množstvo citlivých údajov svojich majiteľov. Táto práca sa venuje akvizícii údajov, čiže forenznej analýze, zo zariadení firmy Apple Inc. ako sú iPhone, iPad a iPod touch s mobilným operačným systémom iOS. Apple sa snaží svoje zariadenia spraviť čo najbezpečnejšie, aby ochránili údaje svojich užívateľov ako v súkromnom, tak aj firemnom sektore.

2 iPhone, iPad, iPod Touch a iOS

V roku 2007 spoločnosť Apple Inc. prvýkrát predstavila revolučný smartphone iPhone ovládaný pomocou dotykového displeja s mobilným operačným systémom iPhone OS (dnes nazývaným iOS). iOS je mobilný operačný systém odvodený od OS X, založený na rovnakom MACH jadre a zdieľa niektoré elementy s OS X 10.5 (Leopard). iOS je založený na 4 vrstvách, konkrétne ide o Core OS, Core Services API, Media layer a Cocoa Tough layer [3]. Od roku 2007 Apple postupne predstavil ďalšie produkty, ktoré využívajú iOS ako iPad, iPod Touch a Apple TV. Aktuálne verzie zariadení sú iPhone 5, iPad 4, iPod Touch 5 a najnovšia verzia iOS je 6.1.3.

3 Jailbreak

iOS je uzavretá platforma, čo znamená, že ak by chcel niekto publikovať aplikáciu do Apple Storu, tak by musela spĺňať prísne Apple smernice. Všetky obmedzenia sa obhajujú najmä bezpečnosťou systému a kvalitou aplikácií v Apple store. Existujú však aj aplikácie, ktoré by nijako nenarušovali bezpečnosť a čo viac, vylepšili by iOS tak, že by to užívateľovi uľahčilo prácu. Z toho dôvodu vznikol Jailbreak.

Jailbreak je proces, ktorý obchádza bezpečnostné mechanizmy a obmedzenia definované Applom na iOS zariadeniach. Predchádza ho nájdenie softvérových alebo harvérových zraniteľností pomocou, ktorých je realizovateľné inštalovanie modifikovaných kernelových záplat. Po vykonaní tohoto procesu je možné na zariadeniach spúšťať nepodpísaný kód. Jailbreak zabezpečuje aj kompletný „root“ prístup, ktorý inak nie je prístupný. Pojem „root“ pochádza z UNIX systémov, kde root je superpoužívatel s neobmedzenými právami a prístupom ku všetkým súborom. [1]

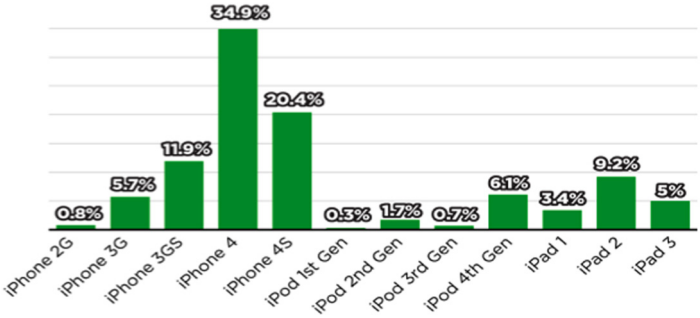
Jailbreaknuté zariadenie umožňuje svojmu majiteľovi upravovať a inštalovať ľubovoľné aplikácie. Napríklad vylepšenia užívateľského rozhrania, alebo neoficiálne aplikácie, ktoré porušujú Apple smernice a nikdy by sa nedostali do Apple Store. Na distribúciu takýchto aplikácií existuje alternatívny obchod s názvom Cydia. Vytvoril ho heker známy pod prezývkou Saurik. Cydia sa distribuuje pomocou jailbreakovacích nástrojov, ktoré ju väčšinou nainštalujú pri jailbreakovaní zariadenia. [1]

V niektorých prípadoch jailbreak umožňuje aj takzvaný „unlock“, ktorý odstraňuje SIM obmedzenie. SIM obmedzenie je obmedzenie zariadenie len na jedného operátora.

S jailbreakom sú spojené aj riziká. Jedným z nich je root prístup. Ten je síce potrebný na vykonávanie úprav a inštalácií, ale zároveň otvára aj zadné dvierka pre škodlivý softvér. Ďalšou nevýhodou je v niektorých prípadoch nestabilita,

narastajúci objem stiahnutých dát alebo rýchle vybíjanie batérií, ktoré môže nastať po nainštalovaní aplikácie, ktorá nespĺňa Apple smernice. Pojem jailbreaking vznikol keď, hekeri prvýkrát popisovali tento proces a odkazovali sa na vyňatie iOS zariadenia z iTunes väzenia (ang. jail, break).

Podľa Cydie je celkovo jailbreaknutých asi 10 % zariadení [7]. Percentuálne vyjadrenie jailbreaku na jednotlivých iOS zariadeniach je na obr. 1.



Obr. 1: Počet jailbreaknutých zariadení [7]

3.1 Typy jailbreaku

Jailbreak existuje už niekoľko rokov a je kompatibilný skoro so všetkými verziami iOS, nie všetky jailbreaky majú rovnaké vlastnosti a funkcie. Jedným z hlavných faktorov, ktorý ovplyvňuje jailbreak a určuje aj to, čo jailbreak dokáze, je bezpečnostná zraniteľnosť, ktorú využíva. Táto zraniteľnosť sa používa na zlomenie obmedzení definovaných v iOS.

Akonáhle sa táto zraniteľnosť použije pri jailbreaku, tak sa o nej dozvie aj Apple, a to sa snaží ju zaplátať pri nasledujúcej iOS verzii. Z toho vyplýva, že skoro pre každú novú verziu iOS je potrebné nájsť novú zraniteľnosť. Zraniteľnosť sa môže objaviť aj v hardvéri. Takáto sa považuje za silnejšiu, nakoľko na jej odstránenie nestačí len softvérová aktualizácia, ale vyžaduje si vydanie nového iOS zariadenia. Jednu z najznámejších hardvérových raniteľností využívala limera1r [8]. Išlo o zraniteľnosť na procesoroch A4.

Využitá zraniteľnosť ovlplyvňuje jailbreak aj z pohľadu jeho stálosti. Stálosťou sa myslí, že úpravy vykonané jailbreakom ostanú na zariadení aj po reštartovaní zariadenia. Takto vzniklo rozdelenie, ktoré definuje jailbreak, buď ako tethered (pripojený) alebo untethered (nepripojený) [1].

Tethered jailbreak

Tento typ zmizne akonáhle sa zariadenie reštartuje a na obnovenie jailbreaku si vyžaduje akúsi formu re-jailbreaku. Obvykle to znamená nutnosť pripojenia

zariadenia cez USB kábel ku počítaču. Z tohoto dôvodu vznikol aj jeho názov tethered. Tento pojem sa používa aj na jailbreaky, ktoré nevyžadujú pripojenie pomocou USB kábla k počítaču, ale postačuje navštívenie webovej stránky, alebo spustenie konkrétnej aplikácie.

Ak je zraniteľnosť nájdená v nejakom privilegovanom kóde, tak na tethered jailbreak stačí samotná. Príklad takejto zraniteľnosti je limera1n bootrom exploit, ktorý sa používa vo väčšine jailbreakov na iOS 4 a 5. Ďalšou takou je zraniteľnosť v jadre USB ovládača. V dnešnej dobe však nie sú známe žiadne ďalšie takéto silné zraniteľnosti a na nových zariadeniach boli aj tieto Applom odstránené.

V prípade, ak nemáme znalosť žiadnej silnej zraniteľnosti, tak je možné napadnúť zariadenie cez aplikácie s nižšími právami, ako napríklad Safari. V tomto prípade potrebujem mať aj ďalšiu zraniteľnosť v jadre iOS.

Takže výsledkom pri tethered jailbreaku je to, že na jailbreak potrebujeme buď jednu zraniteľnosť nájdenú v privilégovanom kóde, alebo dve zraniteľnosti, kde jedna z nich nám umožní prístup do zariadenia a druhá zraniteľnoť slúži na obídenie bezpečnostných prvkov iOS.

Untethered jailbreak

Tento typ ostáva na zariadení nastálo a nie je ovplyvnený reštarovaním zariadenia. Považuje sa teda za lepšiu formu jailbreaku.

Untethered jailbreak je zložitejší ako tethered. Vyžaduje si zraniteľnosť na veľmi špecifických miestach v bootchaine. V minulosti ho bolo možné vykonať pomocou hardvérovej zraniteľnosti, ktorá bola nájdená na začiatku bootchain procesu. Dnes je však už odstránená. Preto sa v dnešnej dobe na untethered jailbreak používa kombinácia tethered jailbreaku a následného exploitu, ktorý zabezpečí jeho stálosť.

3.2 Typy zraniteľností

Lokalizácia zraniteľnosti má dopad na prístupový level do zariadenia. Niektoré zraniteľnosti povoľujú nízkoúrovňový hardvérový prístup, ostané obmedzené oprávnenia vnútri sandboxe. Existujú nasledovné typy zraniteľností [1]:

- Bootrom Level
- iBoot Level
- Userland Level

Bootrom Level

Z pohľadu jailbreaku je takýto typ zraniteľnosti najsilnejší. Bootrom je hardvérová súčasť zariadenia a ako sme si už spomínali takáto zraniteľnosť nemôže byť

odstránená len softvérovou aktualizáciou. Je nutné vydať nové zariadenia. Napríklad pri limera1n bola táto zraniteľnosť odstánená až príchodom nových A5 procesorov spolu s iPhonom 4S a iPad 2.

Sila týchto zraniteľností nepozostáva len v tom že je nutné na jej opravu vydať nový upravený hardvér, ale aj to, že umožňujú nahradiť alebo upraviť hociktorú časť bootchain, vrátanie bootovacích argumentov jadra. Keďže sa nachádza na začiatku bootchain, má aj priamy prístup k hardvéru, čo umožňuje prístup k AES hardvérovým kľúčom.

iBoot Level

Táto zraniteľnosť je skoro tak silná ako zraniteľnosť na úrovni bootromu. Jej hlavným rozdielom je to, že sa dá zaplátať softvérovou aktualizáciou. Je však tiež skoro na zažiatku bootchainu, a preto umožňuje napríklad aj čítanie hardvérových AES kľúčov.

Userland Level

Ide o zraniteľnosti, na úrovni iOS prostredia. V tomto prípade treba na jailbreak dve zraniteľnosti. V porovnaní s predchádzajúcimi dvoma typmi ide o slabšie zraniteľnosti, nakoľko pomocou nich nie je možné získať prístup k AES kľúčom. Naviac, je najľahšie ich odstániť.

4 Forenzná analýza

Forenzná analýza je definovaná ako súhrn techník a nástrojov určených na hľadanie dôkazov na elektronických zariadeniach. Jednou z jej hlavných podmienok, ak nie najhlavnejšou je, že počas jej vykonávania nesmie dôjsť k zmene zdrojových informácií. V prípade, že nie je možné vykonať analýzu bez modifikácií, tak musia byť presne zdokumentované spolu s udaním dôvodu ich vykonania [3].

V prípade iOS zariadení je možné forenznú analýzu vykonať pomocou rôznych techník z nasledovných zdrojov:

a) fyzické zariadenie

b) záloha (iTunes, iCloud)

4.1 Forenzná analýza zo zálohy

Jedným z dvoch zdrojov údajov pre forenznú analýzu na iOS zariadení je záloha vytvorená cez iTunes alebo na iCloude. V našej práci sa venujeme tej, ktorá bola vytvorená cez iTunes.

Ide o logickú kópiu iOS súborového systému. Tento spôsob forenznej analýzy číta súbory zo zálohy vytvorenej pomocou AFC (Apple file connection) protokolu, uloženej na osobnom počítači, ku ktorému sme získali prístup. Nevýhodou tejto formy analýzy je, že získame len údaje, ktoré sú explicitne zálohované pomocou AFC.

AFC zálohuje kontakty, SMS, kalendáre, fotky, nastavenia siete, históriu hovorov, konfiguračné súbory, databázové súbory, Keychain (bezpečné úložisko hesiel), históriu, cookies a záložky zo Safari, dáta aplikácií a mnohé ďalšie užívateľské dáta. Nezálohuje obsah synchronizovaný so zariadením cez iTunes. Ide o videá, skladby, podcasty a aplikácie samotné. Záloha navyše obsahuje detaily zariadenia ako sériové číslo, UDID (Unique device ID – 40 hexadecimálnych znakov), telefónne číslo a sériové číslo SIM hardwaru.

Záloha je uložená na disku v zložke, ktorá je vytvorená pri prvom synchronizovaní a jej názov tvorí UDID zariadenia [2]. Proces zálohovania je v iTunes prednastavený tak, že sa zariadenie automaticky zálohuje pri každom pripojení. Táto možnosť sa dá vypnúť, pričom je tu ešte možnosť robiť zálohy manuálne. Zálohovať je možné cez USB alebo Wi-Fi. Pri každej ďalšej zálohe dochádza len k updatovaniu aktuálnej zálohy a zmene časových odtlačkov. V prípade updatovania alebo obnovovania zariadenia, iTunes vytvorí vždy automatickú zálohu, ktorú uloží pod názvom [UDID]+‘–’+[časový odtlačok]. Umiestnenie záloh sa líši v závislosti od použitia operačného systému.

Tab. 1: Umiestnenie záloh

OS	Umiestnenie zálohy
Windows XP	C:\Documents and Settings\[user name]\Application Data\ Apple Computer\MobileSync\Backup\
Windows 7	C:\Users\[user name]\AppData\Roaming\ Apple Computer\MobileSync\Backup\
MAC OS X	∼/Library/Application Support/MobileSync/Backup/

Záloha samotná obsahuje súbory, ktoré nie sú v čitateľnom formáte. Názvy súborov pozostávajú zo 40 alfanumerických hexa znakov bez prípony. Napríklad f968421bd39a938ba456ef7aa096f8627662b74a. Ide o SHA1 hash zložený z príslušného doménového mena a cesty k súboru spojených pomocou ‘–’. iTunes číta a ukladá doménové mená a názvy ciest z meta súborov. Každá záloha spolu s dátami obsahuje aj 4 meta súbory. Ide o Manifest.plist, Status.plist, Info.plist a Manifest.mbdb. Väčšina údajov v zálohe je uložených ako .plist, sqlite databáza alebo obr. [2]. Zobraziť ich je možné jednoduchým pridaním prislúchajúcej koncovky. To, o aký typ súboru ide zistíme z Manifest.mbdb, v ktorom sa nachádza súborová štruktúra a názvy súborov (nie ako hash, tie si musíme vypočítať a poprirad'ovať).

Existujú rôzne nástroje, ktoré sú schopné prečítať iTunes zálohy ako napríklad [6], ktorý sme pri našej práci používali. Dokážu prečítať súborovú štruktúru a preložiť nečitateľné názvy súborov. Niektoré používajú dokonca Apple mobile device API s iTunes. Záloha môže byť šifrovaná alebo nešifrovaná.

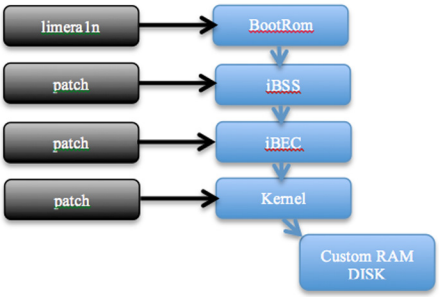
V prípade nešifrovanej, sa pomocou [6] vieme dostať ku všetkým údajom zálohy, ktoré sme spomínali, ako aj dátam IM (instant messaging) aplikácií (napríklad: Whatsapp, Skype, ...). Dáta ktoré sú chránené pomocou data protection sú zašifrované a nie sú čitateľné. Kľúče k nim sú uložené v Backup Keybag (Keybag – slúži na uchovávanie šifrovacích kľúčov). Pri normálnej nešifrovanej zálohe je tento Backup keybag šifrovaný pomocou tzv. „0×835 key“, ktorý je odvodený od hardvérového AES kľúča zariadenia. Z toho vyplýva, že aby sme kompletne dešifrovali zálohu potrebujeme fyzický prístup k zariadeniu a z neho získať 0×835 key. Toto sa dá spraviť dvoma spôsobmi. Buď prelomením ochrany zariadenia - Jailbreak (postup: nainštalujeme si na ňom SSH, patchneme kernel, a spustíme skript na získanie kľúča), alebo druhým spôsobom, tým je načítanie vlastného, upraveného OS s nástrojmi na jeho získanie.

Po získaní tohto kľúča existuje balík Python skriptov [5], ktoré slúžia na dešifrovanie súborov chránených pomocou data protection. Celá akvizícia dát je značne obmedzená až znemožnená ak je záloha šifrovaná. Zálohu je možné šifrovať pomocou hesla zadaného v iTunes. V tomto prípade je celá záloha šifrovaná pomocou tohto hesla a na to, aby sme sa dostali k nejakým údajom, musíme toto heslo poznať. Nie sú naň kladené žiadne bezpečnostné požiadavky. V prípadoch keď sa nejedná o silné heslo, je možné použiť brute-force metódu, či slovníkový útok na zlomenie hesla.

V prípade, ak by sme mali fyzický prístup k zariadeniu, je možné sa k tomuto heslu dostať. Zariadenie si ho ukladá vo svojom Keychaine. Čiže máme opäť dve možnosti, buď zariadenie jailbreaknuť alebo si nahrať vlastný OS s nástrojmi na získanie obsahu keychainu. Po získaní kľúča opäť použijeme Python skripty [5] na dešifrovanie zálohy. Aj v tejto zálohe sa nachádza niekoľko súborov šifrovaných pomocou 0×835 key.

4.2 Forenzná analýza fyzického zariadenia

Jedná sa o získavanie údajov priamo zo zariadenia. Snažíme sa získať fyzickú bit-by-bit kópiu súborového systému. Táto metóda sa oproti predchádzajúcej vyznačuje značnou výhodou, ktorá sa týka množstva získaných údajov a možnosti obnovenia vymazaných údajov. Obnova údajov súvisí s NAND pamäťou a spôsob akým sú na ňu dáta ukladané. Keďže iOS disponuje bezpečnostnými prvkami ako podpisovanie kódu, ASLR, DEP, sandboxing, tak nie je možné priame spustenie forenzných nástrojov na zariadení s originálnym OS. Navyše môže byť chránené heslom a tým sa automaticky aktivuje Data Protection [1].



Obr. 2: Prelomenie bootovania iOS v DFU

Analýzu je možné vykonať dvoma spôsobmi. Prvým je Jailbreak a druhým nabootovanie upraveného iOS, ktorý obsahuje forenzné nástroje.

iOS zariadenia je možné nabootovať s ich vlastným kernelom a minimalistickou verziou operačného systému. Po načítaní takejto verzie OS s vhodnými nástrojmi je možné zahájiť útoky ako napríklad zistenie hesla, dešifrovanie uložených hesiel, skopírovať súborový systém atď [4].

Z predchádzajúceho obr. 2 [4] vidíme, že na to aby sme si nahrali do zariadenia vlastný OS musíme obísť Trusted boot chain (kryptografické podpisovanie každej časti bootovacieho procesu) konkrétne pri DFU móde (Device Firmware Upgrade – umožňuje zariadeniam obnovu z akéhokoľvek stavu). Toto je možné spraviť pomocou nájdenej zraniteľnosti v BootRom za použitia aplikácie limera1n. Táto zraniteľnosť bola na zariadeniach s procesorom A4 ako: iPad 1, iPhone 4 a staršie, bez obmedzenia iOS. S príchodom iPhone 4S a iPad 2 s novým typom A5 procesorov, Apple túto zraniteľnosť odstránil. Z toho vyplýva, že nie sme schopní vykonať takúto analýzu na zariadeniach s procesorom novším ako A4.

Tento postup je možné vykonať v nasledujúcich krokoch:

Potrebné:

1. Xcode Command Line Tool
2. ldid – slúži na podpisovanie častí kódu
3. Fuse – umožňuje Mac OS X čítať iný ako natívny súborový systém
4. Doinštalovanie Python balíčkov M2crypto construct, progressbar, pycrypto2
5. Mercurial
6. iPhone data protection Utilities
7. Redsn0w
8. iOS firmware

Postup:

1. vytvorenie scriptu na šifrovanie a dešifrovanie kernelu
2. vytvorenie patch kernel a shell scriptov
3. vytvorenie RAM DISK-u
4. načítanie RAM disku cez Redsn0w
5. nadviazanie TCP spojenia cez USB s SSL

Možné útoky:

1. získanie 4 pinového hesla (brute – force)
2. získanie AES kľúčov
3. získanie hesiel z Keychainu
4. skopírovanie dát zo zariadenia (bit-by-bit)
5. dešifrovanie skopírovaných dát
6. obnovenie vymazaných súborov

Ani analýza pomocou Jailbreaku nie je na tom lepšie ako nabootovanie vlastného OS. Pre zariadenia s A4 vieme Jailbreaknuť zaheslované zariadenie a následne na ňom previesť akvizíciu dát, ale pre A5+ procesory nie je možné Jailbreak vykonať kým nie je vypnutá ochrana heslom zariadenia a zálohy. Nie sme schopní získať údaje zo zariadenia s procesorom novším ako A4, ktoré má zapnuté Data Protection.

Obiědenie hesla na iOS zariadenič

Od iOS 4, je pozadaní hesla zariadenia aktivované Data Protection. To chráni súbory na zariadené spolu s keychainom. Kôli tomu potrebujeme pred dešífrovaním súborov a dať správne heslo. Väčšina užívateľov si na svojich zariadeniach neaktivuje ochranu heslom. A ak tak už urobia tak v drvivej väčšine ide o heslo tvorené zo 4 numerických znakov. [5]

Overovanie hesla sa vykonáva na dvoch rôznych úrovniach. Prvou je na úrovni springboardu (užívateľské rozhranie iOS) a druhou je na úrovni jadra systému. Spôsob akým sa snažíme v našom prípade získať heslo je útok hrubou silou. Ak by sme tento útok skúsili realizovať na úrovni springboardu, tak môže dôjsť po niekoľkých neúspečných pokusoch ku zmazaniu zariadenia, poprípade enormného zvýšenia oneskorenia medzi jednotlivými pokusmi. Tieto obmedzenia však nie sú na úrovni jadra systému. Preto sa využíva práve tento spôsob. Nakoľko je kľúč hesla odvodený od hesla užívateľa a UID zariadenia, tak je nutnosťou vykonávať útok hrubou silou priamo na zariadení.

Akonáhle je zariadenie chránené len numerickými znakmi, ľubovoľnej dĺžky tak iOS zobrazí len numerickú klávesnicu. Táto skutočnosť nám môže napomocť

pri zvolení útoku. Na vykonanie útoku môžeme vytvoriť jednoduchý Python skript. [5]

Komunikácia so zariadením je cez port 1999. Skript sa prípojí na zariadenie a zozbiera základné údaje o zariadení (sériové číslo, UDID a ďalšie), unikátne kľúče zariadenia a stiahne systémový keybag. Následne sa pokúsi hrubou silou prelomiť numerické heslo dĺžky 4. Postupne prechádza všetky možnosti od 0000 až po 9999. Tento skript sa dá upraviť podľa požiadaviek aj na iné typy hesiel.

5 Modelová situácia

Všetky predchádzajúce poznatky sa dajú využiť na relatívne jednoduché a rýchle získanie údajov z iOS zariadení (v prípade zapnutej Data Protection zariadenia procesorom A4).

Predstavme si, že sme pripravený útočník (pripravené všetky nástroje spolu s upraveným iOS) niekde v reštaurácii. Získame prístup k iPhonu 4 16 GB s iOS 6.1.3, ktorý je chránený 4-pinovým numerickým heslom. Chceme získať jeho kompletnú bit-by-bit kópiu dát. Tú však nebudeme analyzovať na mieste, aby sme mohli vrátiť zariadenie, čím skôr bez povšimnutia. Túto úlohu je možné zvládnuť od 30 do 45 minút. 1–18 minút trvá prelomenie hesla za zvyšný čas vytvoríme kópiu údajov zo zariadenia cez USB. Po útoku sme schopní dešifrovať všetky údaje (kontakty, správy, maily, ...) spolu s heslami uloženými v Keychaine.

6 Bankové aplikácie

V poslednej dobe vzrástlo využívanie smartfónov aj v oblasti bankovníctva. Každá banka chce svojim zákazníkom uľahčiť kontrolu a správu ich účtov. Preto je logickým dôsledkom, že vyvíjajú aplikácie na mobilné telefóny, ktoré majú zákazníci neustále pri sebe a sú vo veľkej miere pripojené na internet.

V tejto časti sa venujeme bezpečnosti bankových aplikácií slovenských bánk, vyvinutých pre mobilnú platformu iOS. Cieľom je pokúsiť sa získať, čo najviac citlivých údajov, ktoré súvisia s bankovým účtom v konkrétnej banke. Používame forenznú analýzu popísanu v predchádzajúcej kapitole. Zamerali sme sa však na dáta konkrétnych aplikácií a hlavne prístupových údajov.

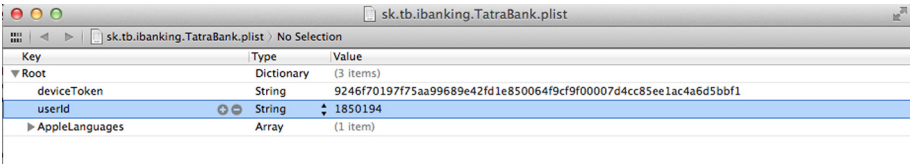
Testovali sme nasledovné aplikácie slovenských bánk:

- Platby a Účty – Slovenská sporiteľňa
- Tatra banka – Tatra banka
- BankAir – UniCredit bank

Na základe vykonanej analýzy sme dospeli k záveru, že aplikácie sú voči tejto forme získavania údajov odolné, a banky si na bezpečnosti svojich aplikácii dali

záležať. Aplikácie prešli aj niekoľkými aktualizáciami, kde boli už opravené rôzne bezpečnosté hrozby.

Z aplikácii sa nám nepodarilo získať žiadne údaje, ktoré by predstavovali pre užívateľa významné bezpečnostné riziko. Aplikácia UniCredit banky si neukladá na zariadenie, žiadne čitateľné údaje. Pri aplikáciach Tatra banky a Slovenskej sporiteľne je možné získať prihlasovacie ID do internet bankingu (ktoré sa dajú použití aj cez web rozhranie a teda sa dajú zneužití pre DoS útok – stačí sa pod nimi 3-krát neúspešne prihlásití a zabránití tak legálnemu bankovému klientovi používati internetbanking). Slovenská sporiteľňa si naviac ukladá aj číslo účtu a logá firiem, s ktorými užívateľ nadviazal finančný styk. Tieto údaje však nepredstavujú bezpečnostné riziko.



Obr. 3: Obsah súboru sk.tb.ibanking.TatraBank.plist

Aplikácie vykonávajú prihlasovanie pomocou tokenov a všetky operácie sú vykonávané na servery. Jediným možným spôsobom, ako sa dostať do aplikácie je získať prístupové užívateľské heslo. Hrubou silou to nie je možné, nakoľko po troch nesprávnych pokusoch sa aplikácia zablokuje a treba vykonať kroky ako pri prvom prihlásení. Najjednoduchšie heslo má Slovenská sporiteľňa, nakoľko ide len o 4 numerické znaky. Ďalšími možnosťami je sociálne inžinierstvo alebo iné formy postranných kanálov. V prípade Slovenskej sporiteľne by sa mohlo zneužiť SMS overovanie pri platbách. Potrebovali by sme však do zariadenia dostať škodlivý kód, ktorý by odchytával komunikáciu medzi zariadením a serverom a pozmenil by odosielaný formulár a následne aj formulár, ktorý sa príjme. Takto by sme mohli pozmeniť odosielanú sumu a číslo účtu, na ktoré sa peniaze posielajú bez povšimnutia užívateľa.

7 Záver

Ako bolo ukázané iOS zariadenia s procesormi A4 nie sú až také bezpečné ako tvrdil výrobca a je relatívne jednoduché z nich získať citlivé údaje majiteľa. Medzi tieto údaje patria kontakty, maily, SMS, heslá uložené v Keychaine a mnohé ďalšie. To však neplatí pre novšie zariadenia, na ktorých sú aktivované všetky bezpečnostné systémy, ktorými táto platforma disponuje.

Literatúra

[1] Zdziarski, J. *Hacking and Securing iOS Application.* USA : O'Reilly Media, Inc, 2012. 336 s. ISBN 978-1-449-31874-1.

[2] iPhone Forensics – on iOS 5.
http://www.securitylearn.net/2012/01/10/iphone-forensics-on-ios-5/, 2012.

[3] Hoog, A., Strzempka, K.: iPhone Forensics,
https://viaforensics.com/resources/white-papers/iphone-forensics/, 2010.

[4] Bédrune, J. B., Sigwald, J.: iPhone data protection in depth
http://esec-lab.sogeti.com/dotclear/public/publications/
11-hitbamsterdam-iphonedataprotection.pdf

[5] iphone-dataprotection, iOS forensics tools, 2012,
http://code.google.com/p/iphone-dataprotection/

[6] iPBA2 iOS Backup Analyzer, 2013, Mario Piccinelli,
http://ipbackupanalyzer.com/development-team

[7] Rossignol, J., Over 22 Million iOS Devices Running Cydia, 2013
http://www.ifans.com/blog/70644/

[8] limera1n, http://limera1n.com

Mobilní zařízení a jejich využití v současné kryptografii

Jan Hajný, Lukáš Malina

E-mail: hajny@feec.vutbr.cz, crypto.utko.feec.vutbr.cz

Abstrakt

S nárůstem výkonu mobilních zařízení stoupá četnost jejich nasazení v náročných kryptografických výpočtech. Zařízení jako jsou programovatelné čipové karty a smart-phony se stávají častou platformou pro implementaci kryptografických algoritmů a protokolů. Stále však existuje mnoho kryptografických aplikací, kde je výkon mobilních zařízení limitujícím faktorem. V příspěvku jsou porovnány možnosti a analyzovány výsledky výkonových testů všech hlavních platforem programovatelných čipových karet (JavaCard, .NET, MultOS). Výsledky jsou srovnány s testy na zařízeních s OS Android. Výkon čipových karet a mobilních zařízení je pak analyzován s ohledem na aplikace v tzv. Privacy-Enhancing Technologies (PETs), tedy aplikace na ochranu soukromí a digitální identity.

Klíčová slova: Kryptografie, soukromí, benchmark, primitiva, protokoly důkazů znalosti, smartkarty, smartphone.

1 Úvod

Tento článek se zabývá problematikou testování výkonu čipových karet. Hlavním cílem je zjistit, zda současné čipové karty umožňují praktickou implementaci pokročilých kryptografických schémat pro ochranu soukromí (Privacy-Enhancing Technologies – PETs), jakými jsou např. atributové autentizační protokoly či tzv. pověření založená na atributech (Attribute Based Credentials – ABCs). Implementace PETs na čipových kartách je důležitá především pro budoucí využití v elektronických osobních dokladech a přístupových kartových technologiích. Díky implementaci PETs na čipových kartách je možné prokázat osobní atributy, jako jsou např. věk, národnost či řidičské oprávnění, zcela anonymně a bez

možnosti danou osobu sledovat. Otevírá se tak prostor pro autentizační systémy nové generace, které umožní ověřovat klienty a zároveň poskytovat pokročilé funkce pro ochranu soukromí. Požadavek na vývoj těchto systémů a jejich budoucí začlenění do současné infrastruktury je obsažen ve strategických plánech jak EU [1] tak USA [2].

Hlavní problém při implementaci PETs na současných čipových kartách je nedostatečný výpočetní výkon v kryptografických operacích, respektive jejich velmi omezená podpora. Současné technologie PETs jsou do značné míry založeny na tzv. důkazech znalosti (PK – proof of knowledge) pomocí protokolů s nulovou znalostí (ZK – zero-knowledge protocols). Tyto protokoly umožňují matematicky prokazatelně bezpečné ověření znalosti tajných klíčů, pravosti výpočtů atp. PK protokoly však pracují, podobně jako asymetrické šifrovací systémy, s čísly o velikosti 1 024–2 048 bitů. S těmito čísly je nutné počítat operace modulární aritmetiky, včetně násobení a mocnění. Ne všechny karty však tyto operace přímo podporují, což značně stěžuje implementaci PETs technologií.

V tomto článku je publikován základní přehled operací, které jsou podporovány na čipových kartách a které jsou využitelné v moderní kryptografii, především u kryptosystémů na ochranu soukromí. Jedná se především o operace modulární aritmetiky a dále pak o jednoduché podpůrné funkce, jako např. hashe či generátory náhodných čísel. Výkon současných programovatelných karet ve vybraných operacích je vzájemně srovnán a porovnán s výkonem mobilních telefonů s OS Android. Článek obsahuje pouze základní přehled, další informace lze nalézt v publikaci [3].

2 Zvolená zařízení a nastavení benchmarků

Výkonové testy byly provedeny na všech hlavních platformách programovatelných čipových karet, konkrétně na platformách JavaCard [4], .NET cards [5] a MultOS [6]. Dále testy proběhly na zařízeních s operačním systémem Android.

JavaCards

V testech byly použity karty Oberthur Technologies ID-One Cosmo V7.0-A [7, 8] a Gemalto TOP IM GX4 [9] se specifikací v Tabulce 1.

.NET Smart-cards

S OS .NET byly použity karty Gemalto .NET V2+ se specifikací v Tabulce 2.

Tab. 1: Specifikace karet platformy JavaCard

Softwarové specifikace		
Typ karty	Oberthur ID-One V7.0-A	Gemalto TOP IM GX4
OS	JavaCard	JavaCard
Transfer. protokol	T=0, T=1	T=0, T=1
Asym. algoritmy	RSA do 2 048 b, EC 521 b	RSA do 2048 b
Sym. algoritmy	DES, 3DES, AES	3DES, AES
Hash	SHA1, SHA2	SHA1
Hardwarové specifikace		
Čip	Atmel AT90SC256144RCFT	S3CC9TC
CPU	8/16 bit	16 bit
Interní/Externí frek.	40 MHz/3.4 MHz	Neznámá
RAM	8 kB	10 kB
ROM/EEPROM	256 kB/144 kB	384 kB/74 kB
Teplotní rozsah	−25 °C až +85 °C	−25 °C až +85 °C
Modulární API	Ne	Ne

Tab. 2: Specifikace karet s OS .NET a MultOS

Softwarové specifikace			
OS	.NET	MultOS	MultOS
Typ karty	.NET V2+	ML2-80K-65	ML3-36K-R1
Asym. algoritmy	RSA do 2 048 b	RSA do 2 048 b, EC do 384 b	RSA do 2 048 b, EC 512 b
Sym. algoritmy	3DES, AES	DES, 3DES, AES	DES, 3DES, AES
Hash	SHA1, SHA2, MD5	SHA1, SHA2	SHA1, SHA2
Hardwarové specifikace			
Čip	SLE 88CFX4000P	SLE66CLX800PEM	SLE78CLXxxxPM
Arch.	32 bit	16 bit	16 bit
Int./Ext. frek.	66 MHz/10 MHz	30 MHz/7.5 MHz	33 MHz/7.5 MHz
RAM	16 kB	702+960 B	1 088+960 B
ROM/EEPROM	80 kB/400 kB	236 kB/78 kB	280 kB/60 kB
Teplotní rozsah	−25 °C až +85 °C	−25 °C až +85 °C	−25 °C až +85 °C
Modulární API	Ne	Ano	Ano

MultOS Smart-cards

Poslední testovanou platformou jsou karty MultOS. Byly použity karty MultOS ML2-80K-65 a ML3-36K-R1, jejich popis je v Tabulce 2.

Mobilní zařízení

Mobilní telefony a tablety představují zajímavou aletrnativu k čipovým kartám. Mobilní telefon nosí uživatelé téměř stále u sebe, jeho využití pro ověření se tak přímo nabízí. Výhodou je vyšší výkon telefonu pro kryptografické operace. Do testů byly zahrnuty zařízení specifikované v Tabulce 3.

Tab. 3: Specifikace mobilních zařízení s OS Android

Softwarové specifikace			
Typ zařízení	Samsung Galaxy S i9000	Samsung Galaxy Nexus I9250M	ASUS TF 300T
Android verze	v2.1 (Eclair)	v4.0 (ICS)	v4.0 (ICS)
Hardwarové specifikace			
Čip	Cortex-A8	Dual-core Cortex-A9	Quad-core Cortex-A9
Frekvence	1 GHz/45 nm	1.2 GHz/45 nm	1.2 GHz/45 nm
GPU	PowerVR SGX540	PowerVR SGX540	ULP GeForce
RAM	512 MB	1 024 MB	1 024 MB
ROM/Úložiště	2 GB/8(16) GB	2/16 GB	2 GB/16(32) GB

2.1 Měřené operace a velikost čísel

Pro testy byly vybrána primitiva, která se nejčastěji používají v moderních kryptografických systémech na ochranu soukromí [10, 11, 12]. Tyto operace představují jádro mnoha komplexních kryptosystémů, jsou zde použity jako modulární stavební bloky.

- **RNG – Random Number Generation**: na všech platformách a zařízeních byl měřen čas generování velkých pseudonáhodných čísel o velikosti 160 bitů (operace **RNG_160**) a 560 bitů (operace **RNG_560**).
- **Hash Functions**: na všech platformách a zařízeních byl měřen čas výpočtu následujících hashovacích funkcí.
  - **SHA1_4256**: SHA1 z 4 256 bitů náhodných dat.
  - **SHA1_7328**: SHA1 z 7 328 bitů náhodných dat.
  - **SHA1_20000**: SHA1 z 20 000 bitů náhodných dat.
  - **SHA2_8448**: SHA2 z 8 448 bitů náhodných dat.
  - **SHA2_14592**: SHA2 z 14 592 bitů náhodných dat.
  - **SHA2_20000**: SHA2 z 20 000 bitů náhodných dat.
- **Modulární aritmetika s velkými čísly**:
  - **MExp1024_160**: Modulární mocnění s 1 024 b modulem a 160 b exponentem.

  - **MExp1024_368**: Modulární mocnění s 1 024 b modulem a 368 b exponentem.
  - **MExp2048_160**: Modulární mocnění s 2 048 b modulem a 160 b exponentem.
  - **MExp2048_560**: Modulární mocnění s 2 048 b modulem a 560 b exponentem.
  - **MMult1024**: Modulární násobení s 1 024 b modulem a operandy.
  - **MMult2048**: Modulární násobení s 2 048 b modulem a operandy.

- **Aritmetické operace s velkými čísly**: byly implementovány následující nemodulární operace.
  - **Mult320**: Násobení dvou 320 b čísel.
  - **Sub400**: Odečítání dvou 400 b čísel.

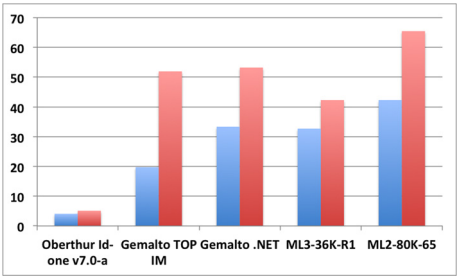
Bitové délky čísel, které byly použity v operacích, odpovídají běžným délkám parametrů v PETs kryptosystémech. Byly tedy zvoleny tak, aby bylo možné odadnout dobu výpočtu, která je nutná pro běh schémat založených na výše uvedených primitivech.

3 Benchmarky

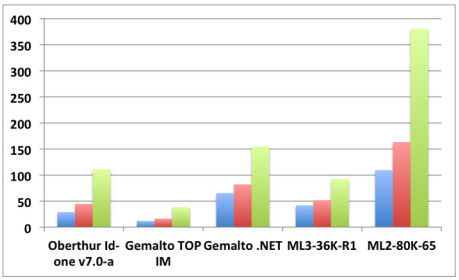
Každá operace byla měřena 25×. Níže je uveden průměr naměřených hodnot. Změřené doby neobsahují čas nutný k vlastní komunikaci s kartou, pouze samotné operace. Pro výpočty byly využity API systémů tam, kde byly k dispozici.

3.1 Benchmarky na smartkartách

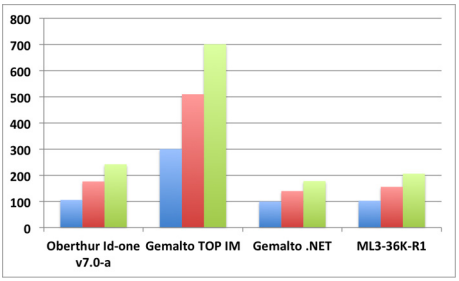
Všechny grafy zobrazují dobu výpočtu v milisekundách operací v titulku.



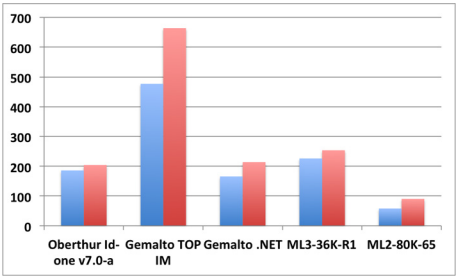
Obr. 1: **RNG_160** (mod.) a **RNG_560** (červ.)



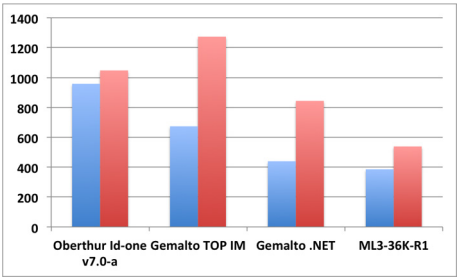
Obr. 2: **SHA1_4256** (mod.), **SHA1_7328** (červ.) a **SHA1_20000** (zel.)



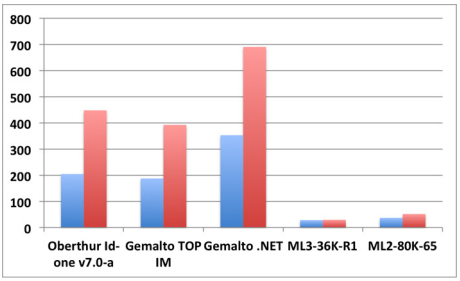
Obr. 3: **SHA2_8448** (mod.), **SHA2_14592** (červ.) a **SHA2_20000** (zel.)



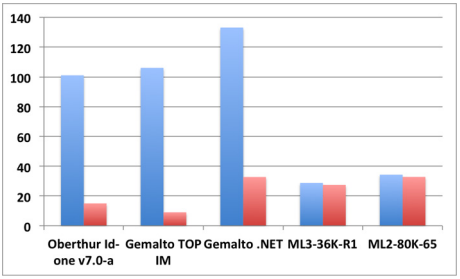
Obr. 4: **MExp1024_160** (mod.) a **MExp1024_368** (červ.)



Obr. 5: **MExp2048_160** (mod.) a **MExp2048_560** (červ.)



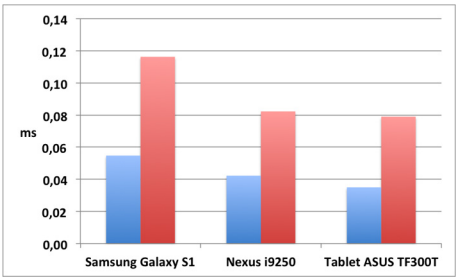
Obr. 6: **MMult1024** (mod.), **MMult2048**



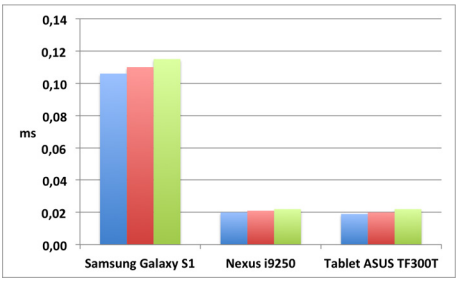
Obr. 7: **Mult320** (mod.) a **Sub400**

3.2 Benchmarky na zařízeních s OS Android

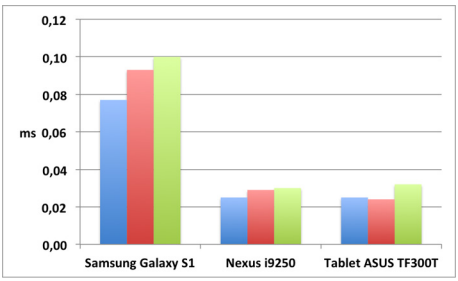
Všechny grafy zobrazují dobu výpočtu v milisekundách operací v titulku.



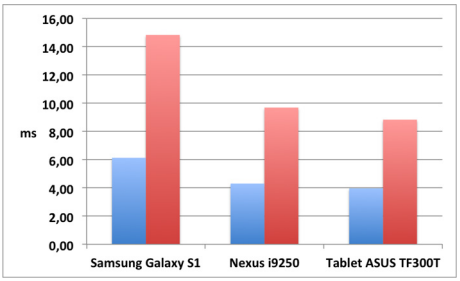
Obr. 8: **RNG_160** (mod.) a **RNG_560** (červ.)



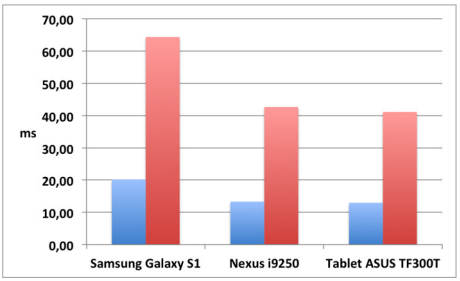
Obr. 9: **SHA1_4256** (mod.), **SHA1_7328** (červ.) a **SHA1_20000** (zel.)



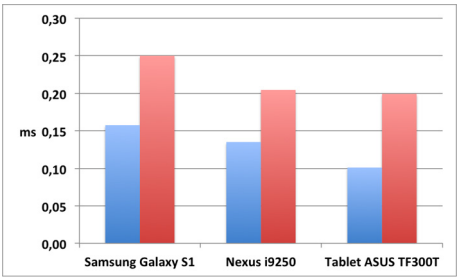
Obr. 10: **SHA2_8448** (mod.), **SHA2_14592** (červ.) a **SHA2_20000** (zel.)



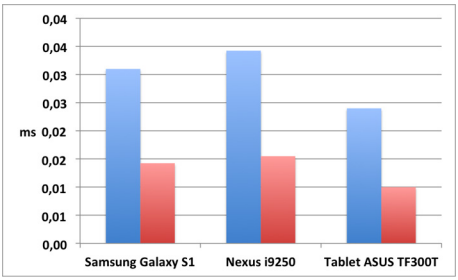
Obr. 11: **MExp1024_160** (mod.) a **MExp1024_368** (červ.)



Obr. 12: **MExp2048_160** (mod.) a **MExp2048_560** (červ.)



Obr. 13: **MMult1024** (mod.), **MMult2048** (červ.)



Obr. 14: **Mult320** (mod.) a **Sub400** (červ.)

3.3 Analýza výsledků

Smartkarty

Všechny operace bylo možné implementovat na vybraných programovatelných čipových kartách, až na výjimku, kartu MultOS ML2-80K-65, která nepodporuje SHA2 a 2 048 b modulárního mocnění. Karty jsou však relativně pomalé, pro praktické využití je možné volit pouze 1 024 b grupy. U 2 048 b už by nebylo možné implementovat ani ty nejjednodužší atributové systémy, čas ověření uživatelů by rostl k 10 s, což je neúnosné. Z pohledu PETs systémů jsou velmi důležité karty MultOS, jako jediné totiž obsahují přímou podporu pro modulární násobení a mocnění, což velmi citelně urychluje dobu výpočtu těchto operací. Při volbě vhodných parametrů (1 024–1 392 b) je možné implementovat na platformě MultOS všechny hlavní PETs systémy (U-Prove [10] od Microsoftu, HM12 [12] i modifikovaný Idemix [11] od IBM) s dobou ověření uživatele okolo 1–2 s. Naopak, využití vyšší bezpečnosti s 2 048 b grupami se s výkonem současných karet zdá být zatím nereálné.

Zařízení s OS Android

Všechny vybrané operace bylo možné implementovat i na zařízeních s OS Android, i díky přímé podpoře operací s velkými celými čísly. Díky mnohonásobně vyššímu výkonu (u některých operací až o dva řády) je možné implementovat primitiva i nadstavbové PETs systémy v bezpečných grupách velikosti 2 048 b. Díky podpoře NFC se tak smartphony stávají velmi důležitou alternativou pro autentizační a přístupové systémy, které v minulosti využívaly pouze čipové karty.

4 Závěr

V článku byly stručně představeny výsledky výkonostních testů programovatelných čipových karet a mobilních telefonů s OS Android. Testy byly primárně zaměřeny na operace využívané v kryptosystémech na ochranu soukromí, především schématech atributové autentizace a atributových pověření. Z výsledků vyplývá vhodnost platformy MultOS pro zmíněný typ operací, především díky přímé podpoře modulárních operací mocnění a násobení. Programovatelné karty jsou ale stále omezeny svým výkonem, práce s čísly většími než 1 024 b je stále nepraktická s ohledem na dobu trvání operací. Toto omezení se však nevztahuje na mobilní telefony a tablety, kde je možné využívat potřebné operace i v grupách o velikosti modula 2 048 b a výše.

Poděkování

Výzkum byl podpořen projekty SIX CZ.1.05/2.1.00/03.007, TAČR TA02011260 a MPO FR-TI4/647.

Literatura

[1] Naumann, I., Hogben, G.: *Privacy features of eid cards.* Network Security Newsletter 2008, 2008, 9–13.

[2] The White House: *National strategy for trusted identities in cyberspace*, 2011. http://www.whitehouse.gov/sites/default/files/rss_viewer/NSTICstrategy_041511.pdf.

[3] Hajný, J., Malina, L., Tethal, O.: Performance evaluation of primitives for privacyenhancing cryptography on current smart-cards and smart-phones. In *8th DPM InternationalWorkshop on Data Privacy Management.* Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

[4] Oracle: *Javacard.* 2013 http://www.oracle.com/technetwork/java/javacard/downloads/index.html.

[5] Gemalto: *.net card.* 2013. http://www.gemalto.com/products/dotnet_card/

[6] MultOS: *Multos card.* 2013. http://www.multos.com.

[7] *Id-one v7.0.* Technical report, French Network and Information Security Agency (Agence Nationale de la Sécurité des Systemes d'Information (ANSSI)), 2009. http://www.ssi.gouv.fr/IMG/certificat/anssi-cc-cible_2009-36en.pdf.

[8] Atmel: *At90sc256144rcft datasheet.* 2007. http://datasheet.elcodis.com/pdf2/104/7/1040758/at90sc256144rcft.pdf.

[9] NIST: *Gemxpresso r4 e36/e72 pk – multiapp id 36k/72k – top im gx4.* 2009. http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp771.pdf.

[10] Paquin, C.: *U-prove cryptographic specification v1.1, 2011.* Technical report. http://research.microsoft.com/apps/pubs/default.aspx?id=166969.

[11] Camenisch, J., et al.: *Specification of the identity mixer cryptographic library, 2010.* Technical report. http://domino.research.ibm.com/library/cyberdig.nsf/1e4115aea78b6e7c85256b360066f0d4/eeb54ff3b91c1d648525759b004fbbb1?OpenDocument.

[12] Hajný, J., Malina, L.: Unlinkable attribute-based credentials with practical revocation on smart-cards. In Mangard, S., ed.: *Proceedings of the Smart Card Research and Advanced Application Conference CARDIS 2012.* Lecture Notes in Computer Science 7771, Springer Berlin Heidelberg, 2013, 62–76.

White-box cryptography

Dušan Klinec

E-mail: xklinec@fi.muni.cz

Abstract

This article is focused on a study of security issues related to an execution of cryptographic algorithms in an untrusted environment. It mainly studies whitebox cryptography methods of transforming algorithms in such a way they resist attacks like key-extraction and inverting in some extent. Particularly it examines whitebox transformations of AES cipher and attacks on these transformations. Transformations construction and implementation is described. We also discovered that the known attack works also on AES transformation using dual ciphers by Karroumi [1] that was supposed to resist the attack. The new improvements for increasing a resistance of transformations to known attacks were proposed.

1 Introduction

In the last few decades we have been witnessing a development in a field of outsourced computations and storage. The rising prevalence of this computing model slightly changes the classical attacker model that cryptography used to dealt with, what gave rise to a *mobile cryptography*.

The classical goals the cryptography addresses are confidentiality, data integrity, authentication and non-repundation [2]. From the data confidentiality perspective, the typical scenario is two remote parties, Alice and Bob, want to communicate via untrusted channel, while the computations on both sides are considered as trusted. A potential attacker resides in a communication link. A bunch of cryptography primitives addressing security issues in this scenario was invented, analyzed and widely used, e.g. symmetric and asymmetric cryptosystems, digital signatures, authentication protocols, etc.

But with the expansion of outsourced computations and storage we are getting to a situation that Alice does not trust even to Bob, but wants to use Bob's resources for her own purpose. Such outsourcing rises concerns about the loss

of privacy of private data what poses the potential barrier in adopting cloud services widely. To ensure the privacy, data is encrypted. The major problem with this model is that in order to evaluate a function over data, e.g. searching in an encrypted database, data has to be decrypted first. This poses an another additional overhead. The *fully homomorphic encryption* provides a solution for these issues.

Another major part of use cases is the protecting a private function computed in an untrusted environment. A typical example of the function to be protected is the license code verification embedded in a software or it is a software that provides access to some protected material, e.g. copyrighted content. The major goal is to protect these functions from analysis, tampering or extraction of a cryptographic material. Software protection techniques like an *obfuscation* addressing these issues are used in practice. This thesis is devoted to a *white-box cryptography*, the field of cryptography that studies the level of security of cryptographic algorithms executed in an untrusted environment.

Here we study in particular a transformation of the AES [3] implementation in such a way that they provide some level of security when executed in an untrusted environment. This transformed implementation has embedded a symmetric key inside and the main goal of the transformation is to resist practical attacks attempting its extraction.

2 Overview

Computing in an untrusted environment is closely related to the notion of *mobile cryptography* [4] which was established two decades ago. It analyzes security problems raised by a concept of mobility of an executable code. The executable code that acts autonomously on behalf a user in collecting and processing information is denoted as a *mobile agent*. Mobile cryptography mainly studies two security threats:

1. protection of the host from malicious mobile code
2. protection of mobile code from the malicious host

The former threat can be mitigated to an acceptable level with countermeasures like sandboxing, virtualization and code signing [2], what is widely adopted by current anti-virus protection software and operating systems.

The latter is much difficult to address. Existing techniques provide protection to some extent, making tampering the code on malicious host difficult for ordinary attacker, but there are no guarantees of protection against very strong and determined attacker with enough resources to invest in the attack.

In order to make tampering of the software very difficult, specialized, tamper-resistant hardware is often used. It is designed with security concerns in mind

so that very advanced techniques and a large amount of resources is needed in order to attack such device, making it practically impossible for an ordinary attacker. For an example hardware security modules used by banks or certification authorities protecting their secret cryptographic material. Another example is a cryptographic smart card, widely used in use cases with high security requirements.

However, the tamper-resistant hardware is not suitable for many applications, due to its cost and physical nature, e.g. need to be distributed somehow, can be lost, forgotten, unintentionally damaged, etc. Then it is preferable to use software-based protection techniques for their low cost and flexibility. The downside of this approach is a limited strength.

3 Obfuscation

An *obfuscation* is another technique addressing the same problem, protecting a software implementation. Roughly speaking, the major principle is the transformation of the code to a form, that is very difficult to analyze and eventually to modify. The potential attacker should not be able to gain any extra knowledge[1] from the running program, in the ideal case, while the original functionality of the program is preserved.

The program obfuscation received an attention when Barak *et al.* formalized the notion of obfuscation in [5], providing significant theoretical result that it is impossible to create a generic obfuscator. They did so by showing the existence of a (contrived) family of functions that are unobfuscatable, i.e. the family of functions always leaking some information. They used assumption of existence of one-way functions.

On the other hand, later was published a first positive result [6] claiming it is possible to construct some provably secure obfuscators for *point functions*. Point function accepts a single input string and reject all other inputs. It was used to obfuscate complex access control functionalities.

The first positive obfuscation result for a traditional cryptographic functionality (that is significantly more complicated that point functions) was presented by Hohenberger *et al.* [7]. They used slightly modified definition of obfuscation in order to construct a secure obfuscator for re-encryption[2].

The question whether there exist a family of potentially interesting functions for which exist provably secure obfuscators and how to construct them, is a subject of a further research. But the work of Goldwasser *et al.* [8] suggest it is

[1] besides input/output behavior

[2] This functionality takes a ciphertext for message m encrypted under Alice's public key and transforms it into a ciphertext for the same message m under Bob's public key. [7]

unlikely. Namely they state that there exist many natural classes of functions that cannot be obfuscated w.r.t. auxiliary input.

An *approximate obfuscation* defined by Barak *et al.* [5] is a relaxation of the functionality requirement of the obfuscated program. They presented impossibility result in the case when an obfuscated program deviates from the original program only with a negligible probability and allows this event to depend only on the coin tosses of the obfuscator. Recently Bitansky *et al.* [9] improved this impossibility result by hardening the requirements. They showed there exist families of robust unobfuscatable functions for which even approximate obfuscation is impossible. According to their definition, obfuscated program is only required to agree with the original one with probability slightly more than 0.5 on a uniformly sampled input, what was the open problem till then.

In a practice, the obfuscation is widely used as a software protection technique that provides some level of protection from attackers, but it often lacks some proof of security. In major cases it is collection of techniques that makes the static and/or run-time analysis of a program significantly more difficult, but it does not rule out the probability of an successful attack by a strong and determined attacker. A rich collection of state of the art obfuscation techniques, protecting from static and run-time analysis can be found in the dissertation thesis of J. Cappaert [10].

The concept of obfuscation is closely related to *computing with a private function*.

4 Computing with encrypted data

As is mentioned in the introduction, the fundamental question whether data can be manipulated without being decrypted has been attracting attention for a long time.

4.1 Secure multiparty communication

The first positive results on this question make use of *interaction*. The concept of a *secure multiparty communication* was introduced by Yao [11] in 1982. Roughly speaking, it enables to evaluate a function over a private data of remote parties, while keeping the private data still confidential. For this, both parties have to follow some protocol. A typical toy example is a well-known Yao's Millionaire's Problem. In this problem two millionaires, Alice and Bob, want to know which is richer, without disclosing their actual wealth. Note the first protocol solving this problem had exponential time, space and communication complexity. This problem has direct applications in e-commerce, e.g. on-line bidding and auctions and data mining [12].

Many protocols for secure multiparty schemes are based on *arithmetic circuits*. This is also a one of cornerstones used in the following chapter, so it is important to describe it.

The function $\mathcal{F}$ is transformed to a network, that forms a directed acyclic graph, of gates performing addition and multiplication operation, what forms an arithmetic circuit that is able to evaluate the function $\mathcal{F}$. It is known that considering arithmetic circuits is without a loss of generality, i.e. any function that is feasible to compute at all can be specified as a polynomial-size Boolean circuit using *and* and *negation*. Note that any such circuit can be simulated by operations in $\mathbb{F}$: Boolean values *true* or *false* can be encoded as 1 resp. 0. Then *negation* of a bit b is $1 - b$ and *and* of bits $b,\ c$ is $b \cdot c$ [12]. The resulting circuit is then evaluated by remote parties in order to compute function $\mathcal{F}$ over their private data. A *depth* of a circuit is the longest path in the circuit.

Ishai *et al.*[13] in 2008 demonstrated a two-party computation protocol of a function $\mathcal{F}$ while communication overhead is a fixed constant factor larger than circuit size of $\mathcal{F}$.

4.2 Fully homomorphic encryption

A cryptosystem is denoted as a fully homomorphic if it supports evaluation of two operations, *addition* and *multiplication*, on ciphertexts where the result after decryption matches the same operation on corresponding plaintexts.

$$a + b = Decrypt\left(Encrypt\left(a\right) + Encrypt\left(b\right)\right) \tag{1a}$$

$$a \cdot b = Decrypt\left(Encrypt\left(a\right) \cdot Encrypt\left(b\right)\right) \tag{1b}$$

The power of the the fully homomorphic system is in its ability to evaluate an arbitrary function over encrypted data, without actually decrypting the data. This is exactly the situation where Alice wants to outsource some computations to Bob, but doesn't want Bob to learn her information. When the function is evaluated homomorphically, the result is again encrypted, so only Alice is able to read it. This is done by using the concept from the previous chapter, *arithmetic circuits*.

The function $\mathcal{F}$ that Alive wants to compute, is converted into the *arithmetic circuit* that computes the same function. This circuit evaluated over plaintext gives the wanted result, but note that it can be evaluated also over ciphertexts using the homomorphic property of the cryptosystem. Intuitively, Alice sends circuit representing $\mathcal{F}$ to Bob, Bob evaluates the circuit on ciphertext and returns a result to Alice. When Alice decrypts the result from Bob, obtains the result of $\mathcal{F}$.

It is important to emphasize that in this use case, the cryptosystem becomes a computational platform, thus the possible space/time overheads slow down entire computation.

History. The concept of *computing with encrypted data* was first proposed by Rivest *et al.* [14] in 1978, a few months ago before introduction of RSA implementation. They suggested that a fully homomorphic encryption may be possible, but were unable to find such scheme. The question whether it is possible to construct a fully homomorphic scheme was an open problem for 30 years.

Some partially homomorphic schemes were known, for example RSA supports homomorphic evaluation of multiplication. There were also some limited homomorphic schemes published, for example [15] in 2005. Their cryptosystem is based on elliptic curves and supports unlimited number of additions and one multiplication operation. Even this restricted scheme has interesting applications, for example efficient election system, as proposed in their paper.

The breakthrough in this field was done by Gentry in 2009 [16]. He demonstrated the fully homomorphic encryption (FHE) scheme is possible to construct, using *ideal lattices*. Since then this field is undergoing a rapid development.

Key ideas. Gentry first constructed *"somewhat homomorphic"* scheme that supports evaluating of low-degree polynomials on ciphertexts (corresponds to evaluating an arithmetic circuit of a small depth). To protect the information (plaintext), it is hidden in a large amount of *noise*. Without going into further details, the main problem is the addition doubles and the multiplication squares the noise level. Once the level exceeds acceptable boundary, decryption is ambiguous even for Alice.

The ingenious idea of a noise reduction is called *refreshing*. It is a process of evaluating decryption circuit homomorphically. Note that such evaluation produces again ciphertext, since the result of homomorphic operation is still encrypted, but the level of noise is normalized.

Using this idea, Gentry built the FHE from the somewhat homomorphic scheme, by periodically applying the refreshing operation when the noise reached the acceptable level.

Both symmetric and asymmetric schemes were proposed.

Recent advances. There are three main FHE schemes known to date:

1. Gentry's original scheme based on ideal lattices. The implementation was introduced by Gentry *et al.* in [17]. The public key has a size 2.3 GB, refreshing operation takes 30 minutes.
2. Dijk's *et al.* [18] scheme DGHV, based on a problem from number theory, approximate Greatest Common Divisor (GCD).
   - Simpler that previous scheme.
   - The latest results by Coron *et al.* [19] from 2012, significantly reduced the public key size to 10.3 MB, refreshing operation takes 11 minutes.

- The result from 2013 by Coron *et al.* [20] added support for performing *batch* operations with plaintexts.
- The fully homomorphic evaluation of AES encryption was performed, with amortized cost 12 minutes per AES block on a standard desktop computer with 32 GB RAM [20].

3. Brakerski *et al.* [21] Ring Learning With Errors (RLWE) scheme, adaptation of previous Learning With Errors (LWE) scheme. The LWE hard problem is to recover $s \in \mathbb{Z}_q^n$ given a sequence of approximate random linear equations of s.

   - The improvement by Brakerski *et al.* [22] changed the noise management via *modulus switching*. The refreshing procedure as used by Gentry is not necessary in this case. The noise is reduced gradually after each multiplication, protecting from growing exponentially.
   - Improvement by Gentry *et al.* [23] adds batch operation, using a *cyclotomic number field*[3].
   - The fully homomorphic evaluation of AES encryption was performed [24] with amortized time 37 minutes per AES block on a standard desktop computer with 256 GB RAM.

Batch operation, also called plaintext "packing", is a technique where multiple independent plaintexts slots are embedded into a single ciphertext, using a proper algebraic structure. Then when an operation is performed on the ciphertext, it has effect like it is performed on the whole vector of plaintexts embedded in the ciphertext. This strongly resembles Single-Instruction-Multiple-Data (SIMD) architecture of a paralell computer. This adds significant improvement, since multiple blocks (like in AES case) are computed simultaneously, what gives better amortized running time of algorithms. Recall that in the original Gentry's scheme, the plaintext space size was only 1 bit.

As an illustration[4] consider that a plaintext space is a group $Z_{15} = Z_3 \times Z_5$, from Chinese Remainder Theorem. Using this structure we obtain 2 slots for plaintext of size 3 and 5. Let's have two ciphertexts c, c' with (p_3, p_5) and (p'_3, p'_5) in their plaintext slots respectively.

Then after $\mathrm{ADD}(c, c')$ the plaintext slots are $(p_3 + p'_3, p_5 + p'_5)$. The analogy holds also for $\mathrm{MULT}(c, c')$ then the plaintext slots are $(p_3 \cdot p'_3, p_5 \cdot p'_5)$.

The batching mechanism proposed in [22] is based on the similar idea but uses ring that optimizes number of plaintext slots in ciphertext, by choosing a more appropriate algebraic structure.

[3] http://www.math.harvard.edu/∼erickson/pdfs/cyclotomic_fields_part_iii.pdf

[4] Example taken from [25]

Somewhat homomorphic encryption schemes. FHE schemes is a very active area of research nowadays, with still better and better improvements on a performance of the schemes, but in spite of this, the practical use of FHE is still out of question due to its computational complexity.

Naehrig *et al.* [26] proposed to sacrifice "fully" property and use just *somewaht homomorphic schemes* (SWHM), with limited number of multiplications. In this setup it is not possible to evaluate arbitrary function, but some families of functions can still be useful. They give examples of an application in medical, financial sectors and advertising.

Boneh *et al.* [27] designed and implemented a protocol for private database queries using somewhat homomorphic encryption.

5 Whitebox cryptography

5.1 Introduction

Whitebox cryptography studies security issues related to an execution of cryptographic algorithms in an untrusted environment, it is than said to be executed in a *whitebox context*.

Whitebox context (also abbreviated as WBC) is itself defined by the attacker model, which was introduced by Chow *et al.* [28] in 2002. The WBC attacker has a full control over execution of the particular algorithm. Namely attacker has the following abilities:

- can observe execution:
  - access to the instructions processing at the moment of the computation
  - trace the algorithm flow
  - sees the memory used
- controls the execution environment — runtime modification:
  - tamper the program memory
  - execute only a specified part of the algorithm (one round of the cipher)
  - modify if-conditions
  - change cycle counters
  - fault induction

It is in contrast to a *blackbox context* (also abreviated as BBC), the standard cryptographic model, where attacker has only access to the output of the cryptographic algorithm. In the BBC the cryptographic algorithm is considered as an oracle/blackbox evaluating some function (an analogy to executing algorithm

in secure environment). Depending on a finer granularity of an attacker model, one can have access only to the algorithm output (ciphertext), or attacker can also query an oracle (chosen plain-text attack) and so on, but has no access to the computation itself.

The cryptographic algorithms (we are mainly interested in symmetric ciphers) were extensively studied for attacks in the BBC in past. They were originaly designed to resist attacks considering only the BBC. But if the context is wrong, it can be a possible entry point for an attacker. Typical example is DRM [5], where software of a vendor (representing the rights owner) is executed in a potentially hostile environment, where user can have motivation to extract protected content without restrictions added by DRM software. In this situation we cannot consider DRM software to be executed in the BBC.

Let's take some symmetric block cipher as an another example. Usually, it is constructed as a keyed permutation (round function) that is repeated several times to add randomness and to improve statistical results of the cipher, increasing security. But if we can inspect such execution, it is very easy to extract encryption keys, since we can read memory during the execution or trace the algorithm flow.

One such whitebox attack is the *Key Whitening Attack* [29]. Key whitening is a technique intended to increase the security of the iterated block cipher. It is typically implemented as adding a key material to the data (usually by simple operation, such as XOR) in the first and the last round. Such key whitening is used by Twofish [30] and in a modified version (only adding the key material in the last round) also by AES [3]. In Key Whitening Attack cipher's binary is modified (we are in whitebox context) in such a way that the output of the cipher will be the key material itself.

Main two attacks in whitebox context are: (1) Key Recovery, i.e. an extraction of a embedded symmetric key. (2) Plaintext recovery under Chosen Plaintext Attack (PR-CPA), e.g. perform decryption with implementation of cipher with embedded encryption that is supposed to be able only to perform encryption.

Whitebox cryptography is closely related to the *obfuscation* mentioned in the section 3. It is also a program transformation, but obfuscation, as defined in the literature, is too restrictive and does not take specific security notios, e.g. cipher invertibility a.k.a. PR-CPA, into account.

The definition of whitebox cryptography could be: "The challenge that white-box cryptography aims to address is to implement a cryptographic algorithm in software in such a way that cryptographic assets remain secure even when subject to white-box attacks. Software implementations that resist such white-box attacks are denoted white-box implementations." [31]

[5] Digital rights management, http://en.wikipedia.org/wiki/Digital_rights_management

5.2 History

Whitebox cryptography is a quite new field of cryptography. The study of the whitebox implementation of the ciphers started by first whitebox implementation of AES [28] and DES [32] by Chow *et al.* in 2002.

At first, the cryptanalysis of DES focused on its simplified variant. The first published in 2002 by Jacob *et al.* uses fault injection [33], another one published in 2005 by Link *et al.* uses statistical analysis [34]. Later cryptanalysis of fully encoded variant of DES was published by Wyseur *et al.* in 2007 using truncated differentials.

The similar case holds for AES. Two years after publishing the whitebox AES scheme the successful cryptanalysis [35] was published by Billet *et al.* that enabled to recover embedded symmetric key in less that 2^{30} steps. Later, in 2008, the generalized version of the previous attack was published [36] by Michiels *et al.* affecting the larger family of ciphers using the same structure as AES.

There were also attempts to fix whitebox AES scheme by adding additional linear mappings and increasing size of the implementation in [37] as a response to the Billet's attack. The attack against improved scheme using a linear equivalence algorithm was published in 2012 [38].

The another attempt, how to fix whitebox AES, was introducing random perturbations [39], complicating algebraic cryptanalysis, but the effective attack was published by Mulder *et al.* [40].

Last, but not least a whitebox AES scheme using dual ciphers [1] was published in 2011. The paper claimed the scheme is robust enough to resist practical attacks on the implementation. We proved this assumption false by finding out the published attack works in the same way on this implementation as on the original one.

5.3 Resources

For more information about white-box cryptography, whitebox-scheme using dual ciphers, attack on this new scheme, implementation of schemes and the attack and proposed solutions, please refer to my diploma thesis available online http://is.muni.cz/th/325219/fi_m/.

References

[1] Mohamed Karroumi. Protecting white-box AES with dual ciphers. In *Proceedings of the 13th international conference on Information security and cryptology*, ICISC'10, p. 278–291, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[3] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AESÚthe advanced encryption standard.* Springer-Verlag, 2002.

[4] Tomas Sander and Christian F. Tschudin. Towards mobile cryptography. In *Security and Privacy — 1998 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 3–6, 1998, Proceedings*, p. 215–224. IEEE Computer Society, 1998.

[5] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, p. 1–18, London, UK, UK, 2001. Springer-Verlag.

[6] Benjamin Lynn, Manoj Prabhakaran, and Amit Sahai. Positive results and techniques for obfuscation. In *In EUROCRYPT '04*, 2004.

[7] Susan Hohenberger, Guy N. Rothblum, Abhi Shelat, and Vinod Vaikuntanathan. Securely obfuscating re-encryption. In *Proceedings of the 4th conference on Theory of cryptography*, TCC'07, p. 233–252, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In —it Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science, FOCS '05, p. 553–562, Washington, DC, USA, 2005. IEEE Computer Society.

[9] Nir Bitansky and Omer Paneth. On the impossibility of approximate obfuscation and applications to resettable cryptography. *IACR Cryptology ePrint Archive*, 2012:729, 2012.

[10] Jan Cappaert. *Code Obfuscation Techniques for Software Protection.* Phd. thesis, Katholieke Universiteit Leuven, 2012.

[11] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, p. 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[12] Ronald Cramer, Ivan Damgard, and Jesper Buus Nielsen. Secure multiparty computation and secret sharing — an information theoretic appoach. [online], accessed 26. 5. 2013, 2012.

[13] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer — efficiently. In *Proceedings of the 28th Annual conference on Cryptology: Advances in Cryptology*, CRYPTO 2008, p. 572–591, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press*, p. 169–179, 1978.

[15] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Proceedings of the Second international conference on Theory of Cryptography*, TCC'05, p. 325–341, Berlin, Heidelberg, 2005. Springer-Verlag.

[16] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, p. 169–178, New York, NY, USA, 2009. ACM.

[17] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Proceedings of the 30th Annual international conference on Theory and applications of cryptographic techniques: advances in cryptology*, EUROCRYPT'11, p. 129–148, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of the 29th Annual international conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'10, p. 24–43, Berlin, Heidelberg, 2010. Springer-Verlag.

[19] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Proceedings of the 31st Annual international conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'12, p. 446–464, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Batch fully homomorphic encryption over the integers. *IACR Cryptology ePrint Archive*, 2013:36, 2013.

[21] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ringlwe and security for key dependent messages. In *Proceedings of the 31st annual conference on Advances in cryptology*, CRYPTO'11, p. 505–524, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, p. 309–325, New York, NY, USA, 2012. ACM.

[23] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Ring switching in bgvstyle homomorphic encryption. In *Proceedings of the 8th international conference on Security and Cryptography for Networks*, SCN'12, p. 19–37, Berlin, Heidelberg, 2012. Springer-Verlag.

[24] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. *IACR Cryptology ePrint Archive*, 2012:99, 2012. informal publication.

[25] Craig Gentry. Fully Homomorphic Encryption: current state of the art. Africacrypt 2012, [online], accessed 26. 5. 2013, 2012.

[26] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, p. 113–124, New York, NY, USA, 2011. ACM.

[27] Dan Boneh, Craig Gentry, Shai Halevi, and Frang Wang. Private database queries using somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/449, 2012.

[28] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. Van Oorschot. White-box cryptography and an AES implementation. In *Proceedings of the Ninth Workshop on Selected Areas in Cryptography*, SAC 2002, p. 250–270. Springer-Verlag, 2002.

[29] Tim Kerins and Klaus Kursawe. A cautionary note on weak implementations of block ciphers. In *In 1st Benelux Workshop on Information and System Security*, WISSec 2006, p. 12, 2006.

[30] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128-bit block cipher. In *in First Advanced Encryption Standard (AES) Conference*, 1998.

[31] Brecht Wyseur. White-box cryptography: Hiding keys in software. [online], accessed 26. 5. 2013, 2012.

[32] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. Van Oorschot. A white-box DES implementation for DRM applications. In *In Proceedings of ACM CCS-9 Workshop DRM*, p. 1–15. Springer, 2002.

[33] Matthias Jacob, Dan Boneh, and Edward W. Felten. Attacking an obfuscated cipher by injecting faults. In Joan Feigenbaum, editor, *Digital Rights Management Workshop, volume 2696 of Lecture Notes in Computer Science*, p. 16–31. Springer, 2002.

[34] Hamilton E. Link and William D. Neumann. Clarifying obfuscation: Improving the security of white-box des. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) – Volume I – Volume 01*, ITCC '05, p. 679–684, Washington, DC, USA, 2005. IEEE Computer Society.

[35] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *Proceedings of the 11th international conference on Selected Areas in Cryptography*, SAC'04, p. 227–240, Berlin, Heidelberg, 2005. Springer-Verlag.

[36] Wil Michiels and Paul Gorissen. Mechanism for software tamper resistance: an application of white-box cryptography. In *Proceedings of the 2007 ACM workshop on Digital Rights Management*, DRM '07, p. 82–89, New York, NY, USA, 2007. ACM.

[37] Yaying Xiao and Xuejia Lai. A secure implementation of white-box AES. In *Computer Science and its Applications, 2009. CSA '09. 2nd International Conference on*, p. 1–6, 2009.

[38] Yoni De Mulder, Peter Roelse, and Bart Preneel. Cryptanalysis of the Xiao – Lai whitebox AES implementation. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography*, volume 7 707 of *Lecture Notes in Computer Science*, p. 34–49. Springer, 2012.

[39] Julien Bringer, Hervé Chabanne, and Emmanuelle Dottax. White box cryptography: Another attempt. *IACR Cryptology ePrint Archive*, 2006:468, 2006.

[40] Yoni De Mulder, Brecht Wyseur, and Bart Preneel. Cryptanalysis of a perturbated white-box AES implementation. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT*, volume 6 498 of *Lecture Notes in Computer Science*, p. 292–310. Springer, 2010.

Uchovávání dat v SSD

Aleš Padrta, Karel Nykles

E-mail: {apadrta, nykles}@cesnet.cz

Klíčová slova: SSD, Gargabe Collector, Controler, Self-Corrosion, Forensics

Abstrakt

Moderní paměťová média typu Solid State Drive (SSD) mohla dosáhnout svých skvělých vlastností pouze přechodem na jinou technologii než klasické pevné disky (HDD). Jiná technologie pochopitelně vykazuje jiné chování k ukládaným a mazaným datům, což má přímý vliv na objem smazaných dat dohledatelných v obrazu média. Článek mapuje interní mechanismy SSD, které ovlivňují zacházení s daty, od tranzistoru s plovoucím hradlem až po softwarové vlastnosti řadiče jako FTL, garbage collector a deduplikace/komprese dat. Také je rozebrána komunikace s OS pomocí protokolu TRIM a ovladače zařízení. Součástí jsou také výsledky praktických experimentů s SSD.

Abstract

Modern storage media such as Solid State Drive (SSD) has to switch to different technology (than traditional hard disk drives) to achieve its great features. Other technology obviously implies a different approach to data handling, especially the handling of deleted data. The article maps the internal SSD mechanisms that affect data handling, from transistor with floating gate to the software controller properties such as FTL, garbage collector and data deduplication/compression. Communication protocol with the OS using TRIM and device drivers is also discussed. The results of some practical experiments with the SSD are included.

Při analýze obrazů klasických disků (HDD) lze běžně nalézt také smazané soubory nebo jejich fragmenty. Typicky v alokačních jednotkách označených jako volné a u většiny souborových systémů také ve slacku. Způsob, jakým je zacházeno se smazanými daty závisí na použitém médiu, souborovém systému, případně i operačním systému, resp. ovladači daného zařízení. Změna média z HDD

na SSD, které je založeno na jiné technologii vyžadující speciální zacházení, vede také ke změně zacházení se smazanými daty.

V kapitole 1 jsou popsány základy technologie SSD, na který navazuje popis interních rutin manipulujících s daty a shrnutí praktických aspektů. Následně jsou v kapitole 2 popsány vybrané experimenty následované celkovým závěrem v kapitole 3.

1 Technologické základy

Základ SSD tvoří FLASH paměť, typicky v uspořádání NAND, doplněná o kontroler, který řídí operace čtení a zápisu, a také další prvky sloužící k optimalizaci rychlosti zápisu, redukci chyb apod.

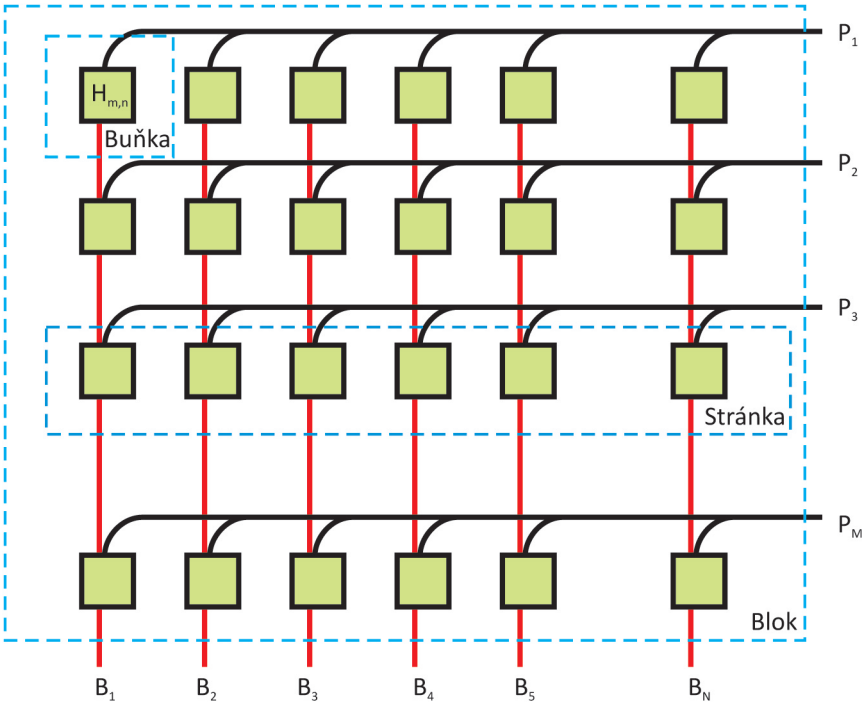
1.1 NAND paměť

Nejmenšími paměťovými jednotkami jsou tzv. paměťové buňky, které jsou tvořeny tranzistory s plovoucím hradlem (floating gate), ve kterém je možné trvale uchovávat náboj [1]. Přítomnost náboje následně ovlivňuje chování tranzistoru, čím větší náboj je přítomen, tím větší napětí V_r je potřeba přivést, aby buňka byla průchozí. Ve výsledku tedy lze rozlišovat binární stav 0 (náboj přítomen v takové velikosti, že při $U < U_r$ buňka není průchozí) a 1 (náboj přítomen ve velikosti, kdy jakékoliv $U \geq U_r$ dostačuje k průchodnosti buňky).

Buňky uchovávající jeden bit výše uvedeným způsobem jsou označovány jako SLC (single layer cell). Nicméně lze uvažovat i o kvantování velikosti náboje do více než dvou stavů, tj. testovat více úrovní Ur_i, pak jedna buňka může obsahovat více bitů. Buňky s více úrovněmi testovaného napětí jsou označovány jako MLC (multi layer cell). Pro uložení N bitů do jedné buňky je však potřeba rozlišovat 2^N úrovní, což vyžaduje přesnější a tedy také delší vyhodnocování. V praxi se zatím lze běžně setkat jen s MLC obsahujícími dva bity a začínají se objevovat MLC se třemi bity, marketingově označované jak TLC (triple layer cell). V článku bude pro jednoduchost dále uvažována SLC.

Jednotlivé paměťové buňky jsou uspořádány do matice a následně propojeny vodiči. Standardně jde o zapojení ve schématu NAND, které vyžaduje méně „drátování“ než jiné varianty. Schéma NAND je znázorněno na obrázku 1 – vždy N buněk v každé řádce je zapojeno paralelně (každá řádka je označována jako stránka, anglicky page) a sloupce buněk jsou zapojeny sériově. Všech M řádků pak tvoří celý blok (ang. block) [3]. Typická velikost stránky je $8\,196 \times 8$ buněk a blok obvykle obsahuje 256 stránek.

Při čtení informací z buněk v m-tém řádku je vodič P_m uzemněn a na ostatní vodiče P_i, $i = 1 \ldots M, i \neq m$ je přivedeno čtecí napětí U_r. Přivedením čtecího napětí jsou buňky donuceny vést proud, nezávisle na uloženém náboji. Hodnota



Obr. 1: Schéma bloku NAND paměti

jednotlivých bitů $H_{m,n}, n = 1 \ldots N$ je pak určena průchodností odpovídajících vodičů $B_j, j = 1 \ldots N$. Buňky, které obsahují logickou 0 obsahují náboj a příslušný vodič tedy proud nevede, naopak buňky s logickou 1 náboj neobsahují a příslušný vodič proud vede. Prakticky lze tedy NAND paměť číst pouze po celých stránkách.

Zápis do buněk je také prováděn po celých stránkách, protože fyzicky je realizován přivedením nízkého napětí na vodič P_i a uzemněním příslušných vodičů B_j, čímž se v příslušných buňkách začne hromadit náboj. Tímto způsobem lze tedy v buňce zvětšit náboj, nikoliv jej však zmenšit, to s nízkým napětím není možné. Což ve výsledku znamená, že lze přepisovat pouze 1 na 0.

Protože je potřeba do každé buňky zapisovat opakovaně, existuje proces označovaný jako reset buňky. Vybití náboje však vyžaduje použití výrazně vyššího napětí (v opačné polaritě) a tedy hrozí, že by mohly být ovlivněny také buňky v blízkém okolí přepisované buňky. Proto je resetování buněk vždy prováděno pro celý blok najednou.

Nepříjemným důsledkem opakovaného zápisu do buňky je postupné trvalé zachycování části náboje v polovodičovém okolí plovoucího hradla, což vyžaduje

stále větší napětí potřebné k zapsání náboje do buňky a zápis je také pomalejší. Po určitém počtu zápisů je už napětí potřebné k zápisu natolik vysoké, že je buňka ztracena pro další používání. Vzhledem k tomu, že zápis je prováděn po stránkách a resetování po blocích, je prakticky ztracen celý blok. Současné SLC NAND paměti by měli vydržet cca 100 tisíc cyklů zápis-reset, u MLC jde o cca 5-10 tisíc cyklů [2]. Rozdíl v životnosti souvisí s tolerancí k množství trvale zachyceného náboje, která je u kvantování do dvou úrovní (SLC) být vyšší než u kvantování do většího počtu úrovní (MLC). Po dosažení kritického počtu cyklů se SSD obvykle přepne do režimu read-only, takže data jsou i nadále dostupná.

Co se týká rychlosti jednotlivých operací, tak operace zápisu i čtení jsou relativně rychlé (v řádu desetin milisekundy). Operace resetování je pak v řádu milisekund, tedy o řád pomalejší než zápis nebo čtení.

Klíčovými charakteristikami NAND paměti vycházejícími z použité technologie, které výrazně ovlivňují chování SSD a jsou důvodem pro vytváření speciálních postupů, jsou:

1. čtení dat probíhá po stránkách, jde o rychlou operaci,
2. resetování je možné provádět pouze po blocích, jde o relativně pomalou operaci,
3. zápis dat je prováděn po stránkách, jde o rychlou operaci a
4. počet zápisů do buňky je omezen.

1.2 Software SSD

Pro zvýšení životnosti paměťových bloků mají moderní SSD implementovánu **funkci pro vyrovnávání opotřebení** (Wear Leveling) [4]. Cílem je zajistit, aby všechny paměťové bloky byly opotřebovány co nejvíce rovnoměrně. Například části patřící k MFT (Master File Table), kde si souborový systém uchovává informace o umístění souborů a další metadata, která se mění při každé manipulaci se souborem, by bez dalšího zásahu byly využívány mnohem intenzivněji než ostatní bloky.

Vlastní implementace spočívá ve vložení překladové tabulky (FTL – Flash Translation Layer) mezi interface a vlastní fyzické čipy. Narozdíl od klasického rotačního disku, kde souborový systém (FS, file system) zapisuje přímo do datových bloků disku, SSD nabízí souborovému systému pouze stejné rozhraní, ale vnitřně má možnost přeuspořádat vnitřní odkazy vhodným způsobem a řídit tím počet zápisů do jednotlivých bloků, aniž by se to projevilo na adresaci datových bloků vně SSD. Vyrovnávání opotřebení může být implementováno jak při operaci zápisu výběrem vhodné stránky, tak i jako operace na pozadí, která průběžně vyhodnocuje dynamičnost ukládaných stránek spolu s využitím příslušných bloků, a následně datové bloky vhodně přesouvá.

Vzhledem k používání FTL nemusí být fyzická kapacita SSD shodná s logickou, deklarovanou na rozhraní. Výrobci SSD toho využívají a zpravidla je fyzická kapacita vyšší než logická. Typicky se jedná o cca 15 %, např. 80 GB disk má ve skutečnosti vnitřní kapacitu 96 GB, existují však i SSD u nichž je fyzická kapacita i několikanásobkem kapacity logické. Tímto trikem, který se v agličtině označuje jako **over-provisioning**, lze totiž značně vylepšit chování disku. Zejména se to týká funkce vyrovnávání opotřebení, dále pak resetování bloků ve chvílích nižšího vytížení a místo navíc lze použít i jako náhradu nefunkčních bloků. Také lze uvažovat o uložení paritních dat, dat pro samoopravné kódy apod.

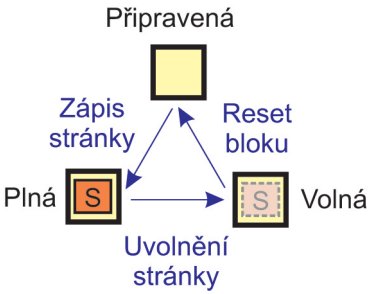
Jak už bylo uvedeno předchozí kapitole, zápis je možno provádět pouze do resetovaných stránek. Obecně se **stránky** na SSD mohou vyskytovat **ve třech různých stavech**:

1. plné – obsahují data, na která je odkazováno z FTL,

2. volné (nezresetované) – obsahují data z minulého zápisu, ale již nejsou používané a

3. zresetované - obsahují samé 1, tj. jsou připravené k zápisu.

Možnosti přechodu mezi těmito stavy jsou znázorněny na obrázku 2.



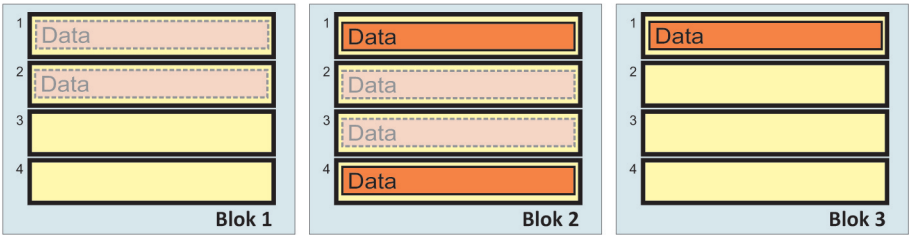
Obr. 2: Možné stavy stránek a přechody mezi nimi

Nový disk obsahuje pouze zresetované bloky a stránky, ale při dalším používání jsou do některých stránek zapsána data, čímž se převedou na plné. K uvolnění stránky dochází ve chvíli, kdy je soubor změněn a data jsou uložena do jiné stránky kvůli vyrovnávání opotřebení. Druhou z možností, jak by mohlo dojít k uvolnění buňky, je smazání souboru. Nicméně disk jako takový nemá žádnou šanci zjistit, že je soubor smazán, protože tato operace je prakticky pouze změna v MFT, souborová data však zůstávají beze změny. Proto byl zaveden

protokol TRIM, kterým může operační systém informovat SSD o uvolněných alokačních jednotkách (clusterech) souborového systému vzniklých smazáním souboru. Některé SSD také obsahující inteligentní podprogram, který je schopný nalézt MFT v uložených datech a následně interpretací zde uložených dat zjistit volné stránky.

Ve chvíli, kdy je stránka označena jako volná, jsou data v ní uložená fyzicky stále přítomna. Při změně dat ve stránce je příslušný odkaz v FTL přesměrován na jinou stránku, takže k původním datům už není možno přistoupit. Při smazání by teoreticky měl odkaz ve FTL na stránku zůstat a data by měla být dostupná až do doby, kdy bude stránka resetována. Některé SSD však mohou pro zdánlivé urychlení mazání odkaz ve FTL zrušit už při označení stránky jako volné. V této situaci jsou data, ač fyzicky stále přítomná, zvnějšku dále nedostupná standardní cestou. U SSD disponujících touto vlastností jsou data prakticky ztracena již v okamžiku smazání, protože při pokusu číst data z příslušného clusteru souborového systému už původní odkaz v FTL neexistuje a je vrácen rovnou prázdný blok.

Změna stránky z volné na resetovanou je mnohem komplikovanější, protože reset musí být prováděn vždy pro celý blok. Aby nedošlo ke ztrátě dat v ostatních stránkách daného bloku, je nutné je přesunout do stránek v jiných blocích nebo počkat, až budou všechny stránky v daném bloku označeny jako volné. Příklad je uveden na obrázku 3), kde jsou tři bloky, přičemž každý obsahuje čtyři stránky. Blok 1 a blok 2 obsahují volné stránky, které by bylo vhodné resetovat, ale zatímco u bloku 1 lze reset provést rovnou, u bloku 2 je potřeba nejprve přesunout stránky 1 a 4 např. do bloku 3.



Obr. 3: Příklad rozložení stránek mezi bloky

Rozhodování, zda se pouštět do resetování bloku obsahujícího ještě nějaké stránky připravené k zápisu nebo stránky plné dat, které je předtím potřeba přesunout, záleží na kontroleru SSD. Každopádně, resetování bloků se vyhnout nelze, jinak by se na SSD dalo zapsat pouze 1×, navíc je záhodno provádět resetování v předstihu, aby nedocházelo k degradaci zápisové rychlosti. Každý SSD tedy obsahuje určitou variantu úklidového podprogramu (**Garbabe Collector**), který zajišťuje dostatečné množství resetovaných stránek.

Za předpokladu, že při smazání nejsou rušeny odkazy FTL, jsou smazaná data ve stránce fyzicky přítomna dokud ji neresetuje garbage collector. Protože se však garbage collector aktivuje podle potřeby SSD a není přímo ovlivněn uživatelem, z vnějšího pohledu to vypadá jako by se data samovolně rozpadala – korodovala. Proto je tento proces někdy označován také jako self-corrosion [4].

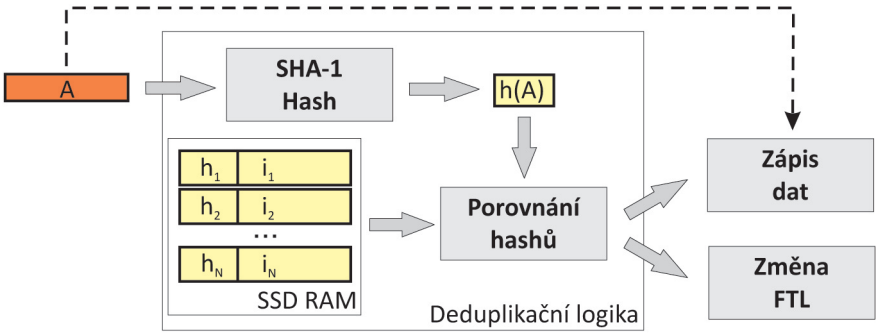
Vzhledem k tomu, že garbage collector potřebuje přesouvat stránky, je z dlouhodobého hlediska objem dat ukládaný souborovým systémem jiný než objem dat, které musí SSD ve skutečnosti reálně zapsat. Přesuny iniciované Garbage Collectorem nebo vyrovnávačem opotřebení spolu s ukládáním servisních údajů způsobuje, že vnitřní objem dat ukládaných SSD je větší. Poměr

$$R_{WA} = \frac{V_{SSD}}{V_{FS}}, \tag{1}$$

kde V_{SSD} je objem dat zapsaných do flash paměti a V_{FS} je objem dat zapsaných souborovým systémem, je označován jako **faktor zesílení zápisu** (Write Amplification Factor – WAF).

Zesílení zápisu nebude tedy z principu nikdy menší než 1, nicméně výrobci SSD se snaží dosáhnout co nejnižší hodnoty, protože čím je vyšší, tím dříve dojde k vyčerpání limitovaného počtu zápisů a tím nižší také bude životnost SSD. Hlavní vliv na zesílení zápisu má způsob zápisu (např. velikost clusterů souborového systému), předzpracování zapisovaných dat (cache SSD) a v neposlední řadě algoritmy vyrovnávání opotřebení a garbage collectoru. Moderní SSD však mohou obsahovat také pokročilé funkce, které ve výsledku umožní efektivně snížit WAF, a to dokonce i pod hodnotu 1.0.

První takovou funkcí je **deduplikace**, běžně využívaná pro snížení objemu uchovávaných dat a úspoře operací zápisu v systémech pro archivaci dat [5]. Díky používání mapovací tabulky (FTL) je SSD již na deduplikaci částečně připraveno. V porovnání s klasickým využitím FTL dochází ke změně mapování logických bloků na fyzické ze schématu $1:1$ na $n:1$. Změna mapovacícho schématu však s sebou přináší také komplikace – zejména složitější fungování garbage collectoru, který při přesunu dat (např. poslední používané stránky v bloku) musí být schopen nyní najít a změnit všechny odkazy v FTL. Obdobně se musí vypořádat s uvolněním pouze části logických stránek sdílejících jednu fyzickou [2]. Vlastní deduplikace je realizována při požadavku na zápis dat, případně může také dodatečně probíhat jako rutina na pozadí. Příslušný postup je ukázán na obrázku 4. Pro zapisovanou stránku A je vygenerován hash $h(A)$ (SHA-1 nebo MD5, přičemž pravděpodobnost kolize je považována za dostatečně malou [2]), který reprezentuje obsah dané stránky. Všechny dosud uložené stránky mají svůj hash h_n, $n = 1, \ldots, N$ uložený v RAM SSD spolu s odkazem i_n na příslušnou stránku. Pokud je nalezen stejný hash, není potřeba příchozí data zapisovat, jelikož již existují, a je tedy vydán pouze pokyn ke změně FTL tak, aby odkaz vedl na odpovídají existující stránku. Opačném případě jsou data vyhodnocena



Obr. 4: Proces deduplikace při zápisu dat

jako nová, zapsána do některé ze zresetovaných stránek, do FTL je uložen odkaz na tuto novou stránku a příslušný hash je přidán do tabulky existujících hashů.

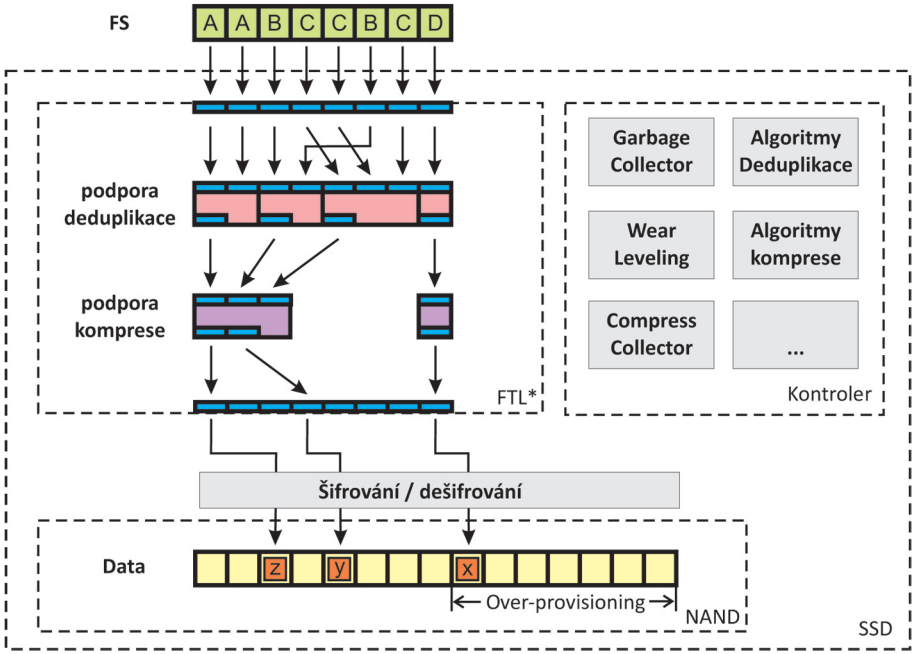
Dále je třeba zmínit **kompresi** dat, která vede ke snížení jejich objemu a tedy k menšímu počtu zápisů. Vzhledem k tomu, že SSD je schopno zapisovat data pouze po celých stránkách a komprese dat v rámci jedné stránky tak nevede k úspoře počtu zápisů, musí být cílem komprimačního algoritmu transformace N vstupních stránek na M výstupních, kde $M \leq N$. Aby vůbec mohl dojít ke kompresi, je nutno zajistit, aby $N > 1$, tj. aby komprimačnímu algoritmu bylo přeloženo naráz více stránek určených k zápisu. Splnění této podmínky je realizováno pomocí vyrovnávací paměti (cache), kde jsou shromažďovány požadavky na zápis dat, takže komprimačnímu algoritmu lze předložit v podstatě stream stránek.

Dále musí být počítáno s budoucími změnami a zajištěno, aby změna v jedné stránce nezpůsobila nutnost načíst, změnit, zkomprimovat a poté znovu uložit příliš velké množství stránek. Řešením je rozdělení vstupního streamu na menší části, které budou samostatně komprimovány. Způsob rozdělení může být různý – od primitivního fixního, kdy je vždy K vstupních stránek komprimováno naráz, přes průběžné adaptivní, které vyhodnocuje výsledek komprese pro konkrétní data na vstupu, až po hledání globálního optima, což může obsahovat i nesekvenční výběr stránek ze vstupního streamu, iterativní prohledávání apod.

V současné době je většina SSD schopna ukládaná data **šifrovat**, čímž šetří systémové prostředky (CPU, paměť). Navíc má pro tuto činnost opět vhodné predispozice – obsahuje vlastní RAM, obsahuje vlastní CPU a uživateli poskytuje pouze zprostředkovaný přístup k datům, takže na pozadí s nimi může libovolně operovat. Ukládaná data jsou před zapsáním do příslušné stránky zašifrována symetrickým klíčem (typicky AES128 nebo AES256 [6]) a při požadavku čtení jsou naopak data načtená ze stránky dešifrována. Je-li nejmenším zapisovatelným blokem část stránky – např. MLC z principu umožňují zapsat každou vrstvu

zvlášť, tj. stránku lze postupně zapsat po částech – je šifrování prováděno na této úrovni. Ostatní vnitřní algoritmy SSD pak totiž mohou zůstat beze změny, změní se pouze zapisovaná data. Sloučením více segmentů při šifrování by docházelo ke zvyšování WAF, protože při změně jednoho segmentu by museli být znovu uloženy i zbývající. SSD používá šifrovací klíč, který si při prvním použití vygeneruje, přičemž jeho použití je chráněno uživatelským heslem.

Zřetězení pokročilých funkcí kontroleru je naznačeno na obrázku 5, přičemž deduplikace musí předcházet kompresi a šifrování (šifrovaná a komprimovaná data jsou prakticky nededuplikovatelná) a šifrování je prováděno až těsně před fyzickým uložením dat.



Obr. 5: Zřetězení pokročilých funkcí v SSD

1.3 Praktické aspekty

Z výše uvedeného vyplývá, že data uložená v SSD jsou

- fragmentována na stránky (typicky 8 kB),
- tyto stránky jsou rozmístěny „náhodně“ v paměťových čipech (díky FTL),

- některé stránky mohou být sdíleny (díky deduplikaci),
- některé uspořádané množiny stránek mohou být dekomprimovány na více stránek a
- navíc mohou být všechny stránky šifrované.

Bez potřebných metadat, tj. FTL, dekomprimačních tabulek a šifrovacího klíče nebo hesla k němu, není možné rekonstruovat data získaná fyzickým přístupem k paměťovým čipům.

Situaci dále komplikuje skutečnost, že SSD neustále provádí operace na pozadí, přičemž k jejich aktivaci stačí připojit napájení. Smazaná data tak mohou bez vnějšího zásahu tak mizet – jakmile byla informace o vymazání zaslána protokolem TRIM, je jen otázkou času, kdy garbage collector daná data smaže. To je poměrně nepříjemná situaci pro doložení konzistence forenzních obrazů, protože dva po sobě sejmuté obrazy se díky aktivite garbage collectoru mohou lišit.

Dále na pozadí také probíhají přesuny na fyzické vrstvě, které nejsou na rozhraní pozorovatelné. Jednak jde o přesuny iniciované garbage collectorem, dále aktivity související s vyrovnáváním opotřebení, optimalizací deduplikace a optimalizací komprese. Všechny tyto aktivity jsou řízeny kontrolerem.

2 Experimenty

Výrobci veřejně neposkytují veškeré informace potřebné k analýze chování SSD, avšak vhodně navržené experimenty mohou vést k identifikaci vlastností kontroleru. Výsledky umožní upřesnit chování SSD k zapsaným a smazaným datům.

Všechny experimenty byly prováděny pouze na logické úrovni, bez přístupu k firmware a bez fyzických zásahů do testovaného disku, prostřednictvím rozhranní SATA. SSD, respektive jeho kontroler, je tedy uzavřený systém o jehož vnitřním fungování nejsou dostupné detailní informace a při testování je považován za blackbox. Uváděné experimenty byly prováděny se SSD Corsair F80 (CSSD-F80GB2-A). Vybrané parametry, důležité pro prováděné testy, jsou uvedeny v tabulce 1.

Při experimentech byl disk připojen rozhraním SATA k počítači s OS Windows 7 v HW konfiguraci CPU Intel Core i7-2600 Processor (8 M Cache, 3.40 GHz), RAM 16 GB DDR3 (4×4 GB 1 333 MHz), integrovaný řadič Serial ATAIII (max. přenosová rychlost 6 GB/s) a HDD 1 TB (7,200 rpm) S-ATA III. Experimenty probíhaly dle potřeby v OS Windows 7 a Linux – Knoppix. Operační systém Window 7 podporuje protokol TRIM, který může být podle potřeby jednotlivých experimentů (de)aktivován.

Tab. 1: Technické parametry SSD disku dle výrobce

Parametr	Hodnota
Kontroler	SandForce
Paměťové moduly	25 nm MLC NAND (2 bity v buňce)
Max. rychlost čtení	280 MB/s
Max. rychlost zápisu	275 MB/s
Podpora TRIM	ano
Self Corrosion	BGC (Background Garbage Collector)
Kapacita NAND	96 GB
Dostupná kapacita	80 GB

Bez zásahu do firmware kontroleru nebo fyzického přístupu k jednotlivým paměťovým modulům lze získat informace o SSD pouze analýzou jeho chování při operacích zápisu a čtení. Pro identifikaci interních paremetrů disku byla zvolena operace zápisu na disk, vzhledem k jednoznačné interpretaci výsledků. Zápis probíhal na prázdný i plný disk. Rozdíly v rychlosti zápisu v závislosti na předchozím stavu disku však nebyly zaznamenány.

Pojem *prázdný disk* popisuje ideální stav, kdy jsou všechny stránky v logické vrstvě (FTL) označeny jako nepoužívané a všechny fyzické bloky jsou zresetované a připravené k zápisu, zatímco *plný disk* odpovídá ideálnímu stavu, kdy jsou všechny stránky v logické vrstvě (FTL) označeny jako použité a do všech odpovídajících fyzických stránek jsou zapsána příslušná data.

2.1 Ověření funkce Garbage Collectoru, FTL a TRIM

Cílem tohoto experimentu byla ověřit aktivitu FTL a TRIM, zejména pak dobu potřebnou k recyklaci odkazů v FTL za různých podmínek. Smazaná data v příslušné stránce zůstávají trvale zachována dokud SSD nedostane informaci o jejich smazání(TRIM, požadavek na zápis do dané stránky) a algoritmus garbage collectoru nerozhodne o resetu bloku s danou stránkou. Z pohledu SATA rozhranní je zjišťováno, jak dlouho trvá protokolu TRIM v součinnosti s FTL přesměrovat odkazy na smazané stránky do `/dev/null`. Doba kdy dojde k resetu příslušné stránky závisí na

- doručení informace o smazání dat v dané stránce (např. TRIM, popř. vlastní analýza MFT),
- vlastní rychlosti resetování spolu s vhodnou dobou spuštění této rutiny a také

- počtu plných stránek v bloku, kde je stránka uložena (u velkého počtu plných stránek v bloku kontroler pravděpodobně nevyhodnotí přesun plných stránek jinam jako vhodný kvůli neopodstatněnému zvyšování WAF).

Testovaná hypotéza č. 1: Není-li doručena informace o smazání souboru protokolem TRIM, zůstanou smazaná data dále dostupná. Zmizí-li data i při vypnutém protokolu TRIM, bude třeba dalším experimentem prověřit schopnost SSD analyzovat uložená data s cílem nalézt MFT a samostatně si odvodit uvolněný adresový prostor.

Testovaná hypotéza č. 2: Nejhorší možná varianta pro získání forenzního obrazu je případ, kdy je rychlost resetování FTL a TRIM vyšší než rychlost vytváření obrazu disku. Je-li tomu tak, získaný obraz bude obsahovat jen minimum smazaných dat nebo dokonce žádné.

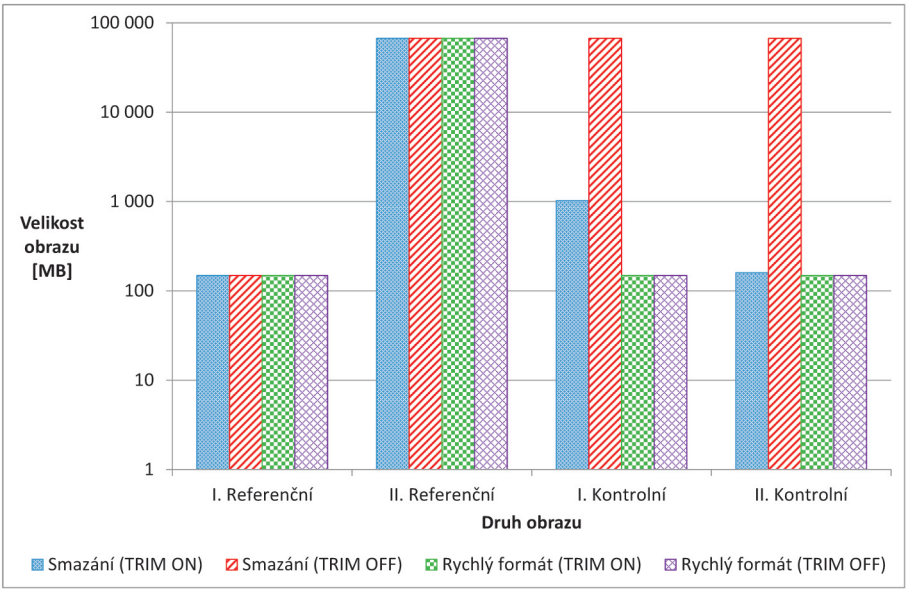
Testovaná hypotéza č. 3: Rychlé formátování je interpretováno jinak než postupné mazání souboru, resp. umožňuje protokolu TRIM specifikovat najednou větší bloky a garbage collector může následně pracovat efektivněji. Pokud je tomu tak, zůstane v obrazu disku sejmutém po mazání více dat než v obrazu disku získaného po rychlém formátování.

Postup experimentu:

1. Připojení prázdného disku
2. Vytvoření referenčního obrazu disku (I. referenční – MFT)
3. Konfigurace operačního systému (zapnutí/vypnutí TRIM)
4. Zkopírování náhodných dat na souborový systém
5. Vytvoření referenčního obrazu disku (II. referenční – MFT + data)
6. Smazání dat / spuštění rychlého formátování.
7. Vytvoření kontrolního obrazu disku (I. kontrolní)
8. Vytvoření kontrolního obrazu disku (II. kontrolní), spuštěn ihned po dokončení I. kontrolního obrazu
9. Porovnání množství zachovaných dat v získaných obrazech.

Uvedený postup byl vyzkoušen ve všech čtyřech naznačených kombinacích (mazání se zapnutým TRIM a mazání s vypnutým TRIM, rychlé formátování se zapnutým TRIM, rychlé formátování s vypnutým TRIM). Velikosti obrazu odpovídají množství dat získaných z disku pouze zhruba, protože obraz je při vytváření komprimován, nicméně nahrávaná data jsou náhodně generovaná, a tedy i minimálně komprimovatelná, takže velikost obrazu lze použít přímo jako měřítko zachování dat.

Naměřené výsledky jsou shrnuty v grafu na obrázku 6. Je zde vidět, že bez aktivace protokolu TRIM zůstavají smazaná data stále přítomna v obou referenčních obrazech (platí hypotéza 1). Při aktivaci protokolu TRIM dochází k fyzickému odstraňování smazaných dat poměrně rychlým tempem, kdy již v druhém obrazu je zachován pouze minimální počet dat (platí hypotéza 2). Rychlé formátování způsobí okamžitou likvidaci všech dat, která se tak nedostanou ani do prvního referenčního obrazu. Vzhledem k tomu, že stejný výsledek je dosažen bez ohledu na aktivaci TRIM, lze předpokládat, že OS, případně ovladač zařízení, použije TRIM bez ohledu jeho na vypnutí. Tuto myšlenku potvrdilo zopakování stejného experimentu s OS Linux, kde smazaná data na disku zůstala i po rychlém formátování. Platnost hypotézy 3 je vázána na konkrétní operační systém.



Obr. 6: Výsledky experimentu

2.2 Rychlost zápisu v závislosti na velikosti bloku

Cílem tohoto experimentu primárně bylo ověřit velikost stránky SSD a možnosti adresování. Znalost těchto údajů odhaluje vnitřní strukturu SSD.

Dle technických parametrů použitých paměťových čipů je velikost stránky 8 kB, přičemž stránka by současně měla být nejmenší adresovatelná jednotka. Nabízí se zde určitá analogie s clustery souborového systému. Pokud je velikost

alokační jednotky souborového systému menší než stránka, dochází k častějšímu zápisu stránek, protože při změně jednoho clusteru je třeba načíst a uložit celou stránku. V důsledku je tedy potřeba častěji resetovat stránky, resp. celé bloky, a při intenzivním zápisu dojde ke zpomalení oproti stavu, kdy je velikost clusteru větší nebo rovna velikosti stránky. SSD také dříve vyčerpá limitovaný počet zápisů a bude dříve zničen.

Testovaná hypotéza 1: Z rychlosti zápisu různě velkých bloků lze odvodit velikost stránky SSD. Postupně bude zvyšována velikosti datového bloku od 1 kB až po 128 MB a měřena doba, za kterou je disk těmito bloky zaplněn. Se zvyšující se velikostí bude klesat doba potřebná k zapsání dat (bude se zvyšovat rychlost zápisu), a to až do doby, kdy bude dosaženo velikosti stránky. Datové bloky větší než velikost stránky jsou zapisovány na více stránek najednou, takže se rychlost zápisu nebude nadále zvyšovat, případně pouze minimálně.

Testovaná hypotéza 2: Lze předpokládat, že SSD může mít implementováno rozdílné zacházení se zapisovanými bloky. Proto byly vyzkoušeny bloky složené pouze bitů obsahujících nuly (zeroes), pouze bitů obsahujících jedničky (ones), ze zcela náhodných bitů (random) a se střídavým opakováním nul a jedniček (Data block „1010“).

Postup experimentu:

1. Připojení plného disku
2. Zaplnění celého disku bloky dané velikosti pomocí nástroje *dd*:

   ```
   dd bs=1k if=/dev/zeroes of=\\?\Device\HarddiskVolume<id>
   ```

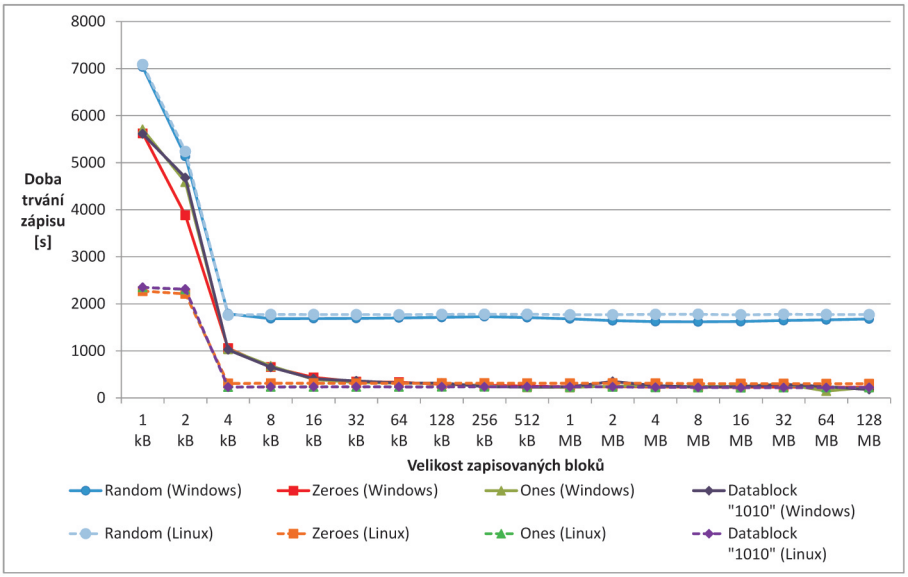
3. Měření doby trvání testu, tj. rozdílu času před spuštěním a po dokončení kopírování.

Kopírování bloků pomocí nástroje *dd* bez použití souborového systému bylo zvoleno kvůli nutnosti specifikovat velikost zapisovaného bloku a eliminaci vlivu vyrovnávací paměti souborového systému. Pro ověření vlivu operačního systému byl experiment proveden na stejném počítači dvakrát, pokaždé s jiným operačním systémem (Window 7 a Linux).

Kromě měření vlastního trvání zápisu na zařízení t_{SSD} byl také pro každý experiment změřen čas t_{null} potřebný k zápisu stejných dat do `/dev/null`. Odečtením těchto dvou údajů je pak získána normalizovaná doba t^*_{SSD}, která je oproštěna od režie operačního systému na generování a předávání dat k zápisu.

Normalizované časy pro jednotlivé provedené experimenty jsou zobrazeny v grafu na obrázku 7. Průběhy experimentů na OS Windows jsou vyneseny plnou čarou a na OS Linux přerušovanou čarou. Barevné označení a tvar bodů na křivce pak odpovídá obsahu bloku (zeroes/ones/random/1010).

Z grafu vyplývá, že doba potřebná pro zápis datových bloků 1 kB a 2 kB je výrazně delší než pro ostatní velikosti. Při zvětšení datového bloku na 4 kB se zápis výrazně zrychlí a touto rychlostí pak probíhá pro jakékoliv větší bloky. Tento



Obr. 7: Doba trvání zápisu pro různé obsahy a velikosti bloků

trend je stejný pro všechny měřené experimenty, nicméně u zápisu zcela náhodných dat (random) je absolutní doba vyšší než u ostatních dat. Nejmenší zapisovatelnou jednotkou testovaného SSD jsou tedy 4 kB (potvrzení hypotézy 1). Tato hodnota koresponduje s předpokládanou velikosti stránky 8 kB při použití dvoubitové MLC, protože každý bit v MLC lze teoreticky zapsat samostatně, tj. po 4 kB. Experiment dokazuje, že dané SSD je schopno této vlastnosti využít.

Dále, rozdílná absolutní doba zápisu pro zcela náhodná data a pro opakující se datové vzory signalizuje, že data jsou před uložením ještě předzpracována (potvrzení hypotézy 2). Zcela náhodná data je nutno uložit v celém objemu, zatímco konstantní data (zeroes/ones) nebo data s nízkou entropií (datové bloky 1010) je možné komprimovat nebo deduplikovat. Tyto procesy lze od sebe na první pohled jen těžko rozlišit, protože jejich výsledný měřitelný efekt – rychlost zápisu – je stejný. Navíc mohou být aktivní oba. Pro přesnější určení je třeba navrhnout další samostatné experimenty.

3 Shrnutí

Solid State Drive se, díky použité technologii, k uloženým a smazaným datům chová jinak než klasické rotační disky, přičemž konkrétní chování závisí na nastavení řadiče (kontroleru) SSD. Zásadním rozdílem je použití FTL a tedy

zprostředkovaný přístup k uloženým datům, který skrývá jejich skutečné fyzické umístění. Optimalizační rutiny, běžící na pozadí, pak manipulují s uloženými daty k docílení optimálního výkonu. Pro forenzní analytiky je špatná zpráva, že smazaná data jsou poměrně rychle (v řádu minut) zničena a tedy nelze počítat s jejich obnovením a následnou analýzou. Na druhou stranu je to pro ně i dobrá zpráva, protože nemusí zpracovávat tolik dat a mohou se soustředit pouze na data nesmazaná.

Z pohledu firem na obnovu dat jde o pěkný zdroj příjmů, protože v závislosti na konkrétním výrobci dosahují SSD poruchovosti od 0,9 % až po 10 %. Vzhledem k používání TRIM je obnova dat snažší v případě hardwarové poruchy, než v případě smazání dat uživatelem. Obnova dat s využitím čtení na fyzické úrovni je možná pouze u SSD, které data nešifrují, protože šifrovací klíč je obsažen v řadiči a nelze jej po havárii disku získat, ani z řadiče vyčíst. Komprese a případná deduplikace obnovu neznemožňují, jen značně komplikují, jelikož po přečtení všech čipů je nutné složit načtená data dohromady ve správném pořadí do celkového obrazu. K tomu je třeba nalézt a dekódovat FTL a reverzovat použité algoritmy.

Literatura

[1] Olson, A. R., Langois, D. J.: Solid State Drives (SSD) Data Reliability and Lifetime, *National Medial Lab White Paper*, April 2008.

[2] Jonghwa, K. et col.: Deduplication in SSDs: Model and Quantitative Analysis, *IEEE 28th Symposium on Mass Storage Systems and Technologies*, MSST 2012, April 16–20, 2012.

[3] Hutchinson, L.: Solid-state revolution: in-depth on how SSDs really work, hardware, http://arstechnica.com/, Jun 2012.

[4] Bell, G. B., Boddington, R.: Solid State Drives: The Beginning of the End for Current Practice in Digital Forensic Recovery?, *The Journal of Forensics, Security and Law*, Vol. 5., Number 3., 2010.

[5] Guo, F., Efstathopoulos, P.: Building a High Performance Deduplication System, *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, 2011.

[6] Visual x laboratories: SSDs with usable built-in hardware-based full disk encryption, http://vxlabs.com, prosinec 2012.

Optimalizace numerických operácií používaných pri šifrovaní

Marek Sýs

E-mail: syso@fi.muni.cz

1 Úvod

Kryptosystémy verejného kľúča (PKC) sú systémy, kde sa vyžaduje verejné šifrovanie pri súkromnom dešifrovaní. Oproti symetrickým šifrám sú tak na PKC systémy kladené vyššie požiadavky, ktoré sa logicky musia niekde prejaviť. Pri PKC sa to prejavuje násobne menším výkonom (pri porovnateľnej bezpečnosti) a preto sa používajú zväčša len na prenos kľúčov pre symetrické šifry. Akokoľvek, v dnešnej dobe už poznáme viaceré jednoduché optimalizácie, ktorými možno niekoľkonásobne zefektívniť ich chod.Z hľadiska optimalizácií je veľkou výhodou PKC systémov, že ich konštrukcia vychádza spravidla z malého množstva „podobných“ algebraických problémov(faktorizácia,diskrétny logaritmus) a je teda postavená na rovnakých operáciach. To má za následok, že ak budeme vedieť optimalizovať jeden PKC systém budeme vedieť optimalizovať aj iný. Otázka je, čo vlastne treba pri PKC optimalizovať? Väčšina PKC systémov ako ElGamal, Diffie-Hellman, RSA, EC má spoločné dve hlavné veci: pracujú s veľkými číslami a používajú modulárnu aritmetiku. Ako možno intuitívne cítiť pri veľkých číslach ako aj pri modulárnej aritmetike je kľučový problém rýchlosť násobenia a násobenie modulo resp. „modulovanie“. Operácie $+, -$ resp. $+\bmod$, $-\bmod$ sú oproti nim zanedbateľné. To, čo možno súčasným poznaním efektívne optimalizovať, možno rozdeliť do troch skupín: násobenie veľkých čísel, násobenie modulo a umocnenie.

2 Veľké čísla a ich násobenie

Veľké čísla sa reprezentujú zvyčajne ako pole (A = A[0],A[1],...) integerov, typicky sú to $(int32, int64)$. Pri implementácii štandardného násobenia veľkých čísel A,B sa vynásobia jednotlivé prvky polí A[i]*B[j] štýlom každý s každým a z výsledkov sa vypočíta výsledná hodnota C=A*B. Tá sa získava z hodnôt

A[i]*B[j] pomocou ich bitového posunu ($\ll, \gg$) a súčtu. Na implementáciu násobenia veľkých čísel, tak treba naiplementovať len súčet resp. prenos takzvaného carry bitu k vyššiemu prvku poľa.

Poznámka Podobne je tomu pri štandardnom sčítaní, kde je potrebné zo súčtu 23+38 preniesť z „jednotiek“ (3+8=2) do „desiatok“ hodnotu 1.

Násobenie veľkých čísel sa teda realizuje štýlom každý s každým po blokoch a teda má zložitosť $O(N^2)$, kde N je počet blokov (veľkosť poľa). Rovnako tomu je pri klasickom násobení, kde každá číslica predstavuje samostatný „blok“. Akékoľvek štandardné násobenie je teda zložitosti $O(N^2)$ vzhľadom na násobenie blokov. V dnešnej dobe poznáme viacero algoritmov, ktoré významne znižujú túto zložitosť. Výhodou týchto algoritmov je, že dokážu pracovať nad blokmi ľubovoľnej dĺžky a tak plne reflektujú reprezentáciu veľkých čísel.

Tieto algoritmy sú postavené na pozorovaní, že pre každý blok čísla C ($C = AB$) totiž nepotrebujeme poznať jednotlivé súčiny $a_i b_j$, ale len nejakú konkrétnu sumu týchto súčinov. Pri týchto algoritmoch sa využíva veľmi jednoduchý princíp – nahradiť konkrétnu sumu súčinov ekvivalentnou s menším počtom násobení. Napr. $a^2 + 2ab + b^2$ sa dá zapísať ako

$$a^2 + 2ab + b^2 = (a + b)(a + b).$$

Konkrétne pre čísla A, B zložené z dvoch blokov $A = a_0, a_1$ a $B = b_0, b_1$ je pre $C = AB$ (reprezentované ako pole troch blokov $C = c_0, c_1, c_2$) potrebné vypočítať tri hodnoty $a_0b_0, a_0b_1 + a_1b_0, a_1b_1$. Pri štandardnom násobení by sme použili 4 násobenia, avšak dá sa to realizovať len pomocou 3. Treba si uvedomiť, že $a_0b_1 + a_1b_0$ sa dá zapísať ako

$$a_0b_1 + a_1b_0 = (a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1$$

a teda sme ušetrili jedno násobenie čísel (blokov). Práve táto konkrétna myšlienka delenia čísla na 2 bloky a nahradenie štyroch násobení troma je základom Karatsubovho násobenia, ktorý možno považovať za prvý z optimalizačných algoritmov násobenia. Ďalšie algoritmy v podstate len zovšeobecňujú túto základnú myšlienku.

Pri optimalizácii možno každý z algoritmov využiť rekurzívne až po blok najmenšej dĺžky a tým získať významné urýchlenie násobenia v každom uzle „stromu násobenia“. Teda napr. pri Karatsubovom násobení súčin $a_i b_j$ blokov a_i, b_j vypočítame opäť tak, že bloky a_i, b_j rozdelíme každý na dva ďalšie bloky a využijeme rovnakú myšlienku atď. Medzi základné algoritmy, ktoré v súčasnosti poznáme, patria nasledovné:

1. Karatsuba násobenie – špeciálny prípad Toom-Cookovho násobenia so zložitosťou $O(N^{\log_2(3)}) = O(N^{1,585})$,

2. Toom-Cook násobenie – veľkosť d delenia bloku určuje zložitosť $O(N^{\log_d(2d-1)})$, d je však pre dané N ohraničené. Pre $d = 3$ je zložitosť $O(N^{1,465})$,

3. FFT násobenie – transformácia najmenších blokov pomocou FFT do frekvenčnej oblasti, súčin vo frekvenčnej oblasti a inverzná FFT transformácia. Zložitosť $O(N \log N)$ pre špeciálne hodnoty $N = 2^k$,

4. Schonhage-Strassen algoritmus – použitie FFT a modulárnej aritmetiky so zložitosťou $O(N \log N \log(\log N))$.

Idey z týchto algoritmov možno použiť aj pri delení čísel, násobení polynómom a matíc. Teda predstavujú univerzálny nástroj na optimalizáciu algebraických operácií. Pri praktickej realizácii je však potrebné počítať s určitou réžiou každého algoritmu. A preto pre malé hodnoty N do 10 000 bitov sú v praxi rýchlejšie jednoduché algoritmy ako Karatsuba a Toom Cook. Konkrétna rýchlosť oboch násobení závisí na implementácii násobenia a použitej platforme.

3 Modulárna aritmetika

Pod pojmom modulárna aritmetika si možno predstaviť základné operácie $+$, $-$, $*$, $/$, ktorých výsledok je potrebné „zmodulovať“ (nájsť kladný zvyšok po delení číslom N) číslom N. Treba pripomenúť, že modulo v niektorých jazykoch (v jazyku C je to operátor %) môže dať aj zaporný výsledok v prípade, že modulujeme zápornú hodnotu. Veľmi dôležitý fakt pri modulárnej aritmetike je, že modulovanie je možné používať priebežne bez dopadu na výsledok. Teda, ak chceme vypočítať $a * b * c \bmod N$ je možné rozdeliť výpočet na viacero krokov

$$a * b * c \bmod N = (a * b \bmod N) * c \bmod N.$$

Priebežným modulovaním získame benefit, že všetky operácie sú realizované s číslami menšími ako je N. Týmto ušetríme pamäť a čas nakoľko veľkosť čísel má priamy dopad na čas každej zo základných operácií. Pozrime sa na to, ako optimálne možno realizovať modulovanie základných operácií. Pre dve čísla A, B menšie ako N možno hodnotu $A + B \bmod N$ získať ako

$$A + B \bmod N = \begin{cases} A + B & \text{pre } A + B < N \quad (1) \\ A + B - N & \text{pre } A + B > N. \quad (2) \end{cases}$$

Obdobne možno vypočítať $A - B \bmod N$ ako $A - B$, resp. $A - B + N$. Je to dané tým, že výsledok operácie mod N si možno predstaviť ako niekoľkonásobné pričítanie/odčítanie hodnoty N, až kým výsledok nepadne do intervalu $\langle 0, N-1 \rangle$. Ako vidíme pre súčet a rozdiel je výsledok rýchly. Horšie to je pri modulárnom násobení $A * B \bmod N$.

4 Modulárne násobenie

Pri modulárnom násobení je možné použiť viaceré optimalizačné techniky, ktoré sú vhodné na použitie v rôznych situáciach. Pri väčšine z nich sa najskôr zrealizuje predvýpočet, ktorý zásadne urýchli samotné násobenie. Tento predvýpočet však môže byť v niektorých prípadoch pomalší ako samotné modulárne násobenie, ale na druhej strane môže urýchliť aj iné násobenia. Pri modulárnom násobení je potrebné rozlíšiť nasledovné prípady:

1. realizujeme jediné násobenie $A * B \bmod N$,
2. realizujeme viaceré násobenia $A_i * B_i \bmod N$ pre rôzne A_i, B_i s rovnakým modulom N.
3. realizujeme veľký počet násobení $A_i * B_i \bmod N$, kde počet čísel A_i, B_i je malý a modulus je opäť rovnaké číslo N.

Pri predpoklade 1. môžeme postupovať klasicky, teda vypočítať súčin a potom modulovať alebo už pri násobení modulovať čiastočné súčty. Vezmime si najskôr prípad, kde máme súčin C a ten následne modulujeme. Súčin C možno získať algoritmami (Karatsuba, ...) z prechádzajúcej časti. Pri modulovaní $C \bmod N$ potom v zásade používame klasický deliaci algoritmus, kde od C postupne odpočítavame násobky N až kým výsledok nie je menší ako N. Odčítavame postupne čísla $2^k N, 2^{k-1} N, \ldots, N$, čo umožňuje využiť rýchly bitový posun. Pri odčítaní testujeme či výsledok nie je záporný. Ak je, vrátime sa k predchádzajúcemu výsledku a pokračujeme v odčítaní ďalšieho čísla. Konkrétne používame pre k-bitové číslo N nasledovný algoritmus:

Input: C, N
Output: $C \bmod N$

$R_0 = C$
$n = 2^k n$
for i = 1 **to** k:
    $R_i = R_{i-1} - n$
    if$(R_i < 0)$ **then** $R_i = R_{i-1}$
    n=n/2
return R_k

Môžeme taktiež využiť násobenie s priebežným modulovaním popísané ako Blakleyho algoritmus. Tu sa však realizuje klasické (pomalé) násobenie spolu s modulovaním. Tento má význam, keď nám nejde o rýchlosť, ale hlavne o úsporu pamäte. Môžeme však taktiež využiť Karatsubovo násobenie s priebežným modulovaním, ktoré je v priemernom prípade len 2 krát pomalšie ako Karatsubovo násobenie [2].

Pri predpoklade z bodu 2. možno postupovať rovnako ako pri 1. Na urýchlenie však možno využiť fakt, že realizujeme viacero násobení. Aj keď modulujeme čísla, ktoré spolu nesúvisia je možné si predvypočítať hodnoty, ktoré možno použiť v každom modulovaní. Ak chceme realizovať $C_0 \bmod N, C_1 \bmod N, \ldots$, má význam si predvypočítať hodnoty $2^{2k} \bmod N, 2^{2k-1}, \ldots, 2^{k+1} \bmod N$. Na výpočet $C \bmod N$ potom stačí spočítať hodnoty $2^i \bmod N$ pre tie i, pre ktoré je i-ty bit jednotkový v bitovom rozvoji čísla C. Výsledkom je číslo menšie ako kN (teda o málo viac bitov ako samotný modulus), ktoré už zmodulujeme veľmi rýchlo. Kľúčový pre algoritmus je bitový rozvoj čísla C, ktorý možno realizovať veľmi rýchlo pomocou bitového posunu jednotlivých prvkov poľa, kde je číslo C uložené.

Montgomeryho násobenie

Pre 3. prepoklad je veľmi efektívne využiť modulárne Montgomeryho násobenie. Táto metóda je obvzlášť efektívna, ak realizujeme modulárne umocnenie. Pri Montgomeryho násobení transformuje modulovanie číslom N do modulovania nami zvoleného čísla R, ktoré je s N nesúdeliteľné. Aby to malo význam je potrebné zvoliť si číslo R tak, aby sme s ním mohli rýchlo modulovať. Optimálne je preto zvoliť si $R = 2^k$, čo umožní realizovať modulovanie jednoduchým vymaskovaním posledných k bitov. Pri Montgomeryho modulárnom násobení postupujeme v troch krokoch:

1. transformácia čísel $A, B \bmod N$ do takzvaného Montgomeryho zvyškového systému $A' = AR \bmod N, B' = BR \bmod N$,

2. výpočet Montgomeryho súčinu(MonPro)

$$u = A'B'R \bmod N = \text{MonPro}\,(A', B'),$$

   kde u je už v Montgomeryho zvyškovom systéme,

3. transformácia u späť do klasického modula.

Kroky 2. a 3. si vyžadujú predpočítanie čísel n, r, pre ktoré platí $nN + rR = 1$. Číslo n sa používa pri výpočte Montgomeryho súčinu. Číslo r sa používa pri transformácii čísla u do klasického zvyškového systému.

Montgomeryho súčin pre získanie $u \bmod N$ z A', B' možno popísať nasledovne:

MonPro(A',B')
Input: A', B', N, R
Output: $u = A'B'RmodN$

$t = A'B'$
$u = (t + (tn \bmod R)N)/R$
if(u < N) **then return** u
else return u-N

Ako možno vidieť, na získanie Montgomeryho súčinu je potrebné realizovať len operácie ako delenie $/Ra \mod R$, ktoré sú však veľmi rýchle, nakoľko R je mocnina dvojky. To znamená, že z hľadiska časovej zložitosti sú kritické kroky 1. a 3. V Montgomeryho násobení sa v 1. kroku realizuje opäť modulovanie $(AR \bmod N)$ a preto jediné modulárne násobenie sa jeho použitím neurýchli. Naopak, ak počítame veľa súčinovmalého počtu čísel je Montgomeryho modulárne násobenie veľmi efektívne. Obzvlášť je efektívne pri modulárnom moumocnení $a^e \bmod N$, kde sa realizujú len dve pomalé operácie. Sú to transformácia a do Montgomeryho zvyškového systému a na konci spätná transformácia výsledku. Všetky operácie vrámci zvyškového systému sú veľmi rýchle a preto tento algoritmus niekoľkonásobne urýchľuje celkový výpočet.

Literatúra

[1] Koc, C. K.: *High-Speed RSA Implementation.* TR 201, RSA Laboratories, 73 p., November 1994. pdf (Also available from RSA Laboratories)

[2] Chow, G. C. T., Eguro, K., Luk, W., Leong, P.: A Karatsuba-Based Montgomery Multiplier. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications* (FPL '10). IEEE Computer Society, Washington, DC, USA, 2010, 434–437.

[3] Bernstein, D. J.: Fast multiplication and its applications. In *Algorithmic number theory: lattices, number fields, curves and cryptography*, edited by Joe Buhler, Peter Stevenhagen. Cambridge University Press, p. 325–384. ISBN 978-0521808545.

[4] Warren, H. S.: Montgomery Multiplication
http://www.hackersdelight.org/MontgomeryMultiplication.pdf

[5] Erdem, S. S., Koç, C. K.: A Less Recursive Variant of Karatsuba-Ofman Algorithm for Multiplying Operands of Size a Power of Two. *IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, 2003, p. 28.

[6] Manochehri, K., Pourmozafari, S., Sadeghian, B.: Montgomery and RNS for RSA Hardware Implementation. *COMPUTING AND INFORMATICS*, North America, 29, jan. 2012.

[7] Jebelean, T.: Practical Integer Division with Karatsuba Complexity, *Proc. ISSAC'97*, 1997, p. 339–341.

Astrofotografie

Petr Švenda

E-mail: svenda@fi.muni.cz

Abstrakt

Poslední dekáda se nesla ve znamení výrazného zlepšení sensitivity snímacích čipů v digitálních fotoaparátech a zároveň jejich velkého rozšíření mezi běžné uživatele díky příjemné ceně. Obyčejný uživatel tak dostal možnost pořizovat snímky nočních astronomických objektů v takové kvalitě, která byla před 20 lety možná jen s velmi nákladnými zařízeními v profesionálních observatořích. Přednáška a článek se vás pokusí přesvědčit, že stojí zato vydržet vzhůru o něco déle, pořídit desítky až stovky snímků a věnovat čas jejich zpracování – vše s cílem zachytit přírodní krásy okem jen stěží viditelné.

Úvod

Přednáška i článek si klade za cíl vám ukázat různé možnosti, které má dnes amatérský fotograf při pořizování snímků objektů denní i noční oblohy. Téma je velice rozsáhlé a jen detailní tutoriál pro jediný typ focení by zabral více než deset stran. Proto raději představíme různé možností, probereme potřebné vybavení a obtížnost digitálního zpracování. Detailnější studium necháme na zvážení zvídavému čtenáři. Nezapomeneme ani na důležité varování nakonec.

Potřebné vybavení

Potřebné vybavení se liší dle typu objektů, které chcete snímat a souvisí především s jasností foceného objektu (nebo objektů). Čím je objekt jasnější (Měsíc, hvězdy, ...) tím je pořizování snímků obecně snažší. Doporučuji tedy začít právě těmito objekty a postupně přecházet na obtížnější cíle (mlhoviny, galaxie, ...).

Pro pořizování pohybu hvězd (tzv. startrails) nebo meteoritický rojů (meteor shower) si vystačíte si se základním, běžně dostupných vybavením:

- Libovolný digitální fotoaparát s možností manuálního ostření a automatické spouště.
- Širokoúhlý objektiv. Při focení hvězdných drah pořizujete typicky širokoúhlé snímky obdobně jako v krajinářské fotografii, abyste zachytili co nejvíce hvězd. Kvalita optiky, její rychlost ani zcela přesné zaostření není příliš podstatná. Některé nežádoucí jevy jako aberace mohou naopak dodat snímku více barev.
- Běžný stativ
- Automatická spoušť. Lze nahradit ovládáním přes notebook (Canon i Nikon nabízejí příslušný nástroj, nebo lze využít např. BackyardEOS).
- Baterie umožňující běh alespoň dvě hodiny, ideální je battery grip nebo přímo napájení ze sítě.
- Ohřev optiky (volitelné) – pokud teplota klesne pod rosný bod, zamlží se vám objektiv. Obranou je jemný ohřev, který si můžete snadno vyrobit doma[1].
- Pokud fotíte v přírodě, nezapomeňte si vzít teplé oblečení a spacák.

Při pořizování fotografií Měsíce se navíc hodí objektiv s delším ohniskem nebo základní astronomický dalekohled. Pokud máte na fotoaparátu funkci živého náhledu (live-view) a notebook, zjednodušíte si pořizování velké série snímků.

Pro pořizování fotografií Slunečních skrvrn se hodí objektiv s delším ohniskem doplněný o jednoduchý solární filtr (speciální stříbrná fólie pohlcující většinu přicházejícího záření) umístěný před čočku objektivu.

Při pořizování fotek hlubokého vesmíru (mlhoviny, galaxie) již budete potřebovat paraktickou montáž, která průběžně kompenzuje rotaci Země a často také různé astronomické filtry, které "ořezávají" vlnové spektrum jen na žádoucí vlnové délky, případně potlačují světlené znečištění. Pro úhlově menší objekty je výhodné použít astronomický dalekohled s co největší světelností. Kvalita optiky začíná být výrazným faktorem ovlivňujícím výsledný snímek.

Při focení emisních mlhovin je výhodné mít upravený filtr před snímačem fotoaparátu, neboť běžný filtr ořezává i vlnové délky patřící emisní mlhovině. Řešením je pořízení fotoaparátu, který je již z výroby opatřen upraveným filtrem nebo tento filtr vůbec nemá (dedikované astrokamery nebo speciální série řadového fotoaparátu např. Canon 60Da[2]) nebo provést v domácích podmínkách vlastní odstranění filtru[3].

[1] http://www.skyandtelescope.com/howto/diy/3304231.html
[2] http://www.milujemefotografii.cz/canon-eos-60da-specialni-zrcadlovka-pro-astrofotografii
[3] http://www.astrolight.cz/Canon400D_IRmod.html

Obr. 1: Různé varianty stativu pro pořizování snímků – fixní stativ (vlevo) a paralaktická montáž německého typu (Vixen GP2 Photoguider, vpravo)

Příprava před focením

Snímat hvězdné objekty lze téměř z libovolného místa, vhodná volba pozice, času a kompozice ale ovlivňuje výsledek u různých tipů focení různou mírou. Zde je krátký seznam seznam tipů před tím, než vyrazíte do terénu:

- Více hvězd bude zachyceno při tmavší a čistější obloze. Pro mapu světelného znečištění navštivte http://www.blue-marble.de/nightlights/2010 nebo http://www.asu.cas.cz/_data/mapa_sv_zn_1236768909.jpg. Obloha je po půlnoci typicky méně světelně znečištěna (některá světla se vypínají) a také létá méně rušivých letadel. Snažte se také fotit směrem od zdrojů světelného znečištění (tj. město za zády).

- Pokud je na obloze Měsíc, dochází k přesvětlení oblohy a budou viditelné jen nejjasnější hvězdy. Fázi a místo východu Měsíce zjistíte pomocí planetária Stellarium (stellarium.org)

- Není vhodné fotit v době, kdy přes hlavní část kompozice přecházejí trhané mraky. Mraky na obloze jsou ve snímku velmi jasné a zastíní tak při výběru maxima pro pixel většinu hvězd. Vzdálená oblačnost na obzoru může být ale naopak efektní. Očekávanou oblačnost lze zjistit na http://medard-online.cz.

- Na místě buďte vždy s předstihem, speciálně pokud fotíte krátkodobé úkazy jako např. zatmění Měsíce nebo Slunce. Nachystání potřebného vybavení vždy zabere netriviální čas, především když něco zkoušíte poprvé.

- Zjistěte si odhad jasnosti objektu. Dejte pozor na rozdíl mezi absolutní, relativní a úhlovou jasností. I objekt s velkou absolutní jasností nemusí být téměř vidět, pokud je hodně vzdálený a zároveň úhlově velký.
- Pro výběr objektů hlubokého vesmíru můžete použít např. službu Google Sky (http://www.google.com/sky/).
- Přepněte do manuálního režimu fotoaparátu. Automatické režimy by vyústily v nežádoucí dílčí expozice s různým nastavením. Dle typu foceného objektu se délka expozice pohybuje od velmi krátké (setiny až tisícíny sekundy, např. Slunce) přes krátké (okolo 1/100 sekundy např. pro Měsíc a planety) a střední (desítky sekund, např. pohyb hvězd nebo metery) až po dlouhé (několik minut, např. mlhoviny nebo galaxie).
- Velká zásobárna znalostí jsou stránky ostatních astrofotografů – jaké objekty fotili, jaké použili dalekohledy, kolik snímků pořizovali, jak je zpracovávali...

Focení pohybu hvězd (Startrails)

Zdánlivý pohyb hvězd vniká v důsledku rotace Země kolem její vlastní osy. Hvězdy vytvářejí soustředné kružnice kolem bodu, který na obloze protíná zemská osa – pro severní polokouli zhruba kolem Polárky. Princip vytvoření efektně vypadajících snímků zachycujících tento pohyb hvězd (anglicky star trails) je jednoduchý.

Pro pořízení snímků lze využít prakticky jakýkoli digitální fotoaparát. Z fixního stativu pořídíte velké množství krátkých separátních expozic. Tyto expozice následně složíte do jediného snímku tak, že pro daný pixel pomocí specializovaného software vyberete nejjasnější hodnotu, která se na dílčích expozicích vyskytla.

Pohyb dostatečně jasné hvězdy je pak v dílčích expozicích vždy tím nejjasnějším bodem a projeví se ve výsledném snímku části kružnice, kterou hvězda stihne opsat. Focení i skládání lze samozřejmě automatizovat.

Detailní popis obsahující tipy pro ostření, nastavení délky expozice a clony, průběh snímání a automatické zpracování naleznete v článku: *Jak fotografovat pohyb hvězd, Petr Švenda, Zoner Blog.*[4]

Focení planet, Měsíce a Slunce (Planetary)

Pokoušeli jste se už někdy o zachycení Měsíce na fotografii a výsledek neodpovídal vaší představě? Například proto, že při větším zvětšení jste dostali mírně

[4] http://www.milujemefotografii.cz/jak-fotografovat-pohyb-hvezd

Obr. 2: Posun hvězd nad jižním obzorem a kostelem G. Santiniho ve Křtinách

rozmazaný a lehce zašuměný snímek, sice s viditelnými známými detaily, ale bez potřebné ostrosti a kontrastu. Přitom i se základním vybavením lze pořídit snímky, které takové nejsou.

Základním trikem, kterým dosáhnete potřebné kvality při focení Měsíce, je pořízení většího množství téměř stejných snímků. Těmi při následném skládání v počítači potlačíte šum a rovněž vám dovolí provádět razantní doostření a saturaci barev. Budete moci prohlížet nejen detaily jako lávová pole, pohoří, ostré hrany a vyvržený materiál z impaktních kráterů ale dokonce i zbarvení hornin v různých částech našeho souputníka.

Pořízení snímků i jejich složení lze automatizovat, takže postup není o mnoho složitější, než pro jediný snímek. Pokud máte navíc v terénu k dispozici notebook a foťák s podporou live-view, stovky snímků pořídíte velice snadno za několik minut. Postup s pořízením videa přes live-view si ale ukážeme až příště, nyní použijeme sekvenci snímků bez videa.

Celý proces se sestává ze čtyř kroků:

1. Naplánování vhodného času a místa focení.

2. Pořízení velkého množství separátních snímků (příp. nahrání videa).

3. Složení separátních snímků pro potlačení šumu a jiných defektů.

4. Upravení složeného snímku ve vhodném fotoeditoru.

Detailní popis obsahující tipy pro ostření, nastavení délky expozice a clony, průběh snímání a automatické zpracování naleznete ve dvou článcích *Jak fotografovat Měsíc II a III, Petr Švenda, Zoner Blog.*[5] První z článků ukazuje postup založený na pořízení desítek až stovek jednotlivých snímku a jejich složení. Druhý postup využívá funkci živého náhledu (live-view) a průběžné snímání náhledu ve velkém rozlišení do notebooku.

Obr. 3: Dorůstající Měsíc po saturaci barev. Výsledný snímek byl vytvořen jako složenina z 900 fotografií pořízených za necelé dvě minuty. Snímáno při 5× zvětšení ve výřezu o velikosti 944 × 632 pixelů rychlostí zhruba dvaceti snímků za sekundu

[5] http://www.milujemefotografii.cz/jak-fotografovat-mesic-ii, http://www.milujemefotografii.cz/jak-fotografovat-mesic-iii

Focení meteoritických rojů (Meteor shower)

V průběhu roku Země zažívá několik významných meteorických rojů, při kterých prolétá proudem drobných částic pozůstalých po některé kometě. Asi nejznámějším je roj Perseidy pocházející od komety 109P/Swift-Tuttle.

Focení meteoritů je podobné, jako focení pohybu hvězd. Použijte co nejširší ohnisko, zvolte 20–30 sekundovou expozici, ISO 800-1600 a zkontrolujte, zda máte oblohu na jednotlivém snímku dostatečně tmavou, aby vám nepřesvítila dráhu případného meteoru (který letí jen zlomek sekundy). Foťte opakovaně tak dlouho, jak jen můžete (zachycené meteory najdete až zpětně, nelze čekat se stisknutím spouště až nějaký uvidíte). Pokud přes vaši scénu přeletí jasnější meteor, zkontrolujte zda a jak je viditelný na snímku a případně upravte expozici (pokud je obloha příliš světlá nebo meteor příliš slabě zachycen). U meteorů nevadí občasné přecházející mraky, naopak dělají kompozici zajímavější. Naopak hodně ruší dorůstající Měsíc, s tím ale kromě zkrácení expozice nic moc nenaděláte. Do scény zahrňte horizont, nejlépe ten východní. Nejvíce meteorů létá typicky po půlnoci, kdy je vaše pozice otočena ve směru pohybu Země.

Roje[6] mají svá maxima, ale mohou být velmi dobře pozorovatelné i několik dnů/týdnů mimo toto konkrétní datum. Pokud provedete registraci hvězd (např. pomocí nástroje IRIS[7]), můžete dokonce odhalit místo na obloze, ze kterého meteory patřící do daného roje „jakoby“ vyletují.

Obr. 4: Jeden snímek s jasným meteorem z celkem 540 pořízených expozic po 30 vteřinách, ISO 1600. Canon 500D a Sigma 10–20mm@10mm f/4

[6] Seznam meteorických rojů naleznete na http://galaxie.web2001.cz/planety/meteority.html.
[7] http://www.astrosurf.com/buil/us/iris/iris.htm

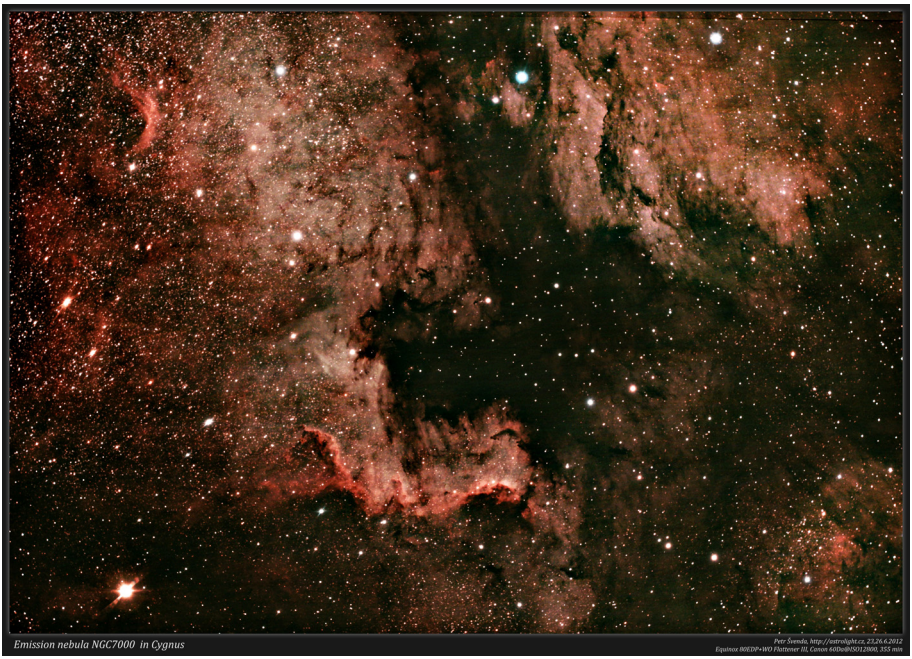
Obr. 5: Galaxie M81 a M82 ve Velkém vozu. Složenina 255 expozic (celkem 8 hodin), ISO 3200. Canon 400Da (modifikovaný), optika Equinox 80EDP 500mm/f6.25

Focení objektů hlubokého vesmíru (Deep-sky)

Focení objektů hlubokého vesmíru je z uvedených možností typicky nejkomplikovanější. Focené objekty jsou typicky velice slabé a ani na snímku trvajícím několik minut nejsou vidět potřebné detaily. Základním trikem je pořízení velkého množství snímků (není výjimkou ini několik desítek hodin kombinovaného expozičního času na jediném objektu) a následné složení snímků v počítači. Na rozdíl od planetární fotografie se ale snímky neprůměrují, ale sčítají tak, aby se ve výsledném snímku projevil každý zachycený foton. Pořizování dlouhých expozic (minuty) vyžaduje paraktickou montáž kompenzující rotaci Země. Čím světelnější je objektiv, tím lépe. Využívající se také různé druhy pomocných kalibračních snímků (darks, flats), které dále zlepšují kvalitu výsledného snímku a snižují všudypřítomný šum.

Velmi detailní popis postupu focení objektů hlubokého vesmíru naleznete na Saratoga Skies[8].

[8] http://www.saratogaskies.com/articles/cookbook/index.html



Obr. 6: Emisní mlhovina NGC7000 v Labuti. Složenina 350 expozic po jedné minutě (celkem 6 hodin), ISO 12800. Canon 60Da, optika Equinox 80EDP 500mm/f6.25

Focení komet

Focení komet se z velké části podobá focení objektů hlubokého vesmíru, neboť většina komet je vizuálně málo jasná. Okem viditelné komety přicházejí nepravidelně jen jednou za více než deset let. Zároveň je však situace specifická v tom, že samotný objekt se pohybuje a proto se typicky registruje nadvakrát. První registrace na hvězdy nám vrátí snímek hvězdného pozadí s mírně protaženou kometou (během focení se stihla posunout). Druhá registrace je na jádro komety a vrátí nám pěknou kometu, ale pozadí s posunem hvězd. Oba snímky se následně skládájí do jediného.

Závěr

Focení nočních objektů je velmi uklidňující a zároveň často velmi flustrující činnost. Uklidnění pod tichou noční oblohou trvá typicky jen do chvíle, než se

poprvé vybijí baterie, zamlží optika nebo přijdou nečekané mraky. Přesto je celá činnost nebezpečně návyková. Začít lze snadným focením pohybu hvězd, ke kterému není potřeba žádné dodatečné speciální vybavení. Nebohý astrofotograf je ale neodolatelně váben k čím dál tvrdším technikám, u kterých tráví čím dál více času a investuje do nich čím dál více peněz. Nakonec nakonec se mu rozpadá manželství a je považován okolím za beznadějného asociála, který z domu vylézá pouze v noci. Astrofotografovi to ale již v tuto dobu nevadí – ví, že honba za vesmírnými fotony vyžaduje velké oběti a že šum ve výsledném obraze klesá nejvýše s druhou odmocninou z počtu pořízených snímků. Nemůže si tedy dovolit propásnout žádnou bezmračnou noc.

The BitLocker schema with a view towards Windows 8

Dan Rosendorf

E-mail: dan.rosendorf@i.cz

Abstract

We give a summary of the BitLocker encryption schema as it is implemented in Windows Vista and Windows 7, along with some information on the key management mechanisms and some of the metadata used. We then point out a sparsely documented change to BitLocker in Windows 8, which is the removal of the Elephant Diffuser and look at some of the consequences of that decision.

1 Introduction

As an inherent part of the newer distributions of Microsoft Windows BitLocker is becoming more widely used by both individuals and corporations. As such it is ever more important to understand what exactly BitLocker does and how it works. While the general algorithm used has been documented in [2] along with an explanation behind the design choices made, the actual implementation details have not been made publicly available by Microsoft. There have been reverse engineering efforts made to try and uncover at least the basics of the structures of key storage and headers for BitLocker volumes to facilitate both restoration of corrupted volumes and mounting normal volumes outside the MS Windows platform. In this article we will give an overview of the results that people have achieved through these means.

We will first give a brief overview of the BitLocker encryption scheme as documented in [2] . In this part we will also give a brief explanation of the various modes that BitLocker can be employed in. We will then give a basic overview of the headers and key storage mechanisms of BitLocker encrypted volumes as described in [3] and as further derived from the source code and outputs from the utility for mounting BitLocker encrypted volumes written by Romain Coltel [1]. We will conclude with a look at BitLocker as it works in

Win8 and an evil maid attack that can be employed against it. We should note that this does not harm the main use of BitLocker as prevention against data misuse in case of straight out theft of a computer.

2 Overview of the BitLocker Schema

The BitLocker schema was originally described by Neils Fergusson in [2]. We will only present a summary here, readers interested in a more detailed treatment should consider reading the original which is publicly available. In the following we shall use block to refer to the block of the AES cipher which is 16 bytes long.

The main cryptographic workhorse in BitLocker is AES in CBC mode. The CBC mode spans a sector of the hard drive which seems to still usually be 512 bytes on most computers though it could be 4 096 or even as much as 8 192 bytes. The length of the sector has some effects on the security of the method, especially in the Win 8 implementation. The second part of BitLocker is the Elephant diffuser. This is a new algorithm which was developed to address some possible security issues that AES–CBC has when applied to disk encryption.

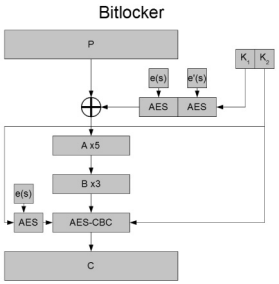


Fig. 1: BitLocker encryption schema

An overview of the encryption process is presented in Fig. 1. The whole encryption is keyed with two keys each of length either 256 or 128 bytes, depending on the setting. The full key is always of size 512 bytes for easier key management. We will think of the key as $K_1||K_2$ where K_1 will be the part of the key that is used to generate the sector key (essentially keying the Elephant diffuser) and K_2 is the key used for the AES–CBC. It is important to note that these two keys are independent according to the documentation, though unfortunately we do not know how key generation is performed. We will write $E(K, P)$ for the encryption of plaintext P with AES using the key K.

The Elephant diffuser part of the encryption has three steps. First the plaintext P is XORred with a sector key K_{sec} which is generated in two steps:

1. First we create $K_s = E(K_1, e(s))||E(K_1, e'(s))$
2. Second we concatenate K_s to itself enough times to generate a string the same length as the sector size. (i.e. if we have a 512 byte sector $K_{sec} = \underbrace{K_s||\cdots||K_s}_{\times 8}$.)

In the above s is the byte offset of the sector, $e(s) = s||\mathbf{0}_8$ and $e'(s)$ is the same as $e(s)$ except the last byte has value 128.

The resulting $P \oplus K_{sec}$ is then further processed with two diffusers A and B. We will omit the exact description of A and B since it is somewhat long and is not essential for our purposes. The goal of A and B is to properly spread information throughout the plaintext. This is particularly important in the decryption direction as the AES-CBC takes care of most of our diffusion needs in the encryption direction.

Once the Elephant diffuser has been applied to the plaintext it is then encrypted with AES in CBC mode. The IV for the AES-CBC is $IV = E(K_2, e(s))$, and the key used is K_2. It should be noted that the IV's for different sectors are thus fixed for the lifetime of a given key. This in practice means they do not change for a given disk unless a reinstallation is performed. The only other way to change them would be to completely decrypt the BitLocker protected volume and reencrypt it under a different key.

3 Usage

There are three main ways to use BitLocker for encryption:

1. Encrypting operating system drives
2. Encrypting data drives
3. Encrypting removable drives (Microsoft calls this BitLocker To Go, but the essentials seem to be the same)

The differences have mostly to do with the possible ways of keys or passwords are input and obtained for the encryption. Operating system drives have in general the highest possible security threshold as it is possible to employ essentially three authentication mechanisms in concert for standard use. These are a TPM (Trusted platform module), a key on a usb drive and a pin to authenticate to the TPM. For encrypting data drives it is possible to use a certificate stored on a smart card, a password, or a usb flash drive with a key. For encrypting removable drives it is also possible to use a smart card with certificate or a password. If the operating system drive is encrypted it is also possible to configure what is

called auto unlock for any data drives or removable drives. This method stores the keys necessary for unlocking the drives in the registry. Before we can explain how this actually works we will give some extra background on how BitLocker deals with keys in general.

4 Keys

BitLocker uses a number of different keys. We have already talked about the two keys used for the actual encryption using AES and keying the diffuser. These two keys together make up a 512 bit key called the FVEK or Full Volume Encryption Key. Note that this key is always of length 512 bits even if the chosen algorithm is AES 128 and thus the sector key used is also 128 bits. The remaining 256 bits are filled with 0's, each of the two keys getting it's own zero padding. The FVEK does not normally leave the encrypted volume and is never stored anywhere in an unencrypted form with the exception of being in memory while it is being used. It is always stored on the disk encrypted with the VMK or Volume Master Key.

Let's look at the general schema of BitLocker keys. In the above we see that any BitLocker encrypted volume will have at least two distinct keys associated with it a FVEK and a VMK. This is not enough though as we still need something to encrypt the VMK with. We now come to the concept of key protectors. Every BitLocker encrypted volume will have one or more key protectors associated with it. These are all actually just structures which hold the VMK encrypted by some third key which BitLocker essentially has to get in clear form. The possible key protectors are

1. **Recovery password:** This is a 48 decimal digit password with 128 bits of real entropy. The digits are calculated from a randomly generated 128 bits with extra info added for checksums. Each 6 digit value represents 16 bits of entropy.

2. **Recovery key:** This is a 256 bit key that can be used for recovery same as the recovery password. Except that it is not possible to type it in. Instead you have to provide a USB flash drive that contains it.

3. **Startup key:** This is a 256 bit key that can be used to unlock the volume in the standard way during startup. Again it has to be loaded from a USB flash disk. There seems to be little difference between it and the recovery except for usage patterns.

4. **TPM key:** This is a (probably) 256 bit key that is stored in the TPM hardware. This should only be released assuming the correct boot process

has been followed and there have been no modifications to the hardware and/or some parts of the OS.

5. **Certificate:** It is possible to add a certificate protector. The inherent assumption is that this certificate be stored on a smart card in some predefined manner. **This has not been tested by the author successfully.**

6. **Password:** This is just a regular password from which a key is generated in some unknown (to us anyhow) way. It does not seem to have any particular restriction except that, if it is being used for OS drive encryption, it is prudent to not use non-US characters or keyboard layouts as the partial environment will not support keyboard mapping.

The way BitLocker handles key protectors is, for the most part, quite straightforward. When you add a new key protector, either by generation when first setting up the volume or by some management tool, such as for example the manage-bde command, BitLocker uses the newly generated or input key to encrypt the VMK and adds this encrypted version, along with some attributes such as the id of the key used for encryption, to the unencrypted metadata where it stores all its key protectors and volume information. So far this is quite unsurprising as something along these lines has to be the case if we wish to unlock the volume with more then one method. Mr. Kornblum in [3] points out one peculiarity of BitLocker though. Along with storing the VMK encrypted by our new key that we wish to use to unlock the volume, BitLocker also encrypts the new key using the VMK and stores the result. While this is not particularly bad it does mean that whoever has legitimate access to the disk gains access to the key of everyone else who has legitimate access.

The key being stored is encrypted by the storage key in AES–CCM mode. The IV for the mode is formed by using a timestamp of when the container was created and a monotonically increasing number of how manyth key container this is. It is not obvious whether this starts from 0 and it seems to go up by more then one per container.

5 BitLocker headers

Each volume encrypted with BitLocker has a 512 byte header which starts with the bytes `2D 46 56 45 2D 46 53 2D` at offset 3 which is `-FVE-FS-` in ascii. This is followed by some information about the volume including the sector size, which seems to be 512 in most cases, GUID, version of BitLocker used and offsets of the 3 BitLocker metadata. The version number seems to be 1 for Windows Vista and 2 for both Windows 7 and Windows 8. The 3 metadata copies are just redundancies in case of corruption. We do not know how BitLocker actually

chooses which of the metadata to use if they differ. The metadata do contain a crc32 checksum though so it is possible to check for some basic tampering.

Each of the metadata itself starts on a sector boundary and with the same `-FVE-FS-` bytes as can be found at offset 3 in the volume header. Further each of the metadata contains the offsets of all the other metadata and itself. The rest of the metadata is made up of key containers. Each of the key containers carries along with the encrypted key also information about the algorithm it is used for. For the FVEK these can be the four combinations of AES 128, AES 256 and active or inactive elephant diffuser. The fact that the keys are encrypted using AES–CCM means that we get an integrity check of the keys. It is important to point out that there are a number of fields that still have unknown uses.

6 TPM

The TPM is a hardware chip built into many modern computers whose main stated function is to provide a chain of trust mechanism for booting into a trusted environment. As the computer boots, the TPM creates hashes of states of the computer at defined stages and then uses them to authenticate the boot process. BitLocker uses the TPM to store part of a key used for the decryption of the VMK. It is necessary to realize that the TPM does not actually provide protection unless it is used in conjunction with another method of authentication, either a pin for the TPM or a USB key. It does however provide the possibility of protecting the key used for encryption in hardware.

The main point of the TPM which is to establish a chain of trust extending from it to the booted OS (i.e. that no one has intentionally tampered with the computer). This is the same chain of trust we see for example in Apple's iPhones and iPads that allows for the very tight controls on what you can do with the device. It is precisely this chain that needs to be broken to allow a user to perform a jailbreak of the device and install software not sanctioned by Apple. In the case of Apple's products the main reason of the chain of trust as implemented is probbably not security as such, but more the desire to keep the platform contained and monopolize the distribution of content to it. There is a fair amount of people that believe this is in reality also the purpose of the TPM, but at this time no computers we know of are sold with TPM's that cannot be turned off. Most computers are actually sold with the TPMs disabled in the default set up.

One of the important current uses of TPMs is as a method of thwarting so called "evil maid" attacks. These are attacks where a third party has multiple, usually short term, opportunities to access the computer. It is so named because the most common situation when a third party can gain such access is when a computer is left in a hotel room which allows hotel staff to access it particularly

maids. It should be noted that this is by no means the only situation in which this attack can be mounted. Repair shops, cleaning and maintenance staff, coworkers and family members are just a sample of people who will generally be able to try and mount an evil maid attack.

The main drawback of the way the TPM works is that it can be quite often oversensitive and the reasons for not realeasing the key are not given to the user. In practice this can mean that for example placing a laptop into a docking station can cause the TPM to not release the key. Other similar types of issues might be BIOS firmware upgrades or changes to the BIOS configuration. If this kind of issue crops up more often, there is a very high chance the user gets used to the TPM occasionally not authenticating the computer will most probably assume that's what happened even once an attacker actually goes and maliciously tampers with the computer. For some more information on how to compromise a computer with BitLocker using a TPM we recommend [4].

7 Changes to BitLocker in Win 8

In Windows 8 a somewhat surprising and rather under-publicized change has been made to BitLocker. The choice of using Elephant diffuser has been removed and it is now always off. It is important to realize that this means that the sector key which used to be XORred into the plaintext has also been removed. Thus BitLocker in Windows 8 has devolved into just AES in CBC mode. Without the sector key and diffuser the cipher text becomes a lot less bound to the sector it is in then before. While CBC makes sure that in the encryption direction small changes still propagate reasonably well. Changes to the cipher text only propagate at most one block away, and moreover moving a cipher text to a different sector, only destroys the first 16 bytes of the corresponding plaintext. We should note that somewhat surprisingly in this case a larger sector size (i.e. 4 096 or 8 192) is detrimental, as it allows less plain text destruction when swapping full sectors.

7.1 Possible attacks

It is important to remember that AES–CBC, which is what BitLocker is in Win 8, is quite safe against attempts to decrypt the cipher text on its own. The problem is not with the attacker being able to decrypt the cipher text when he gains access to it, but rather his ability to make targeted changes to it. There are in essence two types of attack that can be mounted against the BitLocker implementation present in Win 8 which follow from the removal of the Elephant diffuser:

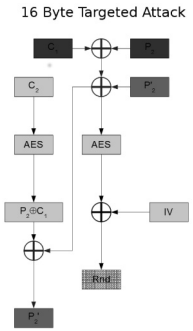


Fig. 2: Targeted 16 byte attack on CBC

The first one, which we already mentioned above is moving a sector of cipher text. Since the decryption of a block of cipher text only depends on its contents and the contents of the block immediately preceding it, or in the case of the first block the IV, if we move a sector of cipher text to a different location it will decrypt to almost exactly its original plain text with the exception of the first block (16 bytes) which will end up being random as it gets XORred with a different IV. This can have quite significant impacts, if we are able to choose the sectors correctly, including compromising other security mechanisms such as antivirus software or firewalls. Another possible application might be to move sensitive data from a part of the disc where it is protected by some security mechanisms to a different part where it can be read.

The second attack is, consists of making specific changes to particular blocks of the plain text by using the well known bit flipping attack on CBC. The main idea of the attack can be summarized in Fig. 2. If we know the original plaintext for a given block, we can change the block to plain text of our choice by XORring our chosen new plain text along with the original plaintext into the preceding cipher text block. This does randomize the preceding plain text block. The main application of this attack would again be to disable some security critical pieces of code that would then allow the attacker to compromise the computer, or possibly even compromise the computer directly by inserting well thought out malicious code into for example some standard library.

We do note that both of these attacks assume that the attacker has knowledge of the layout of data on the targeted disk. While it is not always possible to gain this information, in some cases such as for example imaged computers or computers in corporate environments which are usually installed in a very specific manner, this is quite feasible.

8 Conclusion

Let us once more mention that the attacks and weaknesses presented in this paper, while possibly serious, do not hamper the ability of BitLocker as implemented in Windows 8, to protect data against what Microsoft mainly tries to prevent, which is recovery by an attacker after a one time theft. The attacks presented here need a more sophisticated attacker that is trying to gain data in a targeted manner. Even so, the exclusion of Elephant Diffuser from BitLocker in Windows 8, has left the schema significantly weakened. Looking back to Niels Fergussons paper [2] and his definition of a successful attack on page 9 we can see that in Windows 8 such attacks are very easy to mount.

References

[1] Coltel, R.: Hsc tools dislocker, March 2013.

[2] Ferguson, N.: Aes-cbc + elephant diffuser: A disk encryption algorithm for windows vista, 2006.

[3] Kornblum, J. D.: Implementing bitlocker drive encryption for forensic analysis. *Digital Investigation*, 5(3):75–84, March 2009.

[4] Türpe, S., Poller, A., Steffan, J., Stotz, J. P.: Attacking the bitlocker boot process, 2009.

Může šifrování dat na přenosných počítačích ochránit EU proti průmyslové špionáži?

Martin Kákona

E-mail: martin.kakona@i.cz

Abstrakt

Stále více produktivní lidé v Evropě pro svou práci používají notebooky, což donedávna bylo výsadou manažerů. Zároveň díky velkým kapacitám disku může dnes obchodník nebo manažer mít na svém notebooku technické/obchodní podrobnosti dříve nevídaného rozsahu, protože destilace dat je dražší než náklady na přenosná úložiště. V praxi to ovšem znamená, že se stále více citlivých dat objevuje na lokálních discích notebooků. Jsou to data bezprostředně související s technologií, tedy s největším obchodním artiklem „starého kontinentu“. Přitom ochrana přenosného počítače je v mnoha směrech problematická, protože hrozí jeho krádež, snadný odposlech, nemůže být pod trvalým dozorem a hrozí jeho modifikace...

Dá se předpokládat, že právě datová úložiště přenosných počítačů se mohou stát snadným cílem průmyslové špionáže. Ukážeme si zde některé možné útoky proti kryptografii, která je použita na ochranu lokálních dat. Budeme zde diskutovat pouze metody špionáže/hrozby, které bezprostředně souvisí s kryptografickou ochranou lokálně uložených dat na přenosných počítačích.

V tomto článku se zaměříme na praktické příklady v prostředí notebooků a operačních systémů MS Windows. Většinu zde uvedených problémů lze však uvažovat i v prostředí jiných operačních systémů a jiných přenosných zařízení.

Způsoby kryptografické ochrany lokálně uložených dat

V dalším textu budeme používat označení CT pro šifrový text a OT pro otevřený text. Jako IV označujeme iniciační vektor. Předpokládáme, že čtenář zná základní módy blokové šifry: CBC, CFB, OFB, ... [1].

Šifrování dat uložených v notebooku na lokálním úložišti (HDD/SSD) můžeme rozdělit do třech skupin:

1. Off-line šifrování souborů.
2. On-line šifrování souborů.
3. Full Disk Encryption

Off-line šifrování souborů

Jde o šifrování, kdy je vždy atomicky zašifrován/dešifrován celý soubor. Před zašifrováním souboru je vždy vygenerován nový IV nebo nový SEED. IV je dostatečně náhodný, aby byla zanedbatelná pravděpodobnost, že se použije shodný IV pro stejné nebo podobné OT. SEED by měl být vždy jedinečný pro každou operaci šifrování. Výsledkem dešifrování je většinou soubor s OT.

Typickým příkladem off-line šifrování je například komprese souborů doplněná o šifrování (například soubory s příponou .zip).

Off-line šifrování je velmi odolné proti útokům na CT. Pokud je správně aplikován mód šifry, zejména pokud je věnována zvláštní pozornost poslednímu bloku v módu CBC, a pokud je dostatečně dimenzován šifrovací klíč, jedná se o způsob šifrování souborů, který je odolný proti všem dnes známým útokům.

Velkou praktickou nevýhodou off-line šifrování je, že šifrování musí být implementováno přímo v aplikaci. V opačném případě totiž musí po dešifrování vzniknout na disku počítače soubor s OT, který je velmi těžké z disku beze stop odstranit. Poznamenejme, že i v případě, že aplikace načítá OT přímo do paměti, musí se jednat o nestránkovanou paměť nebo musí být v operačním systému zakázáno vytváření stránkovacího souboru anebo musí být stránkovací soubor šifrován nějakou z dalších metod.

Velkou výhodou off-line šifrování je, že můžeme snadno na CT aplikovat kryptografickou kontrolu integrity.

On-line šifrování souborů

Dešifrování souboru v on-line režimu se provádí výhradně do operační paměti počítače. Nevzniká tedy na disku pracovní kopie OT, jak je to běžné u off-line šifrování. Na druhou stranu, aby bylo možné se šifrovaným souborem efektivně pracovat, musí použitý mód šifry umožňovat změnu velmi malé části OT (typicky jeden byte) bez nutnosti přešifrovat celý soubor.

Praktickým příkladem on-line šifrování souborů je šifrování na úrovni souborového systému, například EFS (nadstavba NTFS).

On-line šifrování souborů je náchylné k vyzařování sémantické informace vzhledem k tomu, že modifikace OT má za následek modifikaci pouze části CT [2].

Full Disk Encryption

Dnešní praktické implementace FDE provádějí šifrování celého disku jedna ku jedné. To znamená, že velikost CT je stejná jako velikost OT. Šifrování se provádí po blocích délky stejné nebo menší, než je velikost sektoru na disku.

Typickým příkladem FDE jsou SW systémy TruCrypt a BitLocker nebo HDD standardu Opal.

Společnou nevýhodou předchozích výše zmíněných druhů šifrování (off-line a on-line šifrování souborů) je zbytková informace, která vzniká v OS a aplikacích při přístupu k OT (dočasné soubory, stránkovací soubor, soubory s náhledy (thumbnails), ...). Pomocí FDE mohou být šifrována jak vlastní data souborů, tak zbytkové informace bez závislosti na jejich umístění na disku. Nevýhody FDE jsou podobné jako u on-line šifrování, je vyzařována sémantická informace na úrovni bloků módu šifry. Další nevýhodou je absence kontroly integrity [3].

Vybrané Hrozby

Krátký klíč

Za dostatečnou délku klíče pro symetrickou kryptografii se do konce roku 2030 považuje 112 bit [4]. Obecně se však doporučuje (s ohledem na nejistý vývoj v oblasti kvantových počítačů) minimální délka klíče 128 bit. Pokud použijeme písmena anglické abecedy (bez ohledu na velikost (case sensitivity)) a číslice, vyvarujeme se při tom znakům „o“ a „0“, „l“ a „1“, „y“ a „z“ kvůli záměně a různým národním klávesnicím, musí mít klíč (passphrase) délku přibližně 26 znaků. Přičemž nesmí existovat závislost mezi jednotlivými znaky klíče, takže klíč nesmí obsahovat slova, slabiky a podobně. Takový klíč je pro člověka nezapamatovatelný. Je tedy nezbytné, přenášet klíč pomocí nějakého technického zařízení; např. čipové karty nebo USB Flash Drive.

Nyní výše zmíněnou informaci porovnejte s tím, že uživatel používá prostředek TrueCrypt pro šifrování systémového disku počítače. TrueCrypt používá klíč dlouhý 256 bit, což odpovídá 53 znakům podle výše zmíněných pravidel. TrueCrypt přitom neumožňuje pro šifrování systémového disku zadat klíč jiným způsobem než z klávesnice. To je možná důvod, proč zpravodajské služby mají rády notebooky šifrované TrueCryptem. Přitom v ostatních ohledech je tento prostředek velmi dobrý, používá standardní algoritmus AES s délkou klíče více než dostatečnou, jeho kód je zveřejněn a je širokou programátorskou a kryptologickou obcí zkoumán, takže implementační problémy jsou málo pravděpodobné.

Evil Maid

Útok předpokládá, že uživatel zanechá notebook bez dozoru na hotelovém pokoji. Útočník (hotelový personál nebo člověk vydávající sebe za hotelový personál) nainstaluje na počítač škodlivý software (například tak, že zmodifikuje zavaděč operačního systému) za účelem získat kontrolu nad počítačem. Škodlivý software pak může například uložit na určené místo na disku citlivé informace nebo odeslat tyto informace po síti a podobně.

Tento útok je zvlášť nebezpečný, pokud se může uskutečnit opakovaně. Například při prvním útoku se zmodifikuje zavaděč a při druhém útoku se přečte heslo z konkrétního místa na disku, kam ho uložil zmodifikovaný zavaděč. Potřebný kód pro tento útok přitom není složitý, pouze stačí zobrazit obrazovku, která vypadá stejně, jako obrazovka, kam uživatel zadává běžně heslo. Zadání hesla může klidně skončit chybou a software se pak může sám smazat. Pak už jen stačí notebook uživateli odcizit.

Proti takovémuto útoku by měla uživatele ochránit technologie Secure Boot. Ovšem pouze za předpokladu, že Evil Maid nedokáže zmodifikovat firmware počítače, který prakticky Secure Boot prosazuje. Diskuze kolem Secure Boot je na samostatný článek. Navíc autor tohoto článku se necítí být nestranný, neboť trpí přesvědčením, že technologie Secure Boot vznikla za účelem ochrany trhu některých firem sdružených v TCG, ne za účelem ochrany dat uživatelů. Pokud by měl být Secure Boot ochranou proti škodlivému software, musel by ho prosazovat přímo hardware počítače, tedy procesor.

Modifikace CT bez ztráty integrity

Předpokládáme, že útočník má informaci o rozložení OT na disku. Například si obstaral image, který se používá pro instalaci nových počítačů v dané firmě nebo si koupil stejný počítač, jako má oběť, s předinstalovanou OEM verzí operačního systému.

Útočník zmodifikuje CT na určitém místě. Modifikace CT má za následek poškození jednoho bloku OT, které není systémem detekováno. Modifikace má za následek cílenou změnu OT a následně změnu chování OS nebo vybrané aplikace.

Na takovýto útok je zvláště náchylné FDE. Technika potřebná pro provedení takového útoku je popsána v [3].

Vrácení OT do původního stavu

Útočník se dostane k CT a uloží si ho pro další použití. Následně po změnách v OT se útočník dostane k zařízení a vymění nový CT za původní CT (částečně nebo v celém rozsahu).

Tento útok lze provést prakticky na všech běžných systémech FDE, protože z výkonnostních a kapacitních důvodů nemají implementovánu kontrolu integrity.

Přečtení obsahu dynamických pamětí

Útočník se dostane k notebooku, který je v režimu spánku nebo se dostane k notebooku velmi brzo po jeho vypnutí. Přečte obsah dynamické paměti a získá tak zejména šifrovací klíče. Tento útok byl již na EurOpen diskutován [5].

Modifikace BIOS/UEFI počítače

Útočník zmodifikuje firmware počítače tak, aby zachytával hesla/klíče/případně jiné citlivé informace nebo aby upravil chování OS, například selektivně vyřadil firewall. Taková modifikace je možná ve výrobě počítače nebo v obchodním řetězci, proto by se měl v počítači před zahájením jeho používání vyměnit firmware. Ovšem za předpokladu, že firmware, který stáhneme z webu výrobce, nemá v sobě zakomponován backdoor. Je třeba si uvědomit, že firmware počítače je dnes velice komplexní záležitost, která má v sobě implementovány drivery pro obsluhu většiny zařízení v počítači a také například síťové komunikační protokoly. Kromě BIOSu/UEFI jsou v počítači rozšiřující adaptéry (například síťový adaptér), které mohou obsahovat vlastní firmware, pevné disky jistě také obsahují firmware a podobně. Přitom většina periférií dnes provádí přístup k operační paměti pomocí DMA, tedy bez vědomí hlavního procesoru/procesorů, tedy bez přímé kontroly operačním systémem.

Modifikace firmware počítače je samozřejmě vážnou hrozbou pro Secure Boot a Measured Boot.

Vyzařování sémantické informace

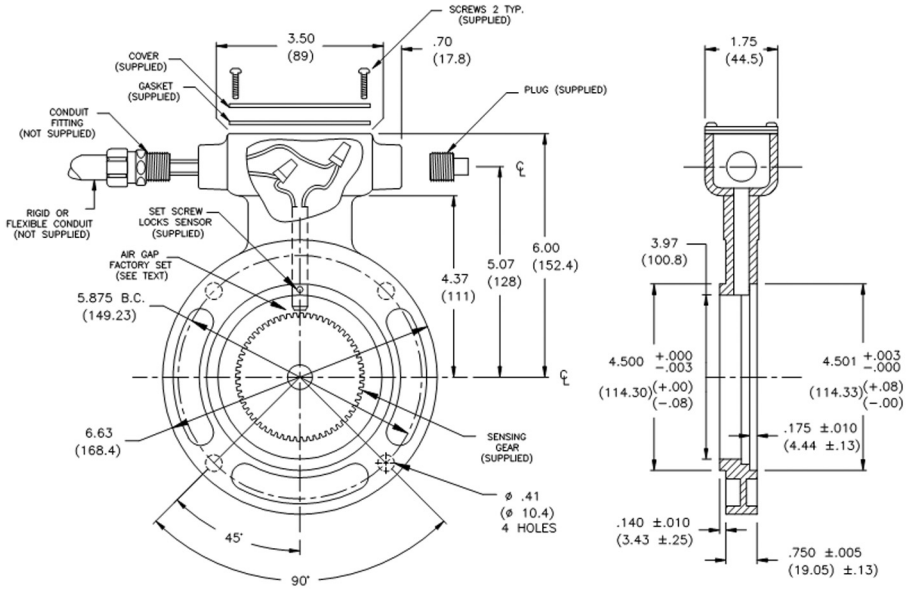
V předešlém textu jsem se již zmínil o možném vyzařování sémantické informace. Považuji za nutné, popsat tuto hrozbu podrobněji. Abych snáze vysvětlil, co myslím pod pojmem sémantická bezpečnost dat, ukáži to na příkladu.

Dále v textu uvedený příklad je vymyšlený a jakákoli podobnost s realitou je tedy čistě náhodná ;-)

Představte si tuto situaci:

V jednom z výrobních závodů nejmenované firmy se vyrábí magnetorezistivní snímač polohy hřídele motoru. Jeho výkres je na Obr. 1. Stáhnul jsem ho pro vás z Internetu, abyste viděli, jak to zařízení vypadá.

Konkurence šlape firmě na paty a vyrábí téměř totožný díl. Konstruktéra napadlo, jak zefektivnit výrobu a vydal se tedy na služební cestu do výrobního závodu, aby dohodl podrobnosti. Stáhl si výkres a uložil ho na notebook, který



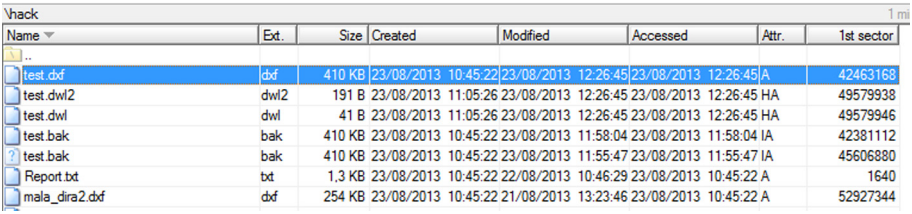
Obr. 1: Výkres magnetického snímače

má disk zašifrován v módu XTS-AES. Protože nechce mít problémy s kompatibilitou CAD systémů, uložil si ho ve formátu .DXF (to jsem ho nechal udělat, protože tento formát je dokumentovaný [6] a nechtělo se mi zabývat drobnými nuancemi formátu .DWG, které nejsou dobře popsány ;-).

Už v minulosti se stalo, že konkurence záhadným způsobem zjistila tajné technologické podrobnosti a vyrábí konkurenční snímače za neuvěřitelně nízkou cenu, kterou si management firmy nedovede vysvětlit. Už jenom když se sečtou „overheady“, tak jim cena vychází vyšší a to se ještě musejí pokrýt náklady na výrobu! Proto odborníci z interního IT nasadili na notebooky konstruktérů diskové šifrování a donutili je, aby si pamatovali náhodné desetiznakové heslo. (Jenom na okraj, polovina konstruktérů má heslo uloženo v mobilu, protože nechtějí být za blbce a volat z druhého konce Evropy, že ho zapomněli. Také poznamenejme, že délka klíče 10 znaků odpovídá přibližně 50 bitům. V roce 1999 trvalo stroji EFF DES cracker, zjistit klíč o délce 56 bitů, 22 hodin a 15 minut.)

Než konstruktér odjel, tak jeho pracovní adresář na notebooku vypadal jako na Obr. 2.

Náš výkres se jmenuje test.dxf. Je vidět, že kromě tohoto souboru jsou na disku ještě soubory test.bak a ještě jeden smazaný test.bak. Ty vznikly při pravidelném ukládání souboru, než konstruktér vytelefonoval s logistikou, aby mu



\hack

Name	Ext.	Size	Created	Modified	Accessed	Attr.	1st sector
..							
test.dxf	dxf	410 KB	23/08/2013 10:45:22	23/08/2013 12:26:45	23/08/2013 12:26:45	A	42463168
test.dwl2	dwl2	191 B	23/08/2013 11:05:26	23/08/2013 12:26:45	23/08/2013 12:26:45	HA	49579938
test.dwl	dwl	41 B	23/08/2013 11:05:26	23/08/2013 12:26:45	23/08/2013 12:26:45	HA	49579946
test.bak	bak	410 KB	23/08/2013 10:45:22	23/08/2013 11:58:04	23/08/2013 11:58:04	IA	42381112
test.bak	bak	410 KB	23/08/2013 10:45:22	23/08/2013 11:55:47	23/08/2013 11:55:47	IA	45606880
Report.txt	txt	1,3 KB	23/08/2013 10:45:22	22/08/2013 10:46:29	23/08/2013 10:45:22	A	1640
mala_dira2.dxf	dxf	254 KB	23/08/2013 10:45:22	21/08/2013 13:23:46	23/08/2013 10:45:22	A	52927344

Obr. 2: Stav adresáře s výkresem před jednáním

sehnali ubytování blízko továrny. Tyto soubory mají stejný obsah, liší se pouze časovými údaji. Například se do souboru ukládá kumulativní čas, jak dlouho se se souborem pracovalo.

Konstruktér notebook s výkresem zahibernoval, uložil do tašky a odjel.

Do hotelu se konstruktér dostal až večer, dal tedy notebook na pokoj a šel do restaurace na večeři.

Co nikdo netušil, že pro konkurenci pracuje státní rozvědka nejmenované mocnosti. (Kdyby to kontrarozvědka věděla, tak by určitě naší firmu ve státním zájmu informovala ;-)

Výzvědná služba nejmenované mocnosti si za pomoci Evil Maid pořídila během večeře snapshot disku z notebooku na hotelovém pokoji.

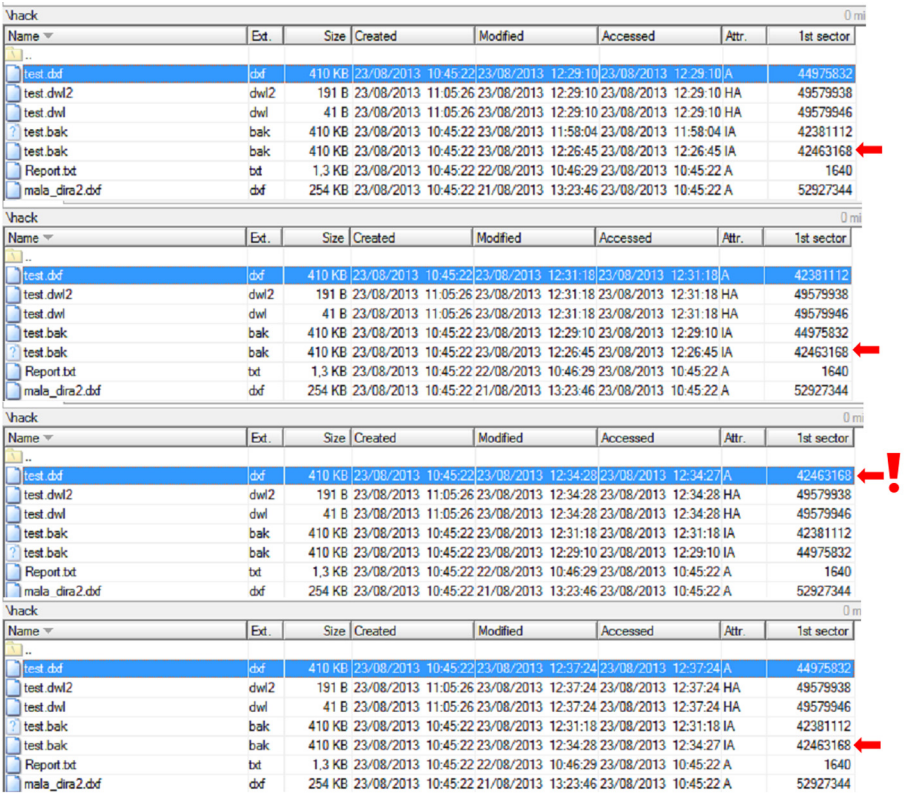
Co asi tak mohli ze zašifrovaného disku zjistit? Téměř nic. Všechna data jsou zašifrovaná včetně hibernačního souboru a počítač byl už dlouho vypnut, takže v operační paměti nic nezbylo. Možná snad jen polohu prázdného místa na disku. Notebook má totiž SSD s funkcí TRIM v módu DZAT.

Ráno konstruktér odejel z hotelu do výrobního závodu. K obědu měl pizzu, kterou na poradu nechal přivézt šéf výroby, protože lidé kolem ISO 9001 měli stále nějaké připomínky, přestože hlavní technolog neviděl žádný problém. Asi se jim nechtělo měnit směrnice.

Po poradě jel konstruktér zase do hotelu, protože mu letadlo odlétalo až druhý den ráno. (Zapomněl jsem napsat, že výrobní závod je v té části Evropy, kde lidská práce ještě není tak drahá.) Večer šel konstruktér zase na večeři do restaurace a hádáte správně, že si Evil Maid opět pořídila snapshot disku.

Teď už na tom rozvědka byla lépe. AutoCAD totiž pracuje tak, že když ukládáte soubor, tak nejdříve smaže soubor .BAK, potom přejmenuje soubor .DXF na .BAK a do souboru .DXF zapíše nová data. To je celkem normální a racionální postup, jak nepřijít o data. Co je ale možná překvapivé, že soubory jsou jen málo fragmentované nebo vůbec (pokud je dostatek místa na disku), a že pokud nově zapisovaný soubor má velikost stejnou jako právě smazaný soubor, tak ho souborový systém uloží s velkou pravděpodobností do uvolněného místa po smazaném souboru .BAK. Prakticky to znamená, že pokud v AutoCADu

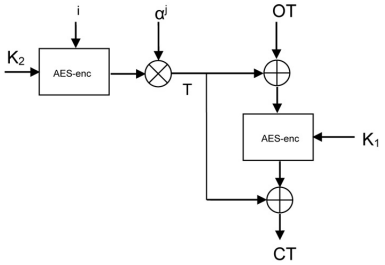
zapisujete stále soubor stejné délky, tak se postupně střídá jeho umístění tak, jak je to vidět na Obr. 3. Porovnejte ještě s původním stavem na Obr. 2. Samozřejmě to platí pouze v případě, že příliš mnoho jiných programů zrovna ve stejném čase nezapisuje soubory podobné délky. Nicméně, je vidět, že zápis stejných dat na stejné místo disku není nepravděpodobný. V našem konkrétním případě spíše zákonitý.



Obr. 3: Poloha souborů s výkresem na disku po 1., 2., 3. a 4. zápisu (shora dolů)

Jak to vypadá se změnami souboru .DXF? Pokud jde například o délku, tak ta se prakticky nemění. Museli bychom do něj dokreslit nové prvky (čáry, kóty, popisky, . . .) nebo něco smazat. Pokud uděláme pouze drobné změny, například posuneme čáru, délka souboru zůstane stejná, protože se pouze změní parametry čáry. Navíc popis konkrétní čáry zůstane ve změněném souboru na stejném místě.

Nyní se podíváme, jak funguje mód šifry XTS-AES [7]. Je to znázorněno na Obr. 4, kde



Obr. 4: Kryptografické schéma XTS-AES

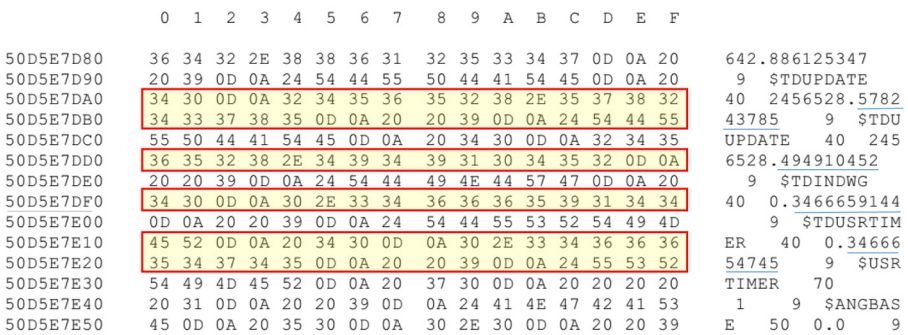
K_1 a K_2	jsou šifrovací klíče,
i	je pořadové číslo sektoru na disku,
j	je pořadové číslo 16 bytového bloku v rámci sektoru,
OT	je otevřený text v délce 16 bytů,
CT	je šifrový text v délce 16 bytů,
α^j	je vektor, který má různou hodnotu v závislosti na j,
T	je tweak, který má různou hodnotu pro každý 16 bytový vektor na disku a tato hodnota je závislá na klíči.

Z uvedeného vyplývá, že každých 16 po sobě jdoucích bytů na disku je šifrováno jinak a nezávisle na ostatních bytech. Kdybychom tedy věděli, kde na disku leží soubor s naším výkresem, mohli bychom porovnáním šifrového textu z prvního snapshotu s textem z druhého snapshotu zjistit, co se na výkresu změnilo, protože víme, že AutoCAD uloží nový zmodifikovaný soubor s velkou pravděpodobností na stejné místo na disku, kde byl uložen minulý záložní soubor. Sice nemůžeme zjistit konkrétní podrobnosti změny, ale věděli bychom, čeho se změna týkala.

Jak ale zjistíme, kde se v šifrovém textu na disku o velikosti stovek GB nachází náš soubor?

Je to velmi jednoduché. DXF soubor má na offsetech 0xDA5, 0xDCE, 0xDF5 a 0xE1A časová razítka, která se změní při každém uložení souboru [6] (viz Obr. 5). Vzhledem k tomu, že snapshoty nejsou od sebe pořízeny déle jak jeden den, hledáme pouze rozdíly v nejméně významných číslech. Hledáme tedy dva po sobě jdoucí sektory na disku, které mají změněny právě a pouze bloky dat o délce 16 bytů na offsetech 0xDA0, 0xDB0, 0xDD0, 0xDF0, 0xE10 a 0xE20 od začátku alokačního bloku. Kdyby to náhodou nestačilo, tak AutoCAD dělá v souboru i další změny na fixních místech, které jsou dostatečně daleko od sebe, aby byly rozeznatelné. Také můžeme hledat změny v rohovém razítku výkresu a podobně.

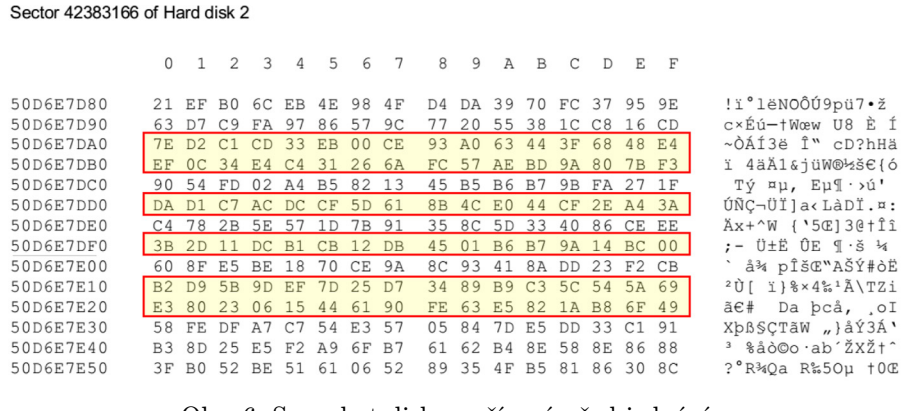
Porovnáním snapshotů a prohledáním výsledku porovnání na výše uvedený vzor tedy zjistíme umístění souborů typu .DXF, které se od minulého snapshotu změnily. Jenom poznamenejme, že pokud bychom výkresy upravovali v jiném



```
              0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

50D5E7D80    36 34 32 2E 38 38 36 31  32 35 33 34 37 0D 0A 20   642.886125347
50D5E7D90    20 39 0D 0A 24 54 44 55  50 44 41 54 45 0D 0A 20    9  $TDUPDATE
50D5E7DA0    34 30 0D 0A 32 34 35 36  35 32 38 2E 35 37 38 32   40  2456528.5782
50D5E7DB0    34 33 37 38 35 0D 0A 20  20 39 0D 0A 24 54 44 55   43785     9  $TDU
50D5E7DC0    55 50 44 41 54 45 0D 0A  20 34 30 0D 0A 32 34 35   UPDATE    40  245
50D5E7DD0    36 35 32 38 2E 34 39 34  39 31 30 34 35 32 0D 0A   6528.494910452
50D5E7DE0    20 20 39 0D 0A 24 54 44  49 4E 44 57 0D 0A 20       9  $TDINDWG
50D5E7DF0    34 30 0D 0A 30 2E 33 34  36 36 36 35 39 31 34 34   40  0.3466659144
50D5E7E00    0D 0A 20 20 39 0D 0A 24  54 44 55 53 52 54 49 4D       9  $TDUSRTIM
50D5E7E10    45 52 0D 0A 20 34 30 0D  0A 30 2E 33 34 36 36 36   ER   40  0.34666
50D5E7E20    35 34 37 34 35 0D 0A 20  20 39 0D 0A 24 55 53 52   54745     9  $USR
50D5E7E30    54 49 4D 45 52 0D 0A 20  37 30 0D 0A 20 20 20 20   TIMER    70
50D5E7E40    20 31 0D 0A 20 20 39 0D  0A 24 41 4E 47 42 41 53    1     9  $ANGBAS
50D5E7E50    45 0D 0A 20 35 30 0D 0A  30 2E 30 0D 0A 20 20 39   E   50  0.0     9
```

Obr. 5: Časové značky v souboru .DXF

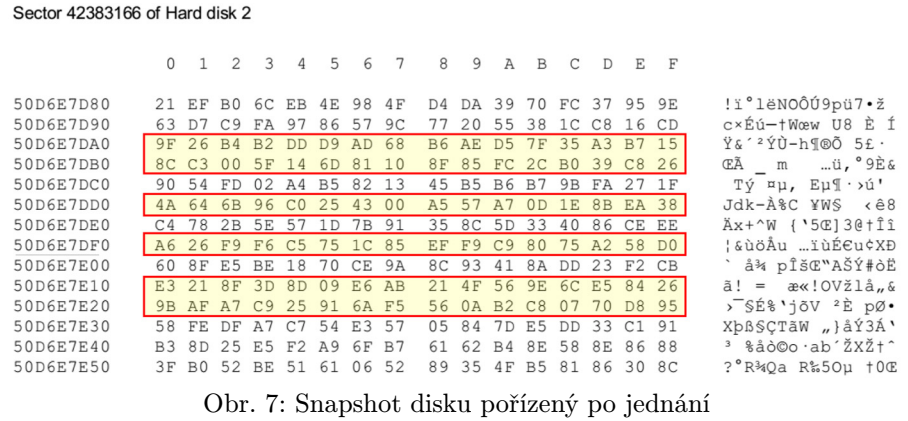


Sector 42383166 of Hard disk 2

```
              0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

50D6E7D80    21 EF B0 6C EB 4E 98 4F  D4 DA 39 70 FC 37 95 9E   !ï°lëNOÔÚ9pü7•ž
50D6E7D90    63 D7 C9 FA 97 86 57 9C  77 20 55 38 1C C8 16 CD   c×Éú—†Wœw U8 È Í
50D6E7DA0    7E D2 C1 CD 33 EB 00 CE  93 A0 63 44 3F 68 48 E4   ~ÒÁÍ3ë Î" cD?hHä
50D6E7DB0    EF 0C 34 E4 C4 31 26 6A  FC 57 AE BD 9A 80 7B F3   ï 4äÄ1&jüW®½šЄ{ó
50D6E7DC0    90 54 FD 02 A4 B5 82 13  45 B5 B6 B7 9B FA 27 1F    Tý ¤µ, Eµ¶·›ú'
50D6E7DD0    DA D1 C7 AC DC CF 5D 61  8B 4C E0 44 CF 2E A4 3A   ÚÑÇ¬ÜÏ]a‹LàDÏ.¤:
50D6E7DE0    C4 78 2B 5E 57 1D 7B 91  35 8C 5D 33 40 86 CE EE   Äx+^W {'5Œ]3@†Îî
50D6E7DF0    3B 2D 11 DC B1 CB 12 DB  45 01 B6 B7 9A 14 BC 00   ;- Ü±Ë ÛE ¶·š ¼
50D6E7E00    60 8F E5 BE 18 70 CE 9A  8C 93 41 8A DD 23 F2 CB   ` å¾ pÎšŒ"AŠÝ#òË
50D6E7E10    B2 D9 5B 9D EF 7D 25 D7  34 89 B9 C3 5C 54 5A 69   ²Ù[ ï}%×4‰¹Ã\TZi
50D6E7E20    E3 80 23 06 15 44 61 90  FE 63 E5 82 1A B8 6F 49   ã€#  Da þcå‚ ¸oI
50D6E7E30    58 FE DF A7 C7 54 E3 57  05 84 7D E5 DD 33 C1 91   XþßŞÇTãW „}åÝ3Á'
50D6E7E40    B3 8D 25 E5 F2 A9 6F B7  61 62 B4 8E 58 8E 86 88   ³ %åò©o·ab´ŽXŽ†ˆ
50D6E7E50    3F B0 52 BE 51 61 06 52  89 35 4F B5 81 86 30 8C   ?°R¾Qa R‰5Oµ †0Œ
```

Obr. 6: Snapshot disku pořízený před jednáním



Sector 42383166 of Hard disk 2

```
              0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

50D6E7D80    21 EF B0 6C EB 4E 98 4F  D4 DA 39 70 FC 37 95 9E   !ï°lëNOÔÚ9pü7•ž
50D6E7D90    63 D7 C9 FA 97 86 57 9C  77 20 55 38 1C C8 16 CD   c×Éú—†Wœw U8 È Í
50D6E7DA0    9F 26 B4 B2 DD D9 AD 68  B6 AE D5 7F 35 A3 B7 15   Ÿ&´²ÝÙ-h¶®Õ 5£·
50D6E7DB0    8C C3 00 5F 14 6D 81 10  8F 85 FC 2C B0 39 C8 26   ŒÃ _ m   …ü,°9È&
50D6E7DC0    90 54 FD 02 A4 B5 82 13  45 B5 B6 B7 9B FA 27 1F    Tý ¤µ, Eµ¶·›ú'
50D6E7DD0    4A 64 6B 96 C0 25 43 00  A5 57 A7 0D 1E 8B EA 38   Jdk–À%C ¥W§  ‹ê8
50D6E7DE0    C4 78 2B 5E 57 1D 7B 91  35 8C 5D 33 40 86 CE EE   Äx+^W {'5Œ]3@†Îî
50D6E7DF0    A6 26 F9 F6 C5 75 1C 85  EF F9 C9 80 75 A2 58 D0   ¦&ùöÅu …ïùÉ€u¢XÐ
50D6E7E00    60 8F E5 BE 18 70 CE 9A  8C 93 41 8A DD 23 F2 CB   ` å¾ pÎšŒ"AŠÝ#òË
50D6E7E10    E3 21 8F 3D 8D 09 E6 AB  21 4F 56 9E 6C E5 84 26   ã! =  æ«!OVžlå„&
50D6E7E20    9B AF A7 C9 25 91 6A F5  56 0A B2 C8 07 70 D8 95   ›¯§É%'jõV ²È pØ•
50D6E7E30    58 FE DF A7 C7 54 E3 57  05 84 7D E5 DD 33 C1 91   XþßŞÇTãW „}åÝ3Á'
50D6E7E40    B3 8D 25 E5 F2 A9 6F B7  61 62 B4 8E 58 8E 86 88   ³ %åò©o·ab´ŽXŽ†ˆ
50D6E7E50    3F B0 52 BE 51 61 06 52  89 35 4F B5 81 86 30 8C   ?°R¾Qa R‰5Oµ †0Œ
```

Obr. 7: Snapshot disku pořízený po jednání

nástroji než AutoCAD, například LibreCAD nebo MS Visio, dostali bychom jiný vzor změn. To, jaký je oblíbený editační nástroj v potrefené firmě, se ale dá zjistit z veřejných souborů, které firma produkuje.

Jak to konkrétně na zašifrovaném disku vypadá, je vidět na Obr. 6 a Obr. 7.

Zbývá ještě zjistit, o jaký výkres při jednání šlo. To je problém. Musíme porovnat potenciální možné výkresy, které máme k dispozici (v rozsáhlé databázi originálních výkresů, které jsme už v minulosti vyšpiónili), se zjištěným vzorem změn. To je skutečný úkol pro výzvědnou službu. Zdá se vám to nemožné nebo přinejmenším příliš obtížné? V tom případě si vyzkoušejte, jestli můžete v nějaké rozvědce nebo kontrarozvědce pracovat. Třeba v britské MI5: https://www.mi5.gov.uk/careers/use-your-it-skills.aspx ;-)

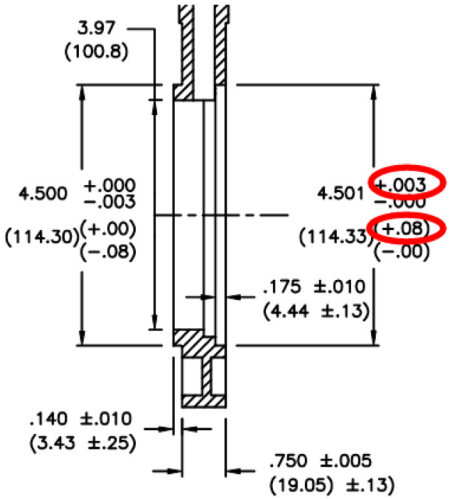
My si tento úkol zjednodušíme, protože tušíme, o jaký výrobek jde, a zkusíme změny porovnat s naším výkresem, který jsem zmiňoval na začátku. Zjistíme, že podezřelé bloky jsou na offsetech 0x4F5F0 a 0x4F730 od začátku souboru (začátek souboru leží 6 sektorů před naším hledaným vzorem z Obr. 5). Na těchto offsetech najdeme kladné tolerance, které patří ke kótě 4.501 palce. Viz Obr. 8.

```
          0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F

0004F580  41 63 44 62 45 6E 74 69  74 79 0D 0A 20 20 38 0D   AcDbEntity    8
0004F590  0A 30 0D 0A 31 30 30 0D  0A 41 63 44 62 54 65 78    0  100  AcDbTex
0004F5A0  74 0D 0A 20 31 30 0D 0A  32 36 2E 36 38 37 35 32   t   10  26.68752
0004F5B0  39 38 36 33 38 37 39 34  35 0D 0A 20 32 30 0D 0A   986387945   20
0004F5C0  34 39 2E 37 36 34 31 35  33 32 31 36 31 31 34 39   49.7641532161149
0004F5D0  36 0D 0A 20 33 30 0D 0A  30 2E 30 0D 0A 20 34 30   6   30  0.0   40
0004F5E0  0D 0A 30 2E 32 0D 0A 20  20 31 0D 0A 2B 2E 30 30     0.2    1  +.00
0004F5F0  33 20 0D 0A 31 30 30 0D  0A 41 63 44 62 54 65 78   3   100  AcDbTex
0004F600  74 0D 0A 20 20 30 0D 0A  54 45 58 54 0D 0A 20 20   t    0  TEXT
0004F610  35 0D 0A 45 44 37 0D 0A  33 33 30 0D 0A 31 37 0D   5  ED7  330  17
0004F620  0A 31 30 30 0D 0A 41 63  44 62 45 6E 74 69 74 79    100  AcDbEntity
0004F630  0D 0A 20 20 38 0D 0A 30  0D 0A 31 30 30 0D 0A 41      8  0  100  A
0004F640  63 44 62 54 65 78 74 0D  0A 20 31 30 0D 0A 32 36   cDbText   10  26
0004F650  2E 36 38 37 35 32 39 38  36 33 38 37 39 34 35 0D   .68752986387945
0004F660  0A 20 32 30 0D 0A 34 38  2E 36 32 32 30 37 39 33    20  48.6220793
0004F670  30 30 32 30 34 35 32 0D  0A 20 33 30 0D 0A 30 2E   0020452   30  0.
0004F680  30 0D 0A 20 34 30 0D 0A  30 2E 32 0D 0A 20 20 31   0   40  0.2    1
0004F690  0D 0A 28 2D 2E 30 30 29  0D 0A 31 30 30 0D 0A 41     (-.00)  100  A
0004F6A0  63 44 62 54 65 78 74 0D  0A 20 20 30 0D 0A 54 45   cDbText    0  TE
0004F6B0  58 54 0D 0A 20 20 35 0D  0A 45 44 38 0D 0A 33 33   XT    5  ED8  33
0004F6C0  30 0D 0A 31 37 0D 0A 31  30 30 0D 0A 41 63 44 62   0  17  100  AcDb
0004F6D0  45 6E 74 69 74 79 0D 0A  20 20 38 0D 0A 30 0D 0A   Entity    8  0
0004F6E0  31 30 30 0D 0A 41 63 44  62 54 65 78 74 0D 0A 20   100  AcDbText
0004F6F0  31 30 0D 0A 32 36 2E 36  38 37 35 32 39 38 36 33   10  26.687529863
0004F700  38 37 39 34 35 0D 0A 20  32 30 0D 0A 34 39 2E 30   87945   20  49.0
0004F710  31 36 39 31 32 30 31 35  30 37 35 33 33 0D 0A 20   1691201507533
0004F720  33 30 0D 0A 30 2E 30 0D  0A 20 34 30 0D 0A 30 2E   30  0.0   40  0.
0004F730  32 0D 0A 20 20 31 0D 0A  28 2B 2E 30 38 29 0D 0A   2    1  (+.08)
0004F740  31 30 30 0D 0A 41 63 44  62 54 65 78 74 0D 0A 20   100  AcDbText
0004F750  20 30 0D 0A 54 45 58 54  0D 0A 20 20 35 0D 0A 45    0  TEXT    5  E
0004F760  44 39 0D 0A 33 33 30 0D  0A 31 37 0D 0A 31 30 30   D9  330  17  100
```

Obr. 8: Porovnání zjištěné pozice změn s výkresem

Změna pravděpodobně patří do tohoto výkresu, protože současně s imperiální tolerancí se změnila odpovídající metrická tolerance. Nalezený vzor souhlasí se vzorem změny kóty. Viz Obr. 9.



Obr. 9: Pozice změn na výkresu

Co říci závěrem našeho příběhu? Rozvědka předala konkurenci, co zjistila, tam se inženýři zamysleli, a jak to dopadlo, nechám na domyšlení laskavému čtenáři.

Tak tohle je ta sémantická bezpečnost.

Možnosti obrany

Projděme si naše možnosti ochrany proti výše zmíněným útokům chronologicky podle životního cyklu počítače. Ve všech případech předpokládáme, že šifrovací mechanizmy, zejména mód šifry, jsou implementovány bezchybně. Za možného útočníka považujeme libovolnou firmu nebo státní instituci, jejíž majoritní kapitál leží mimo Evropu.

Koupíme nový počítač

Q: Můžeme důvěřovat jeho FW?
A: Ne. Než se k nám počítač dostane, existuje mnoho možností, jak firmware vyměnit.

Q: Pomůže výměna FW?
A: Ne vždy. Pokud útočník zmodifikuje Boot Block Flash ROM, může jeho kód infiltrovat nově nahraný FW.

Q: Můžeme důvěřovat HW počítače?
A: Asi ano, pokud byl vyroben v EU.

Q: Známe nějakého evropského výrobce notebooků?
A: Odpovíme otázkou. Znáte firmy Quanta Computer, Compal Electronics, Wistron, . . . ?

Q: Jaké části počítače se vyrábí v EU?
A: Vyrábí se zde některé čipy, například stabilizátory.

Provedeme update BIOS/UEFI počítače

Q: Můžeme důvěřovat FW, který stáhneme z webu výrobce počítače?
A: Nevíme. Možná ano, pokud se jedná o evropského výrobce. Neznáme zdrojový kód firmware. Pouze specifikace UEFI má přes 2 000 stran [8]. FW je tak komplexní, že by nebyl problém v něm schovat prakticky libovolně sofistikovaný backdoor. Přitom většina potřebných funkcí je ve FW již implementována (přístup k diskům, přístup k síti, . . .).

Q: Známe nějakého evropského výrobce BIOSu?
A:

Na počítač nainstalujeme MS Windows

Q: Můžeme věřit tomuto operačnímu systému?
A: Ano, pokud věříme NIST.

Q: Jsou v operačním systému chyby, které může útočník využít?
A: Ano.

Zašifrujeme disk počítače

Q: Může někdo bez znalosti klíče změnit chování OS?
A: Ano, pokud zná rozložení dat na disku.

Na počítači vytvoříme soubory, které jsou chráněny on-line šifrováním

Q: Může útočník zjistit nějakou informaci obsaženou v těchto zašifrovaných souborech bez znalosti klíče?
A: Pokud získá přístup ke zmíněným souborům pouze jednou, tak nemůže.

Na počítač uložíme soubor, který je chráněn off-line šifrováním. Do takovéhoto souboru postupně přidáváme data

Q: Pokud útočník získá všechny záložní kopie takovéhoto šifrovaného souboru, může bez znalosti klíče získat nějaké informace?
A: Pouze změny velikosti souboru v násobcích délky bloku šifry.

Na počítači dešifrujeme soubor, který je chráněn off-line šifrováním

Q: Můžeme z počítače odstranit OT, který vznikne po dešifrování takového souboru?
A: Pokud věříme výrobci disku, můžeme použít funkci SE. Po smazání celého disku touto funkcí by měla být data odstraněna.

Počítač necháme vypnutý na hotelovém pokoji a někdo nám ho ukradne

Q: Jsou data kompromitována?
A: Pokud jste používali dostatečně mohutný klíč, nejsou.

Počítač necháme na hotelovém pokoji a nikdo ho neukradne

Q: Mohu počítač nadále používat ke zpracování citlivých dat?
A: Pokud jsem si jist, že nikdo nezmodifikoval jeho HW a FW, tak ano.

Počítač chceme dát do sběru

Q: Můžeme dát do sběru i disk, pokud jsme používali FDE?
A: Ano, pokud jste používali dostatečně mohutný klíč a prověřenou implementaci šifrovacího algoritmu.

Možná východiska ze současné situace

1. Nemůžeme věřit HW ani FW současných notebooků dostupných na komerčním trhu. Můžeme tedy pouze doufat, že CYBINT [9] neumístí backdoor do všech komerčně vyráběných počítačů. Nebylo by to ani rozumné, protože by to zvyšovalo riziko odhalení takovéto činnosti.

2. Můžeme se pokusit o zpětné inženýrství FW notebooků, které chceme používat. Je nereálné získat zdrojové kódy FW nebo detailní popis HW.

3. Můžeme se pokusit o výrobu HW prostředku v podmínkách EU, který umožní ochránit zavaděč šifrovacího stroje, potažmo zavaděč OS, potažmo šifrovací engine.

4. Nestačí nasadit pouze FDE pro ochranu dat uložených na notebooku. Pomocí běžně dostupného FDE nelze zajistit integritu OS. Následně z toho vyplývají hrozby mající za následek kompromitaci dat.

5. Pro ochranu dat nestačí nasadit pouze šifrování souborů. Pro zdárné šifrování souborů je nutné se vypořádat se zbytkovou informací a se sémantickou bezpečností.

6. Musíme zajistit integritu HW notebooku po celou dobu jeho používání. Například vhodným pečetěním.

Obranné mechanizmy je stále těžší v Evropě realizovat díky tomu, že se výroba počítačů a komponent počítačů v Evropě téměř neprovádí. Je třeba se připravit na to, že jakékoli obranné technické prostředky budou drahé ve srovnání s cenami výpočetní techniky, na které jsme zvyklí.

Závěrem je třeba konstatovat, že EU postupně ztrácí technologické schopnosti bránit se CYBINT.

Literatura

[1] Dworkin, M.: Recommendation for Block Cipher Modes of Operation, *NIST Special Publication*, č. 800-38A, 2001.

[2] Kákona, M.: Extending Security Functions for Windows NT/2000/XP, In *SPI*, Brno, 2003.

[3] Rosendorf, D.: BitLocker attack, In *SPI*, Brno, 2013.

[4] Keylength, [Online]. Available: http://www.keylength.com/. [Přístup získán 2013].

[5] Brož, M.: Šifrování disků (nejen) v Linuxu, In *EurOpen*, klášter Želiv, 2011.

[6] DXF Reference, Autodesk, 2007.

[7] Dworkin, M.: Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices, *NIST Special Publication*, č. 800-38E, 2010.

[8] Unified Extensible Firmware Interface, UEFI Forum, 2013.

[9] List of intelligence gathering disciplines, [Online]. Available: http://en.wikipedia.org/wiki/List_of_intelligence_gathering_disciplines. [Přístup získán 2013].

Šifrování disků a Truecrypt

Milan Brož

E-mail: milan.broz@mail.muni.cz

Abstrakt

Multiplatformní program Truecrypt pro transparentní šifrování disků se za poslední roky stal jedním z nejpoužívanějších nástrojů tohoto typu.

Přestože je zástupcem programů s otevřeným zdrojovým kódem, jeho licence vylučuje zařazení do některých Linuxových distribucí.

Tento problém vedl k nezávislé implementaci Truecrypt kompatibilního formátu v programu cryptsetup. Na základě zkušeností s touto implementací si popíšeme, jak se vyvíjel formát Truecryptu, jak fungují zřetězené šifry a jak se v čase měnily používané šifrovací módy.

Truecrypt [1] je jedním z velmi oblíbených programů pro transparentní šifrování disků. Multiplatformní styl vývoje umožňuje podporovat operačních systémy Windows, Mac OS a Linux.

První verze Truecryptu je původně odvozena od programu E4M (Encryption for Masses).

Licence

Současná verze (7.1a) je licencovaná pod licencí označovanou jako „Truecrypt License Version 3.0“. Licenční text se však od verze Truecryptu 1.0 změnil 21 krát (s odečtením nepodstatných změn jako rozdílné kódování konců řádků).

Verze Truecryptu 2.0 byla uvolněna pod licencí GPL2, ale velmi rychle poté se licence změnila na jinou, mimo jiné s poznámkou „Released under the original E4M license to avoid potential problems relating to the GPL license.“.

Přestože je tedy zástupcem programů s otevřeným zdrojovým kódem, a přestože některé restriktivní podmínky licence byly postupně odstraněny, licence stále vylučuje zařazení do některých Linuxových distribucí, kde je kladen silný důraz na čistotu licenčních podmínek [3, 4].

Tento problém (mimo jiné) vedl k potřebě čisté implementace nástroje pro manipulaci s Truecrypt kontejnery. (Ale existují i pokusy o nezávislou kompilaci původních zdrojových kódů, např. RealCrypt [2]).

Alternativní implementace

Jednou z alternativních implementací je například ScramDisk for Linux [5], který však vyžaduje vlastní jaderný ovladač, nebo tc-play [6], který je velmi dobrou náhradou s volnou BSD licencí a využívající přímo standardní prostředky Linuxového jádra.

Proč tedy další implementace, tentokrát do programu crypsetup? [7] Cílem bylo poskytnout jednoduchý, již integrovaný nástroj na správu transparentního šifrování, který zahrnuje podporu více formátů. Cryptsetup je součástí základního balíku v systému a zároveň poskytuje potřebné rozhraní (není třeba tedy implementovat kryptografické primitiva). V průběhu implementace se pak ukázalo, že by bylo zajímavé částečně implementovat i podporu pro starší Truecrypt kontejnery, která v ostatních alternativních implementacích zcela chybí.

Data na disku

Truecrypt používá k uložení klíče oblast na disku, zde nazývaná Truecrypt hlavička, kde jsou uloženy klíče, kterými se pak šifrují vlastní data. Ostatní místo na disku je pak využito pro uživatelská data, případně jako návnada místo skrytého disku (o skrytém disku viz dále). Celý formát tedy tvoří kontejner pro vlastní datový obsah.

Základním předpokladem vlastností hlavičky je, že ji nelze bez znalosti klíče rozeznat od náhodných dat. Truecrypt kontejner by tedy nemělo být možné bez znalosti klíče ani detekovat.

Hlavička má velikost 512 bytů (tj. jeden sektor disku) a skládá se z nešifrované a šifrované části.

Její umístění na disku závisí od funkce hlavičky. Od verze 6.0 obsahuje Truecrypt i záložní hlavičku na konci disku a zvětšuje velikost hlavičky na 128 KiB. Z této hlavičky je však stále použito jen počátečních 512 bytů strukturou odpovídajících původnímu formátu.

Prvních 64 bytů (nešifrovaná část) je tvořeno náhodně vygenerovanými znaky (tzv. sůl), které jsou rozdílné pro každou hlavičku.

Další část hlavičky je šifrovaná a klíč k dešifrování je heslo, které v kombinaci se solí, algoritmem pro odvození hesla (KDF, Key Derivation Function) a jedním z šifrovacích algoritmů slouží k dešifrování hlavičky.

Formát hlavičky je dostupný v dokumentaci Truecryptu [1]. Rozebírat konkrétní formát hlavičky je nad rámec tohoto příspěvku. Podstatné je, že jej tvoří část s metadaty (různé pomocné atributy jako verze hlavičky a minimální verze programu, který je s hlavičkou kompatibilní) a část s vlastními klíči (a dle použitých algoritmů i například semínka pro generátor IV – inicializačních hodnot vektorů apod.). Obě tyto části jsou zabezpečeny kontrolním součtem (použit je algoritmus CRC32).

Jakmile uživatel zadá heslo, program vyzkouší všechny známé algoritmy (tj. algoritmy šifer, zřetězených šifer a varianty KDF) a pokud po dešifrování naleze na počátku hlavičky řetězec „TRUE“ a zároveň odpovídají kontrolní součty hlavičky, je hlavička považována za korektní a jsou nastaveny šifrovací klíče.

Pro funkci pro odvození klíče je použito PBKDF2 s fixním počtem iterací 1000 (v některých variantách 2 000) a s hash algoritmem RIPEMD-160, SHA1, SHA512 nebo Whirlpool (dle verze Truecryptu lze hash vybrat při formátování).

Pro šifrování datové oblasti je pak použit stejný algoritmus, jaký byl detekován pro dešifrování hlavičky.

Podle funkce existuje několik umístěni hlaviček na disku. Umístění na disku je fixní a je určeno přesně definovanou vzdáleností, buď od počátku disku, nebo od konce disku (pro záložní hlavičky).

Existují tyto hlavičky (všechny mají stejný formát): primární hlavička (začátek disku či particie), záloha primární hlavičky (posledních 128 KiB disku či particie), skrytá hlavička (před verzí 6.0 před koncem disku, od verze 6.0 na 64 KiB od začátku disku), záloha skryté hlavičky (64 KiB od konce disku) a systémová hlavička (pokud je šifrován celý OS, nachází se před začátkem první particie na 62 sektoru celého disku, má vždy 512 bytů a nemá záložní hlavičku).

Šifrovací módy

Jakýkoliv šifrovací nástroj musí reagovat na odhalené slabiny a nabízet pokud možno co nejbezpečnější řešení v rámci možností. Trucrypt je ukázkou programu, který postupně zavádí (a naopak označuje jako zastaralé) používané algoritmy tak, jak se objevují problémy, a nabízí tak state-of-the-art technologie.

Pro přehlednost si uveďme tabulku podporovaných algoritmů a módů, tak jak byly postupně přidávány a modifikovány.

S výjimkou algoritmu IDEA (který byl kompletně odstraněn kvůli licenčním problémům) jsou všechny módy a algoritmy podporované v kompatibilním režimu, ale již s nimi nelze vytvořit nový kontejner.

Zajímavostí je použití Blowfish v módu little-endian (tedy přesně opačně, než jej implementuje např. Linuxové jádro) a použití 64 bitového bloku a LRW módu u některých algoritmů, což sebou nese nutnost implementace operací nad GF(64). (V Linuxovém jádře je dostupný pouze LRW a XTS mód nad 128 bit blokem a GF(128) operace.)

CBC mód (jen u starých kontejnerů)

Při použití CBC módu Truecrypt také používá přidanou operaci, která je velmi podobná tzv. whiteningu [9] (v originálním popisu Trucrypt 1.0 autoří použili

Tab. 1: Šifrovací algoritmy a módy v Truecryptu

Agloritmus	**Blok [bitů]**	**Klíč [bitů]**	**Mód**	**Zavedeno [verze]**	**Zrušeno [verze]**
AES256	128	256	XTS	5.0	–
"	"	"	LRW	4.1	5.0
"	"	"	CBC	2.0	4.1
Serpent	128	256	XTS	5.0	–
"	"	"	LRW	4.1	5.0
"	"	"	CBC	3.0	4.1
Twofish	128	256	XTS	5.0	–
"	"	"	LRW	4.1	5.0
"	"	"	CBC	3.0	4.1
Blowfish (LE)	64	448	LRW	4.1	4.3
"	"	"	CBC	1.0	4.1
CAST5	64	128	LRW	4.1	4.3
"	"	"	CBC	1.0	4.1
TripleDES (EDE)	64	168	LRW	4.1	4.3
"	"	"	CBC	1.0	4.1
IDEA	64	128	CBC	1.0	2.1a

výraz sector scrambling, ale později již používají termínu whitening). Výsledný šifrovaný text je pomocí operace xor a 64 bitové speciální hodnoty (jedinečné pro každý šifrovaný sektor) mixován s výsledným šifrovaným textem.

Inicializační hodnota (IV) pro CBC mód je odvozena ze sekvenčního čísla sektoru a tajné hodnoty z hlavičky (výsledný IV je jen jednoduchý xor obou hodnot).

Whitening a IV generátor již není použit s módem LRW a XTS (tj. od verze 4.1).

Zde je nutné zdůraznit, že jak výpočet IV pro CBC mód, tak pro whitening, není bezpečný.

IV je sice odvozen z tajné hodnoty, ale bohužel je částečně predikovatelný (například jde odvodit, které bity IV se změní pro následující sektor).

Whitening zase používá algoritmu CRC32 (místo kryptograficky bezpečného hash algoritmu) a tento návrh vedle k několika útokům (z nichž jeden, který umožňuje odhalit existenci skrytého disku, si ukážeme dále).

Další zajímavou poznámkou je, že whitening je použit i pří dešifrování hlavičky, kde je použito místo nezávislé hodnoty přímo 64bitů z klíče vygenerovaného pomocí KDF (stejná část klíče je tedy použita ke dvěma účelům – šifrování a whitening).

Zřetězené šifry

V tabulce nejsou zahrnuty zřetězené šifry (algoritmy použité v kaskádě za sebou).

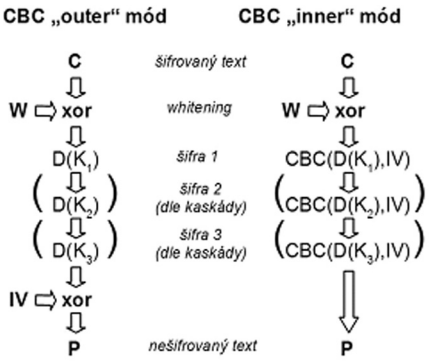
Zřetězené šifry byly zavedeny ve verzi 3.0 a jejich možné kombinace se v průběhu vývoje měnily podle podporovaných algoritmů.

Důvodem pro použití zřetězených šifer je zvýšení bezpečnosti – pokud je jeden algoritmus v kaskádě prolomitelný (například je objeven nový útok na daný algoritmus), další algoritmus v kaskádě zajišťuje, že data jsou stále zabezpečena. Cena za takovéto řešení je podstatná ztráta výkonu šifrovaného disku. Zároveň je také nutný předpoklad, že se algoritmy při takovémto použití vzájemně neoslabují (což je při současných požadavcích na konstrukcí algoritmů nepravděpodobné).

Pokud je použita kaskáda algoritmů, každý klíč v kaskádě je nezávislý. Klíče a semínko IV všech šifer v kaskádě jsou uloženy v hlavičce.

Při použití stejně velkého bloku je implementace zřetězeného módu jednoduchá – výstup z první šifry je použit jako vstup do další. Při použití CBC módu je toto zřetězení označováno jako „outer CBC mód“.

Pokud je však velikost bloku zřetězených šifer různá, je třeba tento problém řešit v rámci kaskády použitím dalšího cyklu, Trucrypt tento mód označuje jako „inner CBC“.



Obr. 1: Rozdíl v dešifrování prvního bloku mezi inner a outer CBC módem v kaskádě

V současné verzi Truecryptu se používají výhradně šifry s velikostí bloku 128 bitů v XTS módu. V Linuxu lze tedy implementovat mód používající zřetězené šifry pomocí zařízení naskládaných nad sebe (což je použito jak v původní implementaci Truecryptu při použití nativních služeb jádra, tak v tc-play a cryptsetupu).

Generátor náhodných čísel

Generátor náhodných čísel (RNG, Random Number Generator) je použit pro generování klíčů, soli a pro algoritmus mazání dat na disku (při formátování). Je zcela zásadní komponentou, na jeho kvalitách závisí kvalita vlastního klíče.

Truecrypt nespoléhá na systém, ale používá vlastní algoritmus (založený na [16, 17]), který mixuje náhodné události (tj. zdroje entropie). Přesný popis algoritmu je součástí dokumentace [1].

Vstupní zdroje entropie se liší podle operačního systému. Společnými zdroji jsou pohyb kurzoru myši (uživatel je vyzván při formátování) a stisk kláves, u Windows dále pak statistiky síťového rozhraní, Crypto API atd. V Linuxu a Mac OS je místo tohoto vstupu přimícháván výstup z /dev/random a /dev/urandom (tj. systémové RNG).

Rozbor interního RNG je nad rámec tohoto popisu a dále budeme předpokládat, že návrh je bezpečný a výstupem jsou dostatečně kvalitní data.

Analýzou RNG se také částečně zabývá [10, 11].

Skrytý disk a skrytý operační systém

Možnost skrytého disku (hidden disk) se objevila ve verzi 3.0, možnost celého skrytého operačního systému pak ve verzi 6.0.

Možnost skrytého disku je založena na myšlence „věrohodného popření“ (plausible deniability). V případě vynucení vydání klíče uživatel klíč vydá, ale jde jen o klíč k návnadě, disku, který je sice šifrován, ale neobsahuje utajovaná data. Utajované data jsou umístěna v další části disku chráněná jiným heslem.

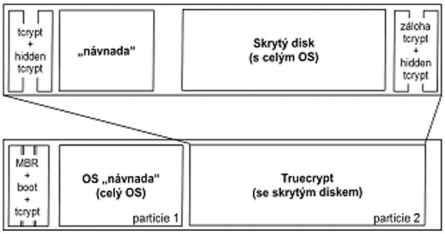
Ponechme stranou otázku sociálního inženýrství a také toho, zda netrénovaný člověk je schopen při pohledu do hlavně nabitého revolveru věrohodně něco popírat, a věnujme se technickým problémům skrytého disku.

Prvním problémem je, že není použito žádné pokročilé techniky na skrytí dat (steganografie), ale je využito jednoduchosti alokační strategie souborového systému FAT. Skrytý disk je jednoduše „ukryt“ v oblasti, kde souborový systém nemá alokována žádná data. Což znamená, že na souborový systém s návnadou smí uživatel zapsat jen tolik dat, kolik nezasáhne do oblasti skrytého disku. Také použití modernějších a obecně žurnálových souborových systému je pro disk s návnadou (outer volume) prakticky vyloučeno – tyto souborové systémy používají zálohy interních metadat rozložené po celém disku (skrytý disk by tedy byl poškozen při nejbližším zápise do nich).

Dalším problémem skrytého disku je prosakování informací o jeho existenci mimo vlastní skrytý disk (například do logů systému apod.). Prakticky je tento problém ukázán v [12]. Například automatické ukládání dokumentů či automatické vytváření zástupců na nedávno použité soubory dokáže existenci disku snadno prozradit (stačí otevřít soubor ve Wordu).

Tento problém vedl (dle doporučení z [12]) k návrhu konceptu takzvaného skrytého operačního systému. Není to nic jiného, než aplikace skrytého disku na celou instalaci OS. Na disku je tedy jako návnada nejen disk s falešnými daty, ale i celý funkční OS.

Prakticky to funguje tak, že Truecrypt zavaděč systému podle zadaného hesla rozpozná, do kterého OS má nastartovat a veškeré stopy po používání jsou zanechány v rámci skrytého OS.



Obr. 2: Skrytý OS a skrytý disk

Příklad útoku na skrytý disk (CBC)

Aby nebyl celý článek jen suchým popisem formátů, zkusme jej doplnit ilustračním popisem útoku popsaném v [18], který má za cíl odhalit existenci skrytého disku pouze z vlastností šifrovaného textu.

Tento útok byl možný pouze do verze 4.1, nejde o aktuální problém. Smyslem je ale ukázat, že je velmi těžké vytvořit bezpečný systém (zejména pokud se v návrhu používají vlastní algoritmy a modifikace).

Předpokládejme, že P je nešifrovaný text (plaintext) a C je šifrovaný text (ciphertext) a předpokládejme použití AES šifry (tj. 128 bit blok).

Předpokládejme, že jsme schopni na skrytý disk dostat libovolný soubor, který je mírně modifikovaný (stačí modifikace prvních bloků v sektorech, tj. pouhé 3 % obsahu) a je správně zarovnaný na počátek sektoru.

Tato operace není tak složitá – stačí emailem zaslat nějaký obrázek, do textového souboru na vhodná místa dát správné komentáře apod. Zarovnání již za nás (ve většině případů) zajistí sám souborový systém.

Máme klíče K, K_{IV}, K_W načtené z hlavičky (tj. šifrovací klíč, semínko IV a klíč pro whitening).

Definujme S_n jako číslo sektoru dělitelné 4, kde platí

$$S_n \operatorname{xor} S_{n+1} = S_{n+2} \operatorname{xor} S_{n+3}.$$

Protože je číslo sektoru obvykle 64bitové, označme si $S_n \parallel S_n$ jako 128 bitové číslo tvořené opakováním čísla sektoru po sobě.

Mějme whitening funkci W, u níž platí

$$W(K_w, S_n \operatorname{xor} S_{n+1}) = W(K_w, S_n) \operatorname{xor} W(K_w, S_{n+1}).$$

Což pro Truecrypt whitening platí, neboť je použit algoritmus CRC32, který splňuje vlastnost

$$\text{CRC32(a xor b)} = \text{CRC32(a) xor CRC32(b)}.$$

Mějme IV_n (inicializační vektor pro sektor S_n) definovaný jako

$$IV_n = K_{iv} \operatorname{xor} (S_n \parallel S_n).$$

Pozměňme sektory $n \,..\, n+3$ obsahující $P_n \,..\, P_{n+3}$ (jejich první blok) takto

P_n obsahuje libovolná data, označme je R

P_{n+1} modifikujme tak, že obsahuje X, kde
$X = R \operatorname{xor} (S_n \parallel S_n) \operatorname{xor} (S_{n+1} \parallel S_{n+1}) =$
$= R \operatorname{xor} (S_{n+2} \parallel S_{n+2}) \operatorname{xor} (S_{n+3} \parallel S_{n+3})$

P_{n+2} obsahuje opět R

P_{n+3} obsahuje opět X

Tj. sektory jdoucí po sobě obsahují v prvním bloku $R \parallel X \parallel R \parallel X$.

Šifrování se nyní dá zapsat pomocí funkce $E(K, P)$

$$\begin{aligned}
C_n &= E(K, R \operatorname{xor} IV_n) \operatorname{xor} W(K_w, S_n) \\
&= \mathbf{E(K, R \operatorname{xor} K_{iv} \operatorname{xor} (S_n \parallel S_n))} \operatorname{xor} W(K_w, S_n) \\
C_{n+1} &= E(K, X \operatorname{xor} K_{iv} \operatorname{xor} (S_{n+1} \parallel S_{n+1})) \operatorname{xor} W(K_w, S_{n+1}) \\
&= E(K, R \operatorname{xor} (S_n \parallel S_n) \operatorname{xor} (S_{n+1} \parallel S_{n+1}) \operatorname{xor} K_{iv} \operatorname{xor} (S_{n+1} \parallel S_{n+1})) \operatorname{xor} \\
&\quad W(K_w, S_{n+1}) \\
&= \mathbf{E(K, R \operatorname{xor} K_{iv} \operatorname{xor} (S_n \parallel S_n))} \operatorname{xor} W(K_w, S_{n+1})
\end{aligned}$$

Obdobně pro C_{n+2} a C_{n+3}.

Nyní můžeme z výrazu eliminovat vliv šifrování a zůstane nám pouze whitening.

$$\begin{aligned}
C_n \operatorname{xor} C_{n+1} &= W(K_w, S_n) \operatorname{xor} W(K_w, S_{n+1}) = W(K_w, S_n \operatorname{xor} S_{n+1}) \\
C_{n+2} \operatorname{xor} C_{n+3} &= W(K_w, S_{n+2}) \operatorname{xor} W(K_w, S_{n+3}) = W(K_w, S_{n+2} \operatorname{xor} S_{n+3})
\end{aligned}$$

a protože jsme si definovali $S_n \operatorname{xor} S_{n+1} = S_{n+2} \operatorname{xor} S_{n+3}$

$$\mathbf{C_n \operatorname{xor} C_{n+1} = C_{n+2} \operatorname{xor} C_{n+3}}$$

Pokud tedy na disku najdeme po sobě jdoucí sektory, jejichž první šifrované bloky splňují tuto podmínku, je velmi pravděpodobné, že jde o modifikovaný soubor, tudíž je v tomto místě zřejmě skrytý disk. (Útok lze samozřejmě zpřesnit opakováním vzoru apod.)

Závěrem však je, že použitý algoritmus umožňuje prozradit existenci skrytého disku, čímž je porušen základní předpoklad pro použití – tedy že šifrovaná data jsou nerozpoznatelná od náhodných dat.

Popsaný útok vychází z popisu v sci.crypt listu [18], kde jsou uvedeny i jednoduché programy na generování a detekci popsaného vzoru, které lze přímo použít na demonstraci výše uvedeného útoku.

Nevhodné generování CBC IV lze zneužít také, viz problém u CBC zvaný watermarking [15].

Autoři reagovali na daný typ problému poměrně rychle a nahradili mód CBC (a whitening) ve verzi 4.1 módem LRW, který byl tehdy horkým kandidátem na schválení IEEE standardu a tyto problémy jednoznačně řeší.

Bohužel, LRW má své problémy také (dokonce v diskuzi o LRW IEEE komise [14] se výslovně potenciální problémy Truecryptu zmiňují).

Dnes je již LRW nahrazen módem XTS, který na odhalení slabin teprve čeká.

Keyfiles

Truecrypt nabízí mimo použití hesla pro odemknutí i možnost použití obsahu souborů tzv. keyfiles.

Jejich použití funguje tak, že uživatel k heslu (nikoliv místo hesla) vybere jeden nebo několik souborů a jejich obsah je přimixován k heslu a výsledný mix použit pro dešifrování hlavičky. Bez přítomností všech daných souborů tedy, alespoň dle definice, nelze odemknout disk.

Existuje zde však několik problémů. První je celkem nepodstatný a spočívá v tom, že z každého keyfile se čte jen maximálně 1 MiB dat (zbytek souboru není použit a z pohledu Truecryptu na jeho obsahu nezáleží).

Druhý problém spočívá v konstrukci celého algoritmu použitého u keyfiles. Ten používá 64 bytů veliký pool (což mimo jiné limituje maximální velikost hesla na 64 znaků), ale hlavně používá CRC32 na mixování poolu s obsahem souborů. Pokud má útočník možnost podvrhnout speciálně upravené soubory keyfiles, lze prakticky eliminovat existenci keyfiles (tj. disk lze dešifrovat i bez přítomnosti souborů, pouhým heslem). Tento útok (včetně odpovědi autorů Truecryptu) je podrobně popsán v [11]. Nutno říci, že útok předpokládá manipulaci se systémem (porušuje tedy security model, pro který je Trucrypt definován).

Použití keyfiles je tedy velmi užitečné ve spojení s různými tokeny, kde je zaručeno, že útočník nemůže modifikovat obsah, ale pochybnosti nad konstrukcí algoritmu zůstávají.

Cryptsetup a jeho Truecrypt podpora

Cryptsetup, pomocí knihovny libcryptsetup, poskytuje základní funkce pro přístup k Truecrypt kontejnerům. Neobsahuje však možnost vytváření a modifikování existujícího kontejneru (což ani nebylo cílem implementace). Cryptsetup je program s otevřeným zdrojovým kódem pod standardní licencí GPL2+ (GPL2 or any later).

Cryptsetup umí načíst všechny formáty hlavičky, jedinou limitací je nedostupnost některých algoritmů v Linuxovém jádře (jmenovitě IDEA a všechny algoritmy s 64 bit blokem používající mód LRW).

Hlavičku lze tedy dešifrovat vždy (což umožňuje nad knihovnou libcrypsetup postavit například nástroj na hledání hesel), ale aktivace vlastního kontejneru je pak možná jen pro kontejnery používající módy LRW a XTS. (Chybí zde podpora operací whitening a kaskádový CBC mód jaderným modulem dmcrypt).

Cryptsetup 1.6 tedy implementuje základní operace, například

- informace z hlavičky:

```
# cryptsetup tcryptDump tst
Enter passphrase:
TCRYPT header information for tst
Version:        5
Driver req.:    7
Sector size:    512
MK offset:      131072
PBKDF2 hash:    sha512
Cipher chain:   serpent-twofish-aes
Cipher mode:    xts-plain64
MK bits:        1536
```

- Aktivace kontejneru

```
# cryptsetup tcryptOpen tst tcrypt_dev
Enter passphrase:
```

  Kde je vytvořeno zařízení /dev/mapper/tcrypt_dev

- Status zařízení

```
# cryptsetup status tcrypt_dev
/dev/mapper/tcrypt_dev is active.
type:    TCRYPT
cipher:  serpent-twofish-aes-xts-plain64
```

```
keysize: 1536 bits
device:  /dev/loop0
loop:    /tmp/tst
offset:  256 sectors
size:    65024 sectors
skipped: 256 sectors
mode:    read/write
```

Více viz poznámky k vydání verze 1.6 [8].

Podpora formátu TCRYPT je rovněž implementována v systéme (který používá libcryptsetup), je tedy možné mapovat zařízení přímo pomocí /etc/crypttab s použitím klíčových slov tcrypt, tcrypt_hidden, tcrypt_system, tcrypt_keyfile podobně jako LUKS zařízení [13].

Závěr

Truecrypt je nepochybně úspěšný program, který přináší uživatelům velmi slušný stupeň ochrany. Avšak ani tady se autoři nevyhnuli chybám v návrhu, které ale byly v následujících verzích (až na výjimky například s algoritmem pro keyfiles) opraveny.

Poučení nevymýšlet vlastní vylepšení algoritmů platí však i zde.

Implementovat podporu Truecryptu (bez použití a dokonce bez nahlédnutí do původních zdrojových kódů) byl zpočátku tak trochu experiment. Časem se ukázalo, že využití je poměrně velké, zejména pro dual boot systémy, kde je potřeba sdílet šifrovaný disk mezi systémy Windows a Linux.

Literatura

[1] *Truecrypt, free open-source on-the-fly encryption.*
http://www.truecrypt.org

[2] *RealCrypt, an encryption application based on TrueCrypt.*
http://wiki.mandriva.com/en/RealCrypt

[3] Fedora wiki, *Truecrypt license troubles.*
http://fedoraproject.org/wiki/Talk:Forbidden_items

[4] Debian legal-list, *Re: licence for Truecrypt.*
http://lists.debian.org/debian-legal/2006/06/msg00295.html

[5] *SD4L – ScramDisk for Linux.*
http://sd4l.sourceforge.net/

[6] *tc-play, Free and simple TrueCrypt Implementation based on dm-crypt.* https://github.com/bwalex/tc-play

[7] *cryptsetup, Setup virtual encryption devices under dm-crypt Linux.* http://code.google.com/p/cryptsetup/

[8] *cryptsetup 1.6 release notes.* http://code.google.com/p/cryptsetup/wiki/Cryptsetup160

[9] Citizendium – the Citizens' Compendium, *Block cipher: Whitening and tweaking.* http://en.citizendium.org/wiki/Block_cipher#Whitening_and_tweaking

[10] Klíma, V., Rosa, T.: *Šifrování USB flash disků zdarma*, ST 8/2007, http://crypto-world.info/klima/2007/ST_2007_08_09_09.pdf

[11] Ubuntu privacy remix, *Security-Analysis of Truecrypt 7.0a.* https://www.privacy-cd.org/en/tutorials/analysis-of-truecrypt

[12] Czeskis, A., Hilaire, D. J. St., Koscher, K., Gribble, S. D., Kohno, T., Schneier, B.: *Defeating Encrypted and Deniable File Systems: TrueCrypt v5.1a and the Case of the Tattling OS and Applications.* http://www.schneier.com/paper-truecrypt-dfs.html

[13] *Systemd crypttab man page.* http://www.freedesktop.org/software/systemd/man/crypttab.html

[14] P1619 working group mailinglist, *Re: P1619: TrueCrypt uses LRW mode.* http://grouper.ieee.org/groups/1619/email/msg01141.html

[15] linux-crypto mailing list, *Re: gonzo cryptography; how would you improve existing cryptosystems?* http://comments.gmane.org/gmane.linux.cryptography/1633

[16] Gutmann, P.: *Software Generation of Practically Strong Random Numbers.* http://www.cs.auckland.ac.nz/_pgut001/pubs/usenix98.pdf

[17] Ellison, C. *Cryptographic Random Numbers.* http://world.std.com/~cme/P1363/ranno.html

[18] sci-crypt mailinglist, *TrueCrypt 4.0 Out* https://groups.google.com/forum/#!topic/sci.crypt/3DxOChZ0lrQ[1-25-false]

PowerShell z pohľadu *nix používateľa

Juraj Michálek

E-mail: juraj.michalek@ysoft.com

PowerShell a jeho miesto vo svete shellov

Shell alebo príkazový riadok umožňuje lepšiu kontrolu počítača. Pomocou príkazou je možné ovládať počítač alebo iné zariadenie. Vo svete *nix operačných systémov sú populárne shelly ako Bash, Zsh.

Podstatný aspekt shellov je možnosť tvorby skriptov. Typicky sa jedná o súbory, ktoré majú prípony .sh.

Veľmi podobné vlastnosti majú napríklad aj skriptovacie programovacie jazyky ako Python alebo Ruby. K týmto jazykom je k dispozícii často aj interaktívny shell. Pre Python existuje ipython, pre Ruby existuje irb.

Dlhú dobu na platforme Windows neexistoval žiadny oficiálne podporovaný shell, ktorý by mal dostatnčné množstvo vlastnosti a bol by aj rozumne použiteľný. Tento nedostatok bol často suplovaný používaním open source nástrojov dostupných pod Cygwinom.

Cygwin obsahuje port väčšiny nástrojov z unixového sveta. Každopádne svet Windows má určité aspekty unikátne a Cygwin ich nedokázal úplne dobre spracovať.

Prvá verzia PowerShellu sa stretla s rozpačitým ohlasom. Každopádne Microsoft začal PowerShell oficiálne distribuovať s novými verziami operačných systémov. Popularita začala pomaly rásť. Podstatný je aj fakt, že okolo PowerShellu vznikla open source komunita.

PowerShell bol navrhnutý tak, aby základná práca bola veľmi podobná unixovým shellom. Tzn. bežné príkazy na prácu sú tiež k dispozícii.

Príklad:
ls – výpis súborov
cd .. – presun o adresár vyššie
cd ∼ – presun do domovského adresára požívateľa

Presmerovanie výstupu a pipe sú tiež k dispozícii. Príklad:
`ls *.png > file-list.txt` – zoznam súborov s príponou .png zapíše do súboru file-list.txt

Príklady

Príklady uvedené v tomto texte môžete nájsť na:
https://github.com/georgik/powershell-examples

Základy PowerShellu

Dopĺňanie príkazov

Bez dopĺňania príkazov je život na príkazovom riadku náročný. PowerShell podporuje dopĺňanie pomocou:

- Tab – dopĺňa možnosti
- Shift + Tab – dopĺňa možnosti spätne

Za zmienku stojí aj fakt, že PowerShell vie dopĺňať nie len príkazy, ale aj parametre k jednotlivým príkazom.

Alternatíva k which

Unixové shelly obsahujú príkaz which, ktorý sa hodí pri ladení a zisťovaní aký program sa spustí pri zadaní nejakého príkazu. PowerShell nemá namapovaný alias na príkaz which, ale má alternatívy príkaz Get-Command (alias gcm).

```
Get-Command ls
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           ls -> Get-ChildItem
```

Vidíme, že príkaz ls je vlastne len alias na príkaz Get-ChildItem
Absolútnu cestu k programu získame nasledovne

```
(Get-Command python).path
c:\Python27\python.exe
```

Služby systému

Pre správcu systému je dôležité mať jednoduchú a rýchlu kontrolu nad službami.

Získať zoznam služieb je možné pomocou príkazu Get-Service. Ako parameter je možné zadať meno služby, prípadne výraz s hviezdičkou.

Zoznam služieb pre Y Soft produkty je možné získať:

```
Get-Service ysoft*
```

Zapnutie služieb je potom možné pomocou:

```
Start-Service ysoft*
```

Zastavenie procesu

Unix obsahuje dôležité príkazy pre prácu s procesmi ako sú ps, kill a pkill.

V PowerShelli je možné na výpis procesov použiť príkaz Get-Process a na zastavenie procesu je je možné použiť Stop-Process.

```
Get-Process *vim
Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
    106      12     3996      11680    85     1,56   2280 gvim

Stop-Process -id 2280
```

Otvorenie súboru v programe

Windows má asociované prípony súborov k jednotlivým programom, ktoré s týmto typom súboru dokážu pracovať. Na otvorenie programu je možné z PowerShellu zavolať príkaz Invoke-Item:

```
Invoke-Item PowerShellNotes.txt
```

Prechádzanie registrov

Po disku je možné pohybovať sa pomocou príkazov ako cd c:\, ls. Čo v prípade registrov?

```
cd HKLM:
cd SOFTWARE\Wow6432Node
ls
```

Registry sa chovajú k PowerShellu veľmi podobne ako súborový systém.

Skripty a bezpečnostné aspekty PowerShellu

Prvým s bezpečnostných aspektov PowerShellu, na ktorý človek narazí, je spúšťanie skriptov. Skripty sú typicky uložené v súboroch s príponou ps1. Pokiaľ chceme takýto skript spustiť, stačí v PowerShelli zadať:

```
.\skript.ps1
```

Výsledok možno prekvapí, ale skript nie je možné spustiť:
cannot be loaded because running scripts is disabled
on this system. For more information, see about_Execution_Policies

Na PowerShell sa totiž aplikuje takzvaná ExecutionPolicy, ktorou je možné vynútiť úroveň zabezpečenia. Aktuálnu hodnotu ExecutionPolicy získame príkazom:

```
Get-ExecutionPolicy
```

V našom prípade sa bude jednať o hodnotu: Restricted
Poďme sa pozrieť na to, aké informácie k tejto oblasti PowerShell poskytuje:

```
Get-Help ExecutionPolicy

Name                        Category  Module                     Synopsis
----                        --------  ------                     --------
Get-ExecutionPolicy         Cmdlet    Microsoft.PowerShell.S...  Gets the
execution policies for the current session.
Set-ExecutionPolicy         Cmdlet    Microsoft.PowerShell.S...  Changes the
user preference for the Windows PowerShell execution policy.
```

K dispozícii sú dva príkaz: Get-ExecutionPolicy a Set-ExecutionPolicy.
Detailnejšie informácie o príkaze Set-ExecutionPolicy nám povedia viac:

```
Get-Help Set-ExecutionPolicy

NAME
  Set-ExecutionPolicy

SYNOPSIS
  Changes the user preference for the Windows PowerShell execution policy


SYNTAX
  Set-ExecutionPolicy [-ExecutionPolicy] <ExecutionPolicy> [[-Scope]
  <ExecutionPolicyScope>] [-Force [<SwitchParameter>]]
  [-Confirm [<SwitchParameter>]] [-WhatIf [<SwitchParameter>]]
  [<CommonParameters>]


DESCRIPTION
  The Set-ExecutionPolicy cmdlet changes the user preference for the
  Windows PowerShell execution policy.

  The execution policy is part of the security strategy of Windows
  PowerShell. It determines whether you can load configuration files
  (including your Windows PowerS hell profile) and run scripts, and
  it determines which scripts, if any, must be digitally signed before
  they will run. For more information, see about_Execution_P
  olicies (http://go.microsoft.com/fwlink/?LinkID=135170).

  NOTE:  To change the execution policy for the default (LocalMachine)
  scope, start Windows PowerShell with the "Run as administrator" option.
```

```
RELATED LINKS
  Online Version: http://go.microsoft.com/fwlink/?LinkID=113394
  Get-AuthenticodeSignature
  Get-ExecutionPolicy
  Set-AuthenticodeSignature
  about_Execution_Policies
  about_Signing

REMARKS
  To see the examples, type: "get-help Set-ExecutionPolicy -examples".
  For more information, type: "get-help Set-ExecutionPolicy -detailed".
  For technical information, type: "get-help Set-ExecutionPolicy -full".
  For online help, type: "get-help Set-ExecutionPolicy -online"
```

Všímnime si poslednej sekcie Remarks. K väčšine príkazov PowerShellu je možné získať príklady použitia pomocou parametru -examples:

```
Get-Help Set-ExecutionPolicy -examples
```

Detailnejšie informácie k príkazu Set-ExecutionPolicy môžeme získať pomocou:

```
Get-Help Set-ExecutionPolicy -detailed

  -ExecutionPolicy <ExecutionPolicy>
      Specifies the new execution policy. Valid values are:

      -- Restricted: Does not load configuration files or run scripts.
         "Restricted" is the default execution policy.

      -- AllSigned: Requires that all scripts and configuration files
         be signed by a trusted publisher, including scripts that you
         write on the local computer.

      -- RemoteSigned: Requires that all scripts and configuration
         files downloaded from the Internet be signed by a trusted
         publisher.

      -- Unrestricted: Loads all configuration files and runs all
         scripts. If you run an unsigned script that was downloaded from
         the Internet, you are prompted for permission before it runs.

      -- Bypass: Nothing is blocked and there are no warnings or prompts.

      -- Undefined: Removes the currently assigned execution policy from
         the current scope. This parameter will not remove an execution
         policy that is set in a Group Policy scope.
```

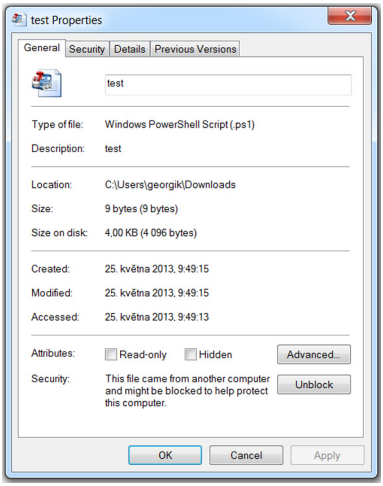
Úrovne ExecutionPolicy Bypass a Unrestricted

Najnižšia úroveň zabezpečenia Bypass nie je odporúčaná, pretože sú vyradené z chodu všetky ochranné mechanizmy. Pokiaľ dôjde k stiahnutiu skriptu z internetu, tak PowerShell nebude overovať jeho pôvod.

Bezpečnejšia úroveň Unrestricted už umožňuje spúšťať používateľovi skripty, ale pokiaľ tento skript zavolá iný skript, ktorý pochádza z neovereného zdroja z internetu, dôjde k zastaveniu vykonávania príkazov. Zobrazí sa dialóg, ktorý umožňuje používateľovi zastaviť vykonávania skriptu.

Skripty prevzaté z internetu

Zaujímavou vlastnosťou systému Windows je sledovanie pôvodu súborov. Pokiaľ prevezmeme z internetu súbor, sytém Windows ho automaticky označí ako zamknutý.



Obr. 1

V prípade pokusu o spustenie takéhoto súboru pri ExecutionPolicy Unrestricted sa PowerShell spýta používateľa, či chce skript spustiť:

```
Security warning
Run only scripts that you trust. While scripts from the internet can be
useful, this script can potentially harm your computer. Do you want to
run C:\Users\georgik\Downloads\test.ps1?
[D] Do not run  [R] Run once  [S] Suspend  [?] Help (default is "D"):
```

Skript je možné odoknúť klepnutím na tlačítko Unblock.

Skripty prevzaté z internetu v archíve zip

Na internete je možné nájsť veľké množstvo balíčkov, napríklad UIAutomation. Tento je distribuovaný vo formáte zip. Je zrejmé, že po prevzatí zip balíčku systém Windows automaticky tento súbor označí.

Môžeme si položiť otázku, čo sa stane so skriptami, ktoré sú v takomto súbore a my ich rozbalíme.

Odpoveď je jednoduchá. Systém Windows automaticky označí všetky rozbalené súbory ako blokované. V prípade UIAutomation balíčku pozostávajúceho z veľkého množstva skriptov to spôsobí nepríjemné chovanie. Pri každom volaní skriptu sa vykonávanie zastaví a bude požadovať potvrdenie od používateľa. Naviac UIAutomation často volá svoje skripty medzi sebou. Dôsledok je ten, že jedne príkaz môže vyžadovať aj viac než 20 potvrdení.

Riešenie je v tomto prípade jednoduché. Je nutné odblokovať priamo zip archív po prevzatí z internetu a rozbaliť archív.

Skripty prevzaté z internetu pomocou iných nástrojov

Blokovanie súborov prevzatých z internetu sa aplikuje na internetové prehliadače. Zaujímavým faktom je, že pokiaľ sa použije iný nástroj na prevzatie súboru, tak Windows ho neoznačí ako blokovaný.

Príklad: curl -O http://georgik.sinusgear.com/wp-content/powershell/test.ps1

Podpisovanie skriptov

Na podpisovanie skriptov potrebujeme certifikát. Na tento účel nám poslúži nástroj makecert, ktorý sa nachádza v SDK balíčku Visual Studia. Spustime príkazový riadok: Developer Command Prompt for VS2012. Vytvoríme si vlastný certifikát.

```
makecert -n "CN=PowerShell Local Certificate Root" -a sha1 -eku
1.3.6.1.5.5.7.3.3 -r -sv root.pvk root.cer -ss Root -sr localMachine
makecert -pe -n "CN=PowerShell User" -ss MY -a sha1 -eku
1.3.6.1.5.5.7.3.3 -iv root.pvk -ic root.cer
```

Vrátime sa späť do PowerShellu a pomocou nasledujúceho príkazu si vypíšeme zoznam certifikátov:

```
Get-ChildItem cert:\CurrentUser\My -codesign
    Directory: Microsoft.PowerShell.Security\Certificate::CurrentUser\My

Thumbprint                                Subject
----------                                -------
D242D8B938F26F2461AD170E6AB807824F87B28C  CN=PowerShell User
```

Podpis nájdeme pripojený do tela skriptu ako komentár na jeho konci. Ak nastavíme ExecutionPolicy na AllSigned, tak sa nás systém pri spustení podpísaného skriptu spýta, či chceme dôverovať vydavateľovi. Po potvrdení sa skript spustí.

Používateľská pohodlnosť PowerShellu

Základná konfigurácia PowerShellu môže byť dostatočná. *nix používatelia sú ale zvyknutí na vyšší komfort.

Vim a PowerShell

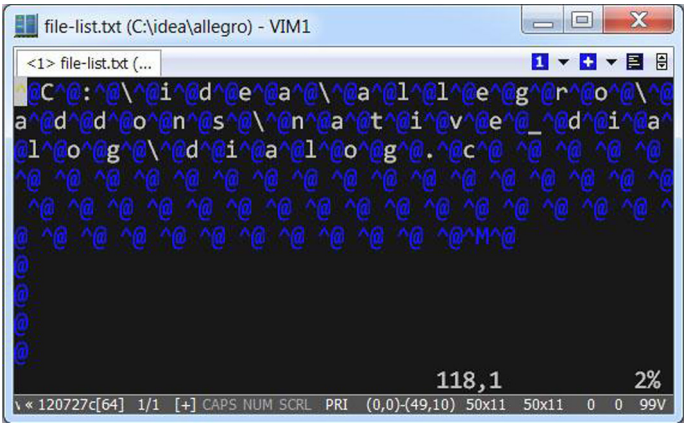
Editor Vim funguje s PowerShellom bez vačších problémov.

Problém nastane pokiaľ sa pokúsime otvoriť vo Vime súbor vytvorený pomocou presmerovania vystupu do súboru. Zistíme, že nie je čitateľný, pretože PowerShell generuje výstup v unicode.

Príklad:

```
Get-ChildItem -Recurse | Select-Object -ExpandProperty FullName
> file-list.txt
```

Výsledok:



Obr. 2

Opravu tohoto stavu navrhol Tony Mechelynck (http://vim.1045645.n5.nabble.com/Opening-files-with-Unicode-names-under-Windows-td1207885.html). Do konfigurácie vim (~/.vimrc) stačí pridať nasledujúci kód:

```
if has('multi_byte')                    " multibyte features compiled-in
  if &encoding !~? '^u'                 " the OS locale is not Unicode
    if &termencoding == ''              " empty means 'use &enc'
      let &termencoding = &encoding     " avoid clobbering keyboard codes
    endif
    set encoding=utf-8 " we can do it, now that the kb is taken care of
  endif
  set fileencodings=ucs-bom,utf-8,latin1 " heuristics for existing files
  setglobal bomb fileencoding=latin1    " defaults for new files
                    " 'bomb' doesn't apply to latin1
                    " it applies when 'fenc' is manually set to Unicode
endif
```

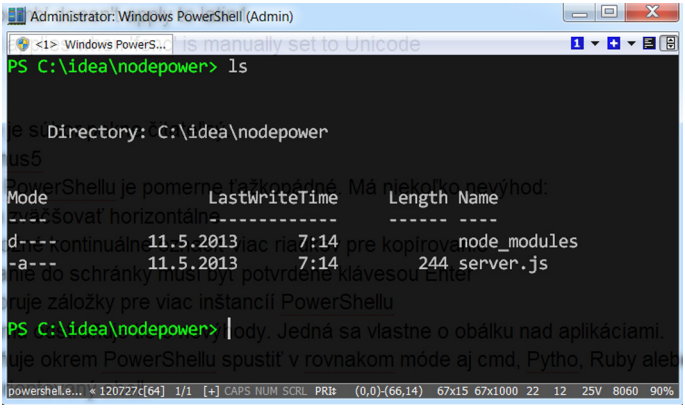
Po tejto úprave je súbor pekne čitateľný.

ConEmu Maximus5

Základné okno PowerShellu je pomerne ťažkopádné. Má niekoľko nevýhod:

- nedá sa zväčšovať horizontálne
- nie je možné kontinuálne označiť viac riadkov pre kopírovanie
- kopírovanie do schránky musí byť potvrdené klávesou Enter
- nepodporuje záložky pre viac inštancií PowerShellu

Program ConEmu odstraňuje tieto nevýhody. Jedná sa vlastne o obálku nad aplikáciami. ConEmu umožňuje okrem PowerShellu spustiť v rovnakom móde aj cmd, Pytho, Ruby alebo iný príkazovo orientovaný shell.



Obr. 3

V ConEmu je možné nastaviť si vlastné klávesové skratky. Za zmienku stojí základné nastavenie kopírovania do schránky: text je možné označiť pri súčastnom stlačení klávesy Shift a ľavého tlačitka myši.

Domovská stránka projektu: https://code.google.com/p/conemu-maximus5/

Nastavenie profilu pre PowerShell

Bash umožňuje uložiť nastavenie do skriptu, ktorý je spustení pri jeho štarte. Typicky sa do takéhoto skriptu ukladá nastavenie ciestu alebo command promptu.

PowerShell má podobnú vlastnosť. Skript je možné uložiť do:
`~\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1`

Príklad nastavenia:

```
$env:path += "c:\Python27;C:\Ruby193\bin"
$env:SVN_EDITOR="vim"
$env:node_path="$home\AppData\Roaming\npm\node_modules"

# Color prompt
function prompt {
    Write-Host ("PS " + $(get-location) +">") -nonewline
       -foregroundcolor Green
    return " "
}
```

Príklady použitia PowerShellu

V tejto kapitole sa pozrieme na praktické príklady použitia PowerShellu.

Spracovanie XML získaného z internetu

PowerShell dokáže efektívne spracovať XML súbory. Skúsme naimplementovať scenár, kedy chceme získať číslo verzie Forsquare Java API a meno licencie API.

Informácie o verzii sú uložené v súbore pom.xml, ktorý je na adrese:
http://foursquare-api-java.googlecode.com/svn/trunk/pom.xml

Obsah súboru je nasledovný:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/maven-v4_0_0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>fi.foyt</groupId>
        <artifactId>foursquare-api</artifactId>
        <packaging>jar</packaging>
        <version>1.0.3-SNAPSHOT</version>
```

```
        <name>Foursquare API</name>
        <licenses>
                <license>
                        <name>GNU LGPL v3</name>
                        <url>http://www.gnu.org/licenses/lgpl.txt</url>
                        <distribution>repo</distribution>
                </license>
        </licenses>
....
</project>
```

Kód:

```
$url = "http://foursquare-api-java.googlecode.com/svn/trunk/pom.xml"
[xml]$data =(New-Object System.Net.WebClient).DownloadString($url)
$data.project.version
$data.project.licenses.license.name
```

Výsledok:

```
1.0.3-SNAPSHOT
GNU LGPL v3
```

Integrácia NodeJS a PowerShellu

NodeJS je populárnou technológiou na vytváranie serverový aplikácii, ktorú sú prístupné cez REST API. Pomocou projektu Edge je možné pomerne jednoducho prepojiť NodeJS server s PowerShell skriptovaním a získať tak omnoho väčšie možnosti NodeJS na platforme Windows.

Do NodeJS aplikácie stačí pridať moduly edge a edge-ps:

```
npm install edge
npm install edge-ps
```

Kód pre NodeJS aplikáciu môže vyzerať nasledovne:

```
var edge = require('edge');

var hello = edge.func('ps', function () {/*
"PowerShell welcomes $inputFromJS on $(Get-Date)"
*/});

hello('Node.js', function (error, result) {
 if (error) throw error;
 console.log(result[0]);
});
```

Spustíme: `node app.js`

Výsledok: PowerShell welcomes Node.js on 05/11/2013 07:14:38

Automatizácia GUI testovania pomocou UIAutomation

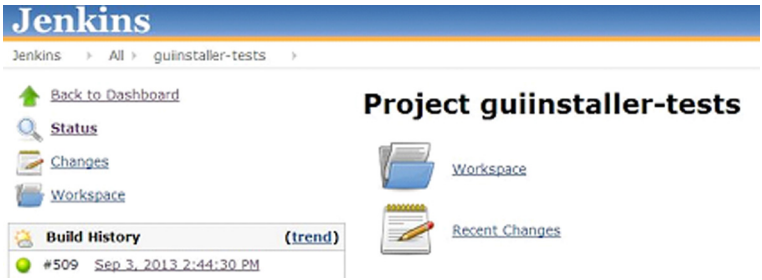
UIAutomation PowerShell Extenstion – (http://uiautomation.codeplex.com/ – umožňuje automatizáciu operácií na aplikáciami s grafickým rozhraním. Podobnú funkcionalitu poskytuje napríklad projekt Sikuli, ale jeho nevýhodou je nepresnosť a nízky výkon.

UIAutomation narozdiel od Sikuli pracuje na úrovni operačného systému a dokáže tak rýchlo nájst GUI elementy podľa ich textu alebo iných atribútov. Nad GUI elementami je potom možné zavolať rôzne operácie, napríklad: Click

PowerShell UI Automation a Jenkins

Skripty automatizujúce inštaláciu sú používané napríklad na testovanie GUI Installerov. Takéto skripty môžu byť vovolávané napríklad pomocou CI nástroja Jenkins. Tým pádom je po každom zostavení aplikácie možné vygenerovať snímky obrazoviek, podľa ktorých je možné posúdiť kvalitu produktu.

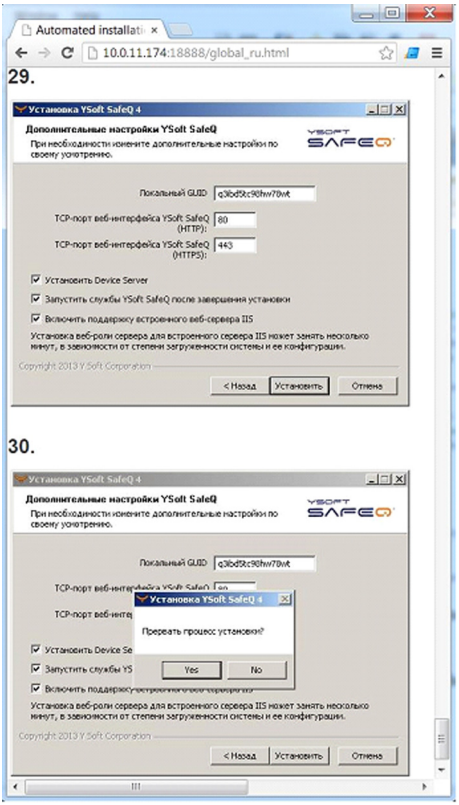
Pozrime sa na príklad použitia pri automatizácii testovanie Ysoft SafeQ GUI Installeru.



Obr. 4: Úloha definovaná v nástroji Jenkins

```
113  # Embedded database settings screen
114  Write-Host "Processing Embedded database configuration screen..."
115  # Change focus to other button. Next button is for some reason pressed, but click was not dispatched
116  Get-UIAButton -Name $language.cancel | Set-UIAFocus
117  ProcessEmbeddedDatabaseSettingsScreen $language
118
119  WaitForButton  $language.next | Invoke-UIAButtonClick
120
121  # Additional settings screen (GUID/ports/device server)
122  Write-Host "Processing Additional settings screen (GUID/ports/device server)..."
123  Get-UIAButton -Name $language.cancel | Set-UIAFocus
124  TakeSnapshot $language.code
125  WaitForButton  $language.install
126
127  Write-Host "Exiting installer..."
128  # abort installation
129  Get-UIAButton -Name $language.cancel | Invoke-UIAButtonClick
130  TakeSnapshot $language.code
131  # yes, we really want to quit
132  WaitForButton  $language.yes | Invoke-UIAButtonClick
```

Obr. 5: Príklad kódu



Obr. 6: Výsledok generovaný do HTML, príklad z jazyku ruština

Automatizácia Web testovania

Nasledujúci príklad ukazuje použitie PowerShellu na vytvorenie automatického skriptu pre testovanie web aplikácie pomocou Internet Explorer.

```
Param (
    [string]$text = "http://europen.cz"
)
$ie = new-object -com "InternetExplorer.Application"
$ie.navigate("http://qr.sinusgear.com")
$ie.Visible = $True
$element=$ie.Document.getElementById("qrCodeInput")
$element.value = $text

# Call JavaScript event to update URL
$ie.document.parentWindow.execScript("qrKeyUpHandler(null)","javascript")
```

Výpočet SHA1

Pri prenose súborou je bežnou praxou overiť si integritu spboru a spočítať SHA1 alebo podobný hash z jeho obsahu. PowerShell umožňuje volanie funkcií zo System.Security. Uložme nasledujúci skript do súboru Get-ReadmeHash.ps1.

```
$sha1 = new-Object System.Security.Cryptography.SHA1Managed
$sha1.ComputeHash((Get-Content -encoding byte .\README.md)) |
%{ Write-Host -NoNewline $_.ToString("x2") }
Write-Host
```

Príkaz:

```
.\Get-ReadmeHash.ps1
```

Výsledok:

```
5750a46aae5ba6d846f865bf078f879bb082bc15
```

Python virtualenv

Dôležitou súčasťou pri vývoji Python aplikácií je možnosť vytvoriť si izolované prostredie, tak aby sa vývojár vyhol nechcenej interakcii medzi balíkmi rôznych aplikácií. Na tento účel slúži nástroj virtualenv.

Virtuálne prostredie (napr. pyenv) vytvoríme nasledovne

```
pip install virtualenv
virtualenv pyenv
```

Python vytvorí skeleton prostredia. Na aktiváciu a nastavenie všetkých ciest už zostáva len v Pythone zadať nasledujúci príkaz:

```
. .\pyenv\Scripts\activate.ps1
```

Ak prebehlo nastavenie prostredia korektne, tak v príkazovom riadku sa nám objaví meno prostredia v zátvorkách.

Zhrnutie

PowerShell je možné efektívne využívať na správu alebo vývoj na platforme Windows. Unix používatelia ocenia podobnosť s prostredím bežných shellov a zároveň získajú novú funkcionalitu.

Je vhodné venovať určitý čas pochopeniu syntaxe. Tu nám môže pomôcť množstvo open source projektov, ktoré podstatne rozširujú základnú funkcionalitu PowerShellu.

Techniky vyhýbania sa sieťovej detekcii

Marián Novotný

E-mail: novotny@eset.sk

Abstrakt

Systémy detekcie sieťových útokov obvykle vytvárajú model sieťovej komunikácie za účelom identifikácie „zlých“ dát. Komplexnosť protokolov, nedodržiavanie špecifikácií výrobcami softvéru, prípadne rôznorodosť implementácií protokolov robia návrh sieťového IDS systému výzvou. Prednáška bude vychádzať z praktických skúseností získaných počas vývoja IDS systému na detekciu zneužitia zraniteľnosti MS WINDOWS sieťových protokolov (SMB, DCE/RPC). Prednáška poskytne príklady útokov, prediskutuje viaceré spôsoby detekcie, príklady techník a nástrojov pre vyhýbanie sa sieťovej detekcii v MS WINDOWS sieťových protokolov.

1 Úvod

Systémy detekcie útokov (*IDS*) na sieťovej úrovni napomáhajú v zabezpečení počítačových sietí a koncových staníc používateľov. Zjednodušene môžeme IDS opísať ako systém, ktorý si z monitorovanej sieťovej komunikácie vytvára vlastný model komunikácie za účelom detekcie „zlých“ dát. Monitorovanie sieťovej komunikácie môže byť realizované na koncovej stanici, alebo z odchytenej sieťovej komunikácii na sieťovej sonde. Účinnosť IDS systému sa obvykle vyjadruje v miere schopnosti detekcie čo najväčšieho počtu útokov a miere *falošných poplachov*, ktoré sa môžu vyskytnúť pokiaľ IDS systém označí legitímnu komunikáciu ako útok. Existuje viacero komerčných IDS systémov, pričom z *„open-source“* nástrojov možno spomenúť systém SNORT [9] a Bro [4].

Komplexnosť sieťových protokolov, nedodržiavanie špecifikácií výrobcami softvéru, prípadne rôznorodosť implementácií protokolov robia návrh sieťového IDS systému výzvou. Ako poznamenal Ptacek a Newsham v [2], sieťový IDS systém má dve *principiálne zraniteľnosti* – nedostatok informácií o sieťovej komunikácií a náchylnosť na útoky zahltenia služby, takzvané *DOS* útoky. IDS

systému môžu pre správne rozhodnutie chýbať potrebné informácie o komunikujúcej koncovej stanici, jej operačnom systéme, prípadne informácie o komunikujúcej aplikácii. Ak model IDS systému nedokáže interpretovať prichádzajúce dáta, tak má v princípe dve možnosti: buď dáta povolí, alebo zakáže. Zakázaním dát vzniká riziko, že IDS zablokoval legitímnu komunikáciu a následne môže dôjsť k zvýšeniu výskytu falošných poplachov. Na druhej strane, ak IDS systém povolí komunikáciu, ktorej nerozumie, tak vzniká bezpečnostná diera, ktorú môže využiť útočník za účelom obídenia inšpekcie IDS systému. Komplexnosť sieťových protokolov a rôznorodosť ich implementácií robia návrh IDS systému komplikovanejším. Je zrejme náročné vyvinúť systém, ktorý by úplne rozumel všetkej sieťovej komunikácii. Príliš detailný model sieťovej komunikácie vyžaduje značné pamäťové a časové výpočtové nároky a s tým súvisí riziko DOS útoku.

Cieľom útočníka môže byť napríklad vyhnutie sa detekcii zlých dát vo forme *„exploitu“* známej zraniteľnosti. V príspevku uvedieme príklad implementačnej zraniteľnosti *MS 08-67*, prediskutujeme spôsoby detekcie zneužitia tejto zraniteľnosti a popíšeme viaceré techniky vyhýbania sa detekcii na rôznych úrovniach TCP/IP protokolu, pričom vymenuje nástroje pre realizáciu útokov.

2 Zraniteľnosť MS 08-67

Zraniteľnosť v operačnom systéme *MS WINDOWS* vo funkcii *NetpwPathCanonicalize* [8] z knižnice netapi32.dll umožňovala vzdialené spustenie kódu pomocou špeciálne pripravených vstupných dát. Táto funkcia má normalizovať cesty súborového systému, pričom je prístupná na diaľku cez operácie *NetprPathCanonicalize, NetprPathCompare* pomoocu *DCE/RPC* rozhrania *ServerService* [8]. Pre túto zraniteľnosť existuje viacero funkčných exploitov, pričom známou sa stala vďaka červovi s názvom *Conficker*, ktorý sa pomocou nej šíril. Sieťový IDS systém, ktorý chce chrániť pred zneužitím tejto zraniteľnosti by mal byť schopný v sieťových tokoch nájsť „zlú“ cestu, ktorá je vstupným parametrom pre spomenuté funkcie. DCE/RPC je v prostredí MS WINDOWS transportované aj pomocou *SMB* (*Server Message Block*) [8] protokolu vo verzi 1.0 (ďalej len SMB).

3 Techniky vyhýbanie sa na IP úrovni

V článku [2] boli ukázané viaceré techniky ako sa vyhnúť IDS detekcii na IP úrovni. Napr. útočník môže poslať „zlú“ cestu v RPC žiadosti z MS 08-67 vo viacerých IP paketoch, navyše v rôznom poradí. Aj napriek tomu, že v *IPv6* nie je fragmentácia súčasťou hlavičky, ale fragmentáciu je možné realizovať odosielateľom pomocou rozširujúcej hlavičky. Navyše aj funkčná IPv6 infraštruktúra môže slúžiť na vyhnutie sa detekcii, pokiaľ IDS systém podporuje len IPv4.

4 Techniky vyhýbania sa na TCP úrovni

Podobne ako na IP úrovni môže útočník rozdeliť DCE/RPC žiadosť so „zlou“ cestou do viacerých TCP fragmentov a tie poslať osobitne. Problém vyskladania TCP dát z jednotlivých fragmentov vyzerá jednoducho. Čo však nastane, ak sa fragmenty prekrývajú? Vzhľadom na to, že špecifikácia TCP/IP protokolu nerieši tento problém, tak existuje viacero rozdielnych implementácií. Prehľad jednotlivých operačných systémov a ich algoritmov poskytuje článok [2].

5 Vyhýbanie sa na aplikačnej úrovni

V tejto časti pre ilustráciu uvedieme príklady vyhýbania sa detekcii v komplexných protokoloch medzi ktoré patrí SMB a DCE/RPC protokol.

SMB

SMB protokol v MS WINDOWS prostredí slúži na zdielanie súborov, tlačiarní a poskytuje prístup pre medziprocesovú komunikáciu pomocou takzvaných *„named pipes“*. Tie slúžia aj pre prístup ku DCE/RPC rozhraniam – napr. pre ServerService je určená rúra s názvom *„srvsvc“*. SMB protokol je komplexný a redundantný, napríklad pripojiť sa na zdielaný priečinok je možné pomocou 2 príkazov, otvoriť súbor sa dá pomocou 7 príkazov, protokol podporuje viaceré kódovania reťazcov, ktoré sa dajú meniť a pod. Útočník môže skúšať, ktoré z týchto možností pozná IDS systém. DCE/RPC žiadosť so zlou cestou môže byť rozfragmentovaná pomocou viacerých zapisovacích príkazov, ktoré navyše môžu byť zreťazené v jednej SMB žiadosti.

DCE/RPC

DCE/RPC slúži pre vzdialené volanie funkcií. Keďže protokol nie je závislý na operačnom systéme ani na transportnom protokole, tak DCE/RPC poskytuje vlastný mechanizmus pre fragmentáciu vstupných dát, podporuje mnoho kódovaní dátových typov a taktiež okrem iného umožňuje zmeniť rozhranie v rámci spojenia. Navyše v prostredí MS WINDOWS môže byť DCE/RPC prenášane cez protokoly: UDP, TCP, SMB, HTTP [8].

6 Nástroje

Impacket [6] umožňuje vytvorenie špecálnych SMB príkazov, ktoré môžu slúžiť na vyhnutie sa detekcii a má taktiež implementované časti SMB, DCE/RPC protokolu potrebné pre autentifikáciu, zmenu RPC rozhrania a pod. Nástroj

Evader [5] má už implementované viaceré techniky pre vyhnutie sa detekcii na úrovni IP, TCP, SMB, RPC protokolu pre exploit zraniteľnosti MS 08-67. Nástroj *Metasploit* [7] taktiež obsahuje viaceré techniky vyhýbania sa detekcii u SMB, DCE/RPC protokolov.

7 Záver

V článku sme prediskutovali principiálne zraniteľnosti sieťového IDS, ktoré vyplývajú z komplexnosti návrhov a realizácii sieťových protokolov. Uviedli sme príklad implementačnej serverovskej zraniteľnosti, kde vstupné dáta boli transportované pomocou DCE/RPC, SMB a TCP protokolov. V článku sme sa zaoberali len *binárnymi protokolmi*, ale podobné techniky existujú aj pre *textové protokoly* akými sú napr. *HTTP, FTP*. Analogický existujú podobné techniky aj pre iné typy detekcií medzi ktoré patrí napr. detekcia zneužitia *klientskych zraniteľnosti*.

Referencie

[1] Novak, J., Sturges, S.: *Target-Based Fragmentation Reassembly*. Sourcefire, 2007.

[2] Ptacek, T. H., Newsham, T. N.: *Insertion, evasion, and denial of service: Eluding network intrusion detection*. Secure Networks, Inc. 1998.

[3] Sotirov, A.: *Decompiling the vulnerable function for MS08-067*. Dostupné na http://www.phreedom.org/blog/2008/decompiling-ms08-067/

[4] http://www.bro.org/

[5] http://evader.stonesoft.com/

[6] http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Impacket

[7] http://www.metasploit.com/

[8] http://msdn.microsoft.com/

[9] http://www.snort.org/