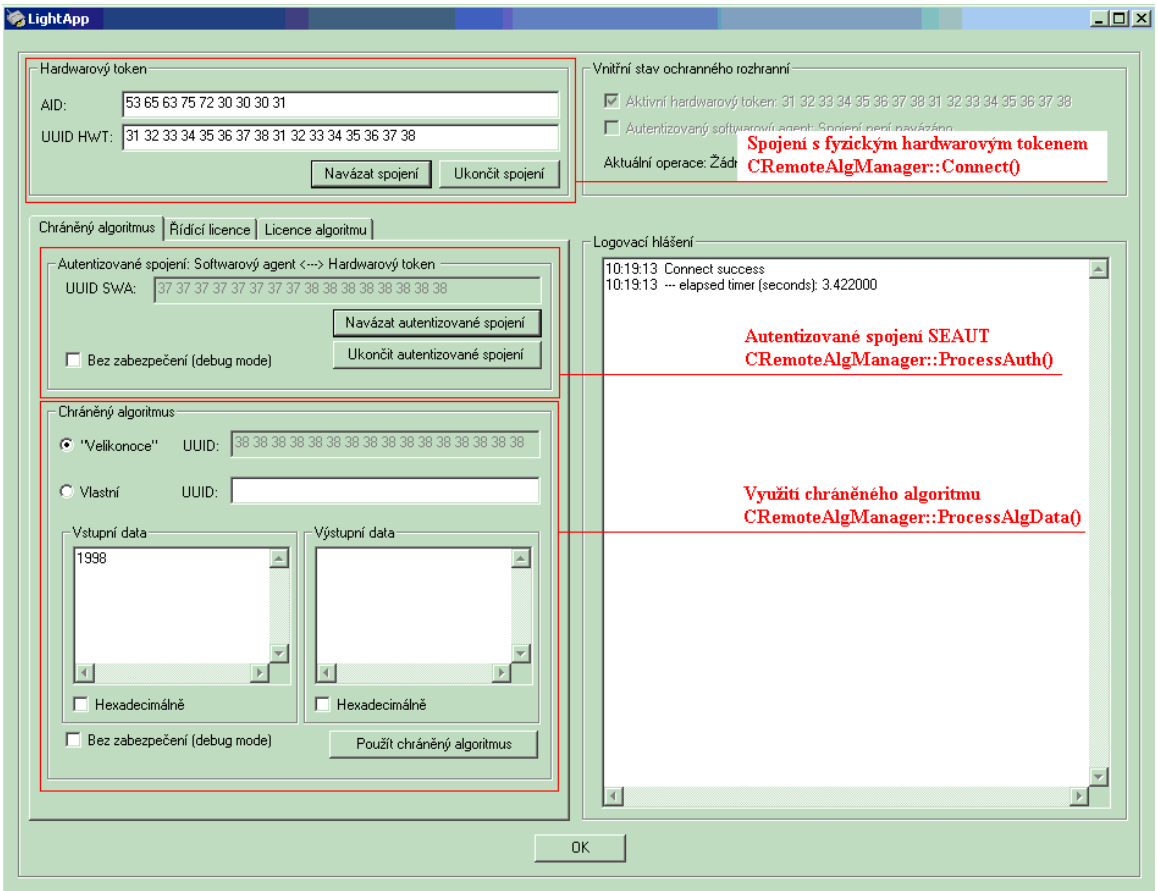
# Příloha A

## A.1 Ukázka užití ochranného rozhranní (LightApp.exe)

Pro praktickou demonstraci principů fungování ochranného rozhranní byl vyvinut ukázkový softwarová agent, aplikace LightApp (CD:\Bin\LightApp\LightApp.exe), která umožňuje interaktivně využívat ochranné rozhranní. Aplikace je napsána v jazyce C++ s využitím knihovny Microsoft Foundation Class. Je určena pro operační systém MS Windows 2000/XP, a vyžaduje nainstalovanou podporu pro čipové karty firmy Gemplus. Pro možnost použití musí být dostupná čipová karta Gemplus GXPPro-R3 s předinstalovaným balíkem SecureAlg.

### A.1.1 Použití chráněného algoritmu

Obrázek 8.3 zachycuje tři kroky využití funkcí ochranného rozhranní. Nejprve dochází k volbě čipové karty dostupné operačnímu systému. Dle zadaného AID dochází k aktivaci „appletu“ s instalovaným hardwarovým tokenem (krok 1A). Následuje ustanovení zabezpečeného komunikačního kanálu dle protokolu SEAUT (krok 2A). Lze zadat unikátní identifikaci softwarového agenta, autentizační a transportní klíče jsou

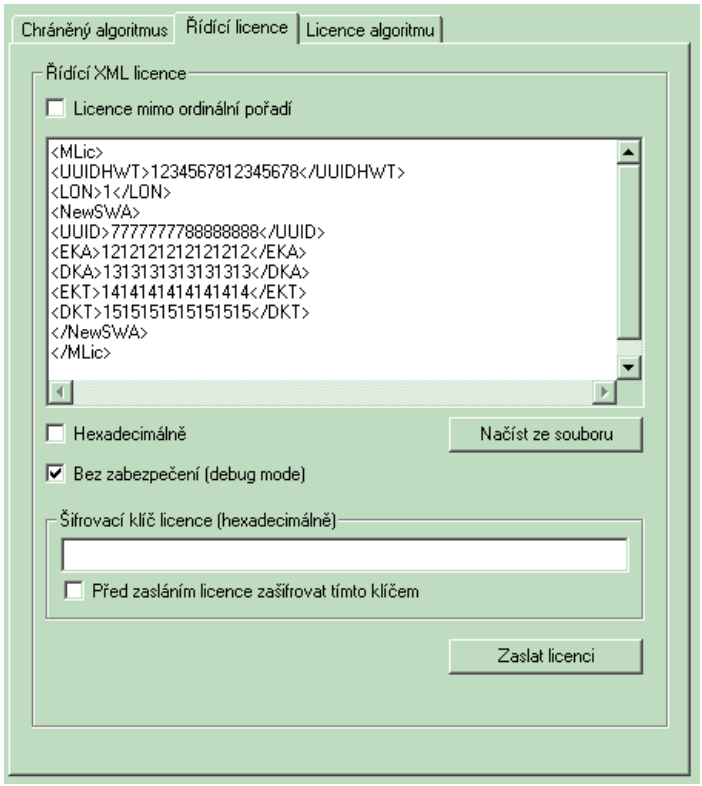



Obr. 8.3: Použití chráněného algoritmu.

vloženy již při kompilaci aplikace. Pokud dojde k úspěšnému vytvoření komunikačního kanálu, lze zaslat požadavek na využití chráněného algoritmu dle jeho UUID (krok 3A).

### A.1.2 Zaslání řídící XML licence

Obrázek 8.4 zachycuje dva kroky vedoucí k aplikaci řídící XML licence. Nejprve dochází k volbě čipové karty dostupné operačnímu systému, stejně jako v předchozím případě (krok 1B). Následuje příprava a zaslání řídící licence hardwarovému tokenu (krok 2B). Lze zadat hodnotu šifrovacího klíče, který bude použit pro zašifrování licence před jejím odesláním, nebo lze zasílat již předem šifrovanou licenci (typické použití). Pokud bude hardwarovému tokenu zasílána řídící licence mimo ordinální pořadí, která způsobí nepoužitelnost licencí s číslem menším než zasílaná, je nutné zaškrtnout příslušné políčko. Příprava a zaslání XML licence k algoritmu probíhá obdobným způsobem.




Obr. 8.4: Zaslání řídící XML licence.

Na přiloženém CD lze nalézt ukázkovou posloupnost operací s podrobně rozepsanými parametry. Ukázková posloupnost umožní:

1. Instalovat hardwarový token s chráněným algoritmem „Velikonoce“. Autorem algoritmu je Karl Friedrich Gauss.
2. Generovat WBACR AES tabulky pro autentizační a transportní klíče.
3. Spustit aplikaci LightApp.
4. Aplikovat řídící licenci, která vytvoří profil softwarového agenta na hardwarovém tokenu.

5. Aplikovat řídící licenci, která vytvoří profil algoritmu „Velikonoce“, určeného pro výpočet dne, na které připadají v zadaném roce velikonoce. Aplikovat řídící licenci, umožní softwarovému agentovi využívat chráněný algoritmus „Velikonoce“.
6. Aplikovat licenci algoritmu „Velikonoce“, která umožní použít algoritmus 5x.
7. Aplikovat licenci algoritmu „Velikonoce“, která umožní použít algoritmus 3x.
8. Aplikovat řídící licenci, zruší profil softwarového agenta na hardwarovém tokenu.

Zdrojové kódy aplikace LightApp jsou dostupné na přiloženém CD (CD:\SourceCodes\LightApp). Aplikace nepoužívá z důvodu přehlednosti vstupní výstupní kódování WBACR AES tabulek pro transportní klíče. Z téhož důvodu není použita žádná z doporučovaných dodatečných ochran typu obfuskace nebo sebekontrolující kód.

## A.2 Generátor WBACR AES (AESGen.exe)

Jednou z klíčových myšlenek komunikačního protokolu SEAUT je implementace šifrovacího algoritmu AES formou WBACR AES tabulek. Řádková utilita AESGen.exe[1] je určena pro generování WBACR AES tabulek. Vygenerované hlavičkové soubory byly použity při překladu aplikace LightApp.

Utilita se spouští s přepínačem /g: a specifikací cesty ke skriptu obsahujícímu konfigurační soubory, například „AESGen.exe /g:Example/wbacraes.cfg“. Konfigurační soubor obsahuje hodnotu šifrovacího a dešifrovacího klíče WBACR AES tabulky, a seznam cest k hlavičkovým souborům, které budou naplněny vygenerovanými hodnotami WBACR AES tabulek. Takto vygenerované hlavičkové soubory, personalizované pro konkrétní hodnoty šifrovacího a dešifrovacího klíče, jsou následně použity pro překlad softwarového agenta se začleněným ochranným rozhranním.

V adresáři CD:\Bin\AESGen\Example na přiloženém CD lze nalézt funkční příklad konfiguračního skriptu a předlohových hlavičkových souborů. Zdrojové kódy generátoru jsou umístěny v CD:\SourceCodes\WBACR_AES.

[1] Využívající metod třídy CWBACRAESGenerator.

# Příloha B

## B.1 Postup začlenění ochranného rozhranní

Body a) až g) popisují postup začlenění ochranného rozhranní z programátorská hlediska.

**a) Příprava datových hodnot**

První částí je příprava následujících hodnot:

- AID „appletu“ – Unikátní identifikace „appletu“ hardwarového tokenu. Tato hodnota je požadována specifikací rozhranní JavaCard a je využita pouze pro výběr aktuálního „appletu“ na fyzickém hardwarovém tokenu. Velikost 5 až 16 bajtů.
- UUID_HWT – Unikátní identifikace hardwarového tokenu v rámci ochranného rozhranní. Tato hodnota je přiřazena během instalace „appletu“ hardwarového tokenu. Velikost 16 bajtů.
- Šifrovací klíče $KE_A$, $KD_A$, $KE_T$, $KD_T$ - Hodnoty klíčů používaných pro autentizaci a utajení dat vyměňovaných mezi hardwarovým tokenem a softwarovým agentem. Tyto hodnoty jsou použity pro generování WBACR AES tabulek a řídících licencí. Velikost 16 bajtů každý klíč.
- Licenční šifrovací klíče - Výchozí hodnota klíče pro řídící licenci je přiřazena během instalace „appletu“. Pomocí řídící licence může být později měněna. Velikost 16 bajtů. Hodnota klíče pro licenci algoritmů je nastavena pomocí řídící licence a může být měněna. Velikost 16 bajtů.

**b) Implementace hardwarového tokenu**

Druhou částí je implementace hardwarového tokenu. Programátor implementuje funkčnost chráněného algoritmu vytvořením třídy implementující rozhranní *SecureAlgProfile.*Programátor modifikuje chování třídy *XMLAlgLicenceParseHandler*, pokud vyžaduje možnost aktualizace vlastních atributů chráněného algoritmu pomocí XML licence. Programátor překládá a konvertuje balík SecureAlg do binární podoby použitelné v čipových kartách s rozhranním JavaCard.

**c) Implementace softwarového agenta**

Třetí částí je implementace softwarového agenta. Programátor generuje potřebné WBACR AES tabulky dle protokolu SEAUT pomocí WBACR AES generátoru. Hodnoty klíčů použité pro generování jsou uschovány. Výsledkem generování jsou hlavičkové soubory obsahující hodnoty WBACR AES tabulek, použité pro překlad třídy *CAESCipher*. Programátor umístí na vhodném místě volání metod třídy *CRemoteAlgManager.* V místech použití chráněného algoritmu je umístěno volání metody ProcessAlgData(). Před prvním voláním metody ProcessAlgData() je umístěno volání metody Connect() a ProcessAuth(). Metoda Connect() vybere fyzický hardwarový token a aktivuje na dle AID požadovaný applet. Metoda ProcessAuth() vytvoří zabezpečený komunikační kanál pro použití chráněného algoritmu.

Pokud softwarový agent umožňuje zasílání licencí, umisťuje programátor volání metod ProcessMasterLicence() a ProcessAlgLicence(). Prvnímu volání musí předcházet volání

metody Connect(). Zabezpečený kanál není vyžadován. Pokud není poskytována licence v ordinálním pořadí, je třeba nastavit implicitní parametr *outOfOrder* na hodnotu TRUE.
Softwarovému agentovi je poskytnuta unikátní identifikace UUID_SWA.
Programátor překládá softwarového agenta do spustitelné podoby.

**d) Instalace hardwarového tokenu**

Čtvrtou částí je instalace hardwarového tokenu. Pomocí vhodné rozhranní podporovaného zvolenou čipovou kartou (Open Plaform) je nahrána binární podoba balíku SecureAlg.
Při instalaci balíku jsou poskytnuty metodě javacard.framework.Applet::install() tyto výchozí hodnoty („user data“):

- DEBUG_MODE (1B) – pokud 0x13, je provedena instalace v ladicím módu.
- UUID_HWT (16B) - unikátní identifikace „virtuálního” hardwarového tokenu.
- MASTER_LICENCE_KEY (16B) – výchozí hodnota šifrovacího klíče řídící licence.

**e) Instalace softwarového agenta**

Pátou částí je instalace softwarového agenta. Pro komunikaci s hardwarovým tokenem je nutná znalost typu čipové karty, AID „appletu“ a UUID_HWT. Tyto hodnoty mohou být součástí konfiguračního souboru distribuovaného se softwarovým agentem, nebo mohou být před kompilací umístěny do kódu softwarového agenta. První způsob poskytuje větší flexibilitu, druhý větší bezpečnost, neboť ztěžuje podvržení hardwarového tokenu útočníkem.

**f) XML licence**

Šestou částí je aktualizace prostředí pomocí XML licencí. Aktualizace může probíhat vzdáleně, utajení a integrita licence je zajištěna šifrovacím klíčem sdíleným mezi hardwarovým tokenem a poskytovatelem softwarového agenta. Čerstvost licence je zajištěna pomocí čítače. Hardwarový token akceptuje pouze takovou licenci, která má vyšší pořadové číslo než naposledy použitá. Pokud poskytnutá licence není bezprostřední následní naposledy použité, je vyžadován speciální příznak příkazu s licencí. Cílem je zabránit náhodnému zpracování licence mimo pořadí, které způsobí nepoužitelnost všech licencí s nižším pořadovým číslem. Zasílání licencí je realizováno pomocí metod CRemoteAlgManager::ProcessMasterLicence() a ProcessAlgLicence().

**g) Využití chráněného algoritmu**

Sedmou částí je vlastní využívání chráněného algoritmu softwarovým agentem. Softwarový agent zvolí dle typu čipové karty, AID a UUID_HWT požadovaný hardwarový token. Využívá volání metody CRemoteAlgManager::Connect().
Softwarový agent poskytuje hardwarovému tokenu nové XML licence. Využívá volání metody CRemoteAlgManager::ProcessMasterLicence() a ProcessAlgLicence().
Softwarový agent iniciuje ustanovení bezpečného komunikačního kanálu. Využívá volání metody CRemoteAlgManager::ProcessAuth().
Softwarový agent využívá chráněný algoritmus umístěný na hardwarovém tokenu prostřednictvím metody CRemoteAlgManager::ProcessAlgData().

Softwarový agent ukončuje spojení s hardwarovým tokenem prostřednictvím metod CRemoteAlgManager::StopAuth() a CRemoteAlgManager::Disconnect().

## B.2 Přehled podporovaných APDU příkazů

| Příkaz | CL | INS | P1 | P2[1] | Vstupní data | Výstupní data |
|---|---|---|---|---|---|---|
| install | 80 | E6 | 0C | 00 | DEBUG_MODE \| UUID_HWT \| MASTER_LICENCE_KEY | N/A |
| select | 00 | A4 | 04 | 00 | AID appletu | N/A |
| ProcessMasterLicence | B0 | 0A | 00[2] | 00 | HWT_UUID \| (MASTER_LICENCE)[3] | N/A |
| ProcessAlgLicence | B0 | 0B | 00[4] | 00 | HWT_UUID \| (ALG_LICENCE)[5] | N/A |
| ProcessAlgData | B0 | 03 | 00 | 00 | HWT_UUID \| SWA_UUID \| (ALG_UUID \| ALG_IN_DATA)[6] | HWT_UUID \| SWA_UUID \| (ALG_OUT_DATA)[7] |
| ProcessSEAUT (krok 1) | B0 | 30 | 01 | 00 | HWT_UUID \| SWA_UUID \| NSWA | UUIDHWT \| UUIDSWA \| (NHWT \| NSWA \| UUIDSWA)[8] |
| ProcessSEAUT (krok 2) | B0 | 30 | 03 | 00 | UUIDHWT \| UUIDSWA \| (NSWA \| NHWT)[9] | N/A |
| StopSEAUT | B0 | 31 | 00 | 00 | UUIDHWT \| UUIDSWA \| („STOP“)[10] | N/A |
| DebugReset[11] | B0 | 40 | 00 | 00 | N/A | N/A |

## B.3 Řídicí XML licence (DTD)

XML parser umístěný v hardwarovém tokenu umožňuje zpracovávat řídící XML licence se strukturou definovanou následujícím DTD:

[1] Lze předávat parametry pro DEBUG MODE.
[2] P1 == 01, pokud není pořadové číslo licence bezprostřední následník předchozí použité.
[3] Šifrováno klíčem pro zabezpečení řídící licence
[4] P1 == 01, pokud není pořadové číslo licence bezprostřední následník předchozí použité.
[5] Šifrováno klíčem pro zabezpečení licence k chráněnému algoritmu
[6] Šifrováno klíči $KE_T$ a $K_R$ dle SEAUT
[7] Šifrováno klíči $KD_T$ a $K_R$ dle SEAUT
[8] Šifrováno klíčem $KD_A$ dle SEAUT
[9] Šifrováno klíčem $KE_A$ dle SEAUT
[10] Šifrováno klíčem $KE_T$ dle SEAUT
[11] Dostupné pouze v DEBUG MODu hardwarového tokenu. Uvede hardwarový token do stavu po install().

```
<!ELEMENT MLic (
     UUIDHWT, LON, MLicKey?, ALicKey?, NewAlg*, ModifyAlg*,
     RemoveAlg*, NewSWA*, ModifySWA*, RemoveSWA*)>
<!ELEMENT NewAlg (UUID, Type)>
<!ELEMENT ModifyAlg (UUID, PermitSWA*, UnpermitSWA*)>
<!ELEMENT RemoveAlg (UUID)>
<!ELEMENT NewSWA (UUID, EKA, DKA, EKT, DKT)>
<!ELEMENT ModifySWA (UUID, EKA?, DKA?, EKT?, DKT?)>
<!ELEMENT RemoveSWA (UUID)>
<!ELEMENT UUIDHWT (#PCDATA)>
<!ELEMENT LON (#PCDATA)>
<!ELEMENT MLicKey (#PCDATA)>
<!ELEMENT ALicKey (#PCDATA)>
<!ELEMENT UUID (#PCDATA)>
<!ELEMENT Type (#PCDATA)>
<!ELEMENT EKA (#PCDATA)>
<!ELEMENT DKA (#PCDATA)>
<!ELEMENT EKT (#PCDATA)>
<!ELEMENT DKT (#PCDATA)>
<!ELEMENT PermitSWA (#PCDATA)>
<!ELEMENT UnpermitSWA (#PCDATA)>
```

## B.4 XML licence algoritmu (DTD)

Ukázkový příklad licence k algoritmu umožňuje zvýšit čítač povolených použití chráněného algoritmu. Struktura licence je definována následujícím DTD:

```
<!ELEMENT ALic (UUIDHWT, UUID, LON, UCount?)>
<!ELEMENT UUIDHWT (#PCDATA)>
<!ELEMENT UUID (#PCDATA)>
<!ELEMENT LON (#PCDATA)>
<!ELEMENT UCount (#PCDATA)>
```

# Příloha C

## C.1 Vybrané klasické techniky ochrany

### C.1.1 Registrační číslo (Serial number / CD Key)

Zadání registračního čísla na výzvu softwarového agenta je v současnosti velmi rozšířená metoda ochrany. Po zadání a ověření korektního registračního čísla je softwarový agent zaregistrován a registrační číslo již nebývá napříště vyžadováno. Registrace umožní softwarového agenta plně používat. Pokud bylo možné softwarového agenta používat v omezené verzi již před registrací, dojde ke zpřístupnění celé funkčnosti. Registrační číslo v sobě může nést informaci, jaká část funkčnosti má být uživateli dostupná. Lze tak odstupňovat úroveň registračního čísla na základě množiny funkcí, které budou jeho zadání dostupné. Registrační číslo je získáno od poskytovatele softwarového agenta zakoupením licence, poskytnutím osobních údajů a podobně.

Při použití této ochrany je třeba chránit před útočníkem hodnoty správných registračních čísel a zajistit integritu ověření zadaného registračního čísla. V závislosti na způsobu použití ochrany se útočník může snažit o získat registrační číslo poskytnuté jinému uživateli, získat hodnotu registračního čísla z kódu softwarového agenta nebo modifikovat softwarového agenta tak, aby zadané chybné registrační číslo vyhodnotil jako korektní.

**a) Registrační číslo je vždy stejné**

Tato verze ochrany registračním číslem používá předem dané registrační číslo společné všem kopiím softwarového agenta. Je snadno implementovatelná a umožňuje distribuci stejných softwarových agentů všem uživatelům. V kódu softwarového agenta není nutné mít uloženou korektní hodnotu registračního čísla, lze využít kryptograficky jednosměrné funkce a uchovávat pouze očekávaný výsledek. Hodnota odvozená z korektního registračního čísla může být použita pro zašifrování částí kódu softwarového agenta. Nevýhodou vždy stejného registračního klíče je jeho snadná přenositelnost mezi uživateli.

**b) Registrační číslo se mění podle zadaných položek**

Tato verze ochrany používá pro ověření korektnosti registračního čísla závislost na dodatečných položkách typu jméno uživatele a jméno firmy. Uživatel nejprve zašle poskytovateli hodnotu těchto položek a obdrží nazpět registrační číslo, které je vygenerováno na základě zaslaných hodnot.

**c) Registrační číslo se mění dle prostředí klienta**

Tato verze ochrany využívá některých charakteristik specifických pro prostředí klienta, například sériová čísla harddisků, síťových karet nebo CPU. Získané charakteristiky mohou být použity pro odvození korektního registračního čísla anebo používány jako výchozí materiál pro tvorbu šifrovacích klíčů nutných pro dešifrování kódu softwarového agenta. Použití této ochrany vyžaduje vhodné řešení v případě, kdy oprávněný uživatel provede změny prostředí, například aktualizaci hardware.

**d) Registrační číslo je kontrolováno v bezpečném prostředí**
Po zadání registračního čísla uživatelem je jeho hodnota zaslána na vyhodnocení do příslušného bezpečného prostředí (on-line server, hardwarový token), často spolu s dalšími údaji (viz. předchozí typy ochran). Softwarovému agentovi je zaslána zpět zprávu obsahující výsledek vyhodnocení. Algoritmus ověření registračního čísla je mimo dosah útočníka. Komunikace s bezpečným prostředím vyžaduje vhodnou autentizaci a zajištění integrity a utajení vyměňovaných zpráv tak, aby útočník nebyl schopen simulovat vůči softwarovému agentovi bezpečné prostředí.

### C.1.2 Registrační soubor („KEY file“)
Ochrana pracuje na principu kontroly přítomnosti korektního registračního souboru. Pokud je takový soubor nalezen, softwarový agent se chová jako registrovaný, v opačném případě pracuje omezeně nebo vůbec. Registrační soubor je rozšířenou variantou registračního čísla, může však obsahovat větší množství použitelných informací. Vhodné využití je umístit do souboru data potřebná pro dešifrování některých částí kódu nebo celý kód některých funkcí, které nejsou bez registrace dostupné. Nevýhodou této ochrany může být potenciální problém distribuce registračního souboru vzhledem k jeho možné velikosti.

### C.1.3 Časové omezení
Ochrana pomocí časového omezení umožňuje uživateli po omezenou dobu vyzkoušet plnou funkčnost softwarového agenta. Po uplynutí příslušného intervalu nelze softwarového agenta používat, případně pouze v omezené podobě. Důležitým bodem je ochrana integrity záznamu o počátku intervalu. Je třeba počítat s užitím monitorovacích prostředků typu RegMon a FileMon [RFMON] umožňujících monitorování přístupů do registrů a souborového systému. Při zjišťování aktuálního času je vhodné použít více nezávislých zdrojů. Varianta časového omezení může povolit pouze omezený počet spuštění softwarového agenta. Zrušení časového omezení lze podmínit zadáním korektního registračního čísla, přítomností registračního souboru a podobně.

### C.1.4 Úmyslné poškození originálního média
Pokud je ke korektnímu běhu softwarového agenta nutná přítomnost originálního (fyzického) média, lze nepovolenému šíření softwarového agenta bránit ztížením tvorby kopie originálního média. Ochrana záměrně poškozuje některé charakteristiky originálního média takovým způsobem, aby bylo obtížné vytvořit identickou kopii. Vzhledem k tomu, že ochrana zároveň nemá bránit v užití oprávněnému uživateli, musí se jednat o typ chyb projevující se pouze při nestandardním užívání. Cílem může být pouze zabránit tvorbě kopií originálního média nebo i vynutit použití pouze v některých zařízeních.

**a) Poškození hlavičky CD**
Mezi záměrná poškození hlavičky CD lze zahrnout chybné označení typu audio a datových stop, chybné počátečních časů jednotlivých stop, umístění začátku první stopy před hodnotu udávanou v specifikaci CDDA jako nejnižší možnou (00:02.00), záznamy o

neexistujících stopách, záznamy o nekorektně uzavřených zápisových sekcí nebo chybné udání velikosti souborů.

**b) Fyzické chyby**
Na médiu jsou umístěny fyzické chyby (i ve značném množství až 20 000), které jsou na umístěny v průběhu vkládání kódu a dat softwarového agenta a jejichž přítomnost výrazně stěžuje kopírování. Přítomnost těchto chyb lze testovat, neboť po vytvoření kopie již přítomné nejsou. Nutnost použít speciální zařízení pro tvorbu chyb výrobu prodražuje.

**c) Softwarové chyby a identifikační znaky**
Tento způsob ochrany před kopírováním nevyžaduje speciální zařízení. Ochrana je založena na vyhledávání záměrně vložených chyb (ne fyzických), například chybné hodnoty opravných kódů simulující fyzické poškození média a jiných identifikačních znaků. Získané údaje jsou často používány pro dešifrování některých částí kódu. Pro překonání ochrany je pak nutno pořídit identickou kopii, což vyžaduje od čtecího zařízení některé speciální vlastnosti.

Jak ale ukazují výsledky testů komerčních ochran založených na poškození originálního média [Ha02], jedná se vždy pouze o řešení využívající aktuální funkčnost a nedostatky používaného hardware a software, které lze v nových verzích příslušně ošetřit.

### C.1.5 Ochrany proti nástrojům pro dynamickou inspekci
Pro ochranu před útočníkem provádějícím dynamickou inspekci pomocí krokování lze využít ochranných technik založených na detekci přítomnosti nástroje, který krokování umožňuje (debugger). Ve většině případů není přítomnost debuggeru nutná a signalizuje pokus o dynamickou inspekci.

Detekce debuggerů může být založena na obecných vlastnostech, které tyto nástroje mají, nebo může být zaměřena speciálně proti nejčastěji používaným (Soft-ICE, TRW). V druhém případě nedojde k odhalení všech debuggerů, ale investice do vývoje vlastního nástroje srovnatelného s uvedenými výrazně zvyšuje obtížnost útoku. Popis konkrétních technik lze nalézt například v [Ce02, Ze02].

### C.1.6 Ochrana proti monitorovacích nástrojům
Pokud softwarový agent ukládá citlivé informace do souborů nebo registrů, je vhodné bránit použití monitorovacích nástrojů typu Regmon (monitorování přístupu k MS Windows registrům) a Filemon (monitorování přístupu k souborovému systému) [RFMON]. Pro jejich zjištění platí podobné podmínky, jako pro ochranu proti debuggerů, konkrétní techniky lze opět nalézt v [Ce02, Ze02].

## C.2 Základní metody obfuskace

Výběr základních obfuskačních metod je převzat ze zdrojů [CTL97, CGJZ01, ClkwTr02, NCJ01, CTL98].

### C.2.1 Výpočetní transformace

**a) Vkládání mrtvého nebo irelevantního kódu**

Do kódu agenta je vložen kód, který nemá žádný vliv na diagram toku dat ani na hodnotu použitých proměnných. Může být realizováno pomocí vkládání tautologických podmínek, jejichž vyhodnocení je nezávislé na hodnotách vstupních proměnných, vkládáním irelevantních podmínek, u kterých je korektní kód vykonán bez ohledu na jejich vyhodnocení, neboť podmíněné větve realizují stejnou operaci nebo vkládáním podmínek, které vždy vyberou podmíněnou větev s korektním chováním, přičemž ostatní větve provedou chybnou operaci.

**b) Rozšíření podmínek skoků o tautologie**

Vložením tautologických výrazů (vždy pravda) nemá vliv na výsledné vyhodnocení podmínky skoku a pouze tedy zvyšuje obtížnost při tvorbě mentálního modelu (a samozřejmě i dobu vyhodnocení podmínky.) Spolu s vývojem technik generování tautologií se zároveň vyvíjí i techniky jejich detekce a je proto vhodné, aby tautologie nebyla snadno odhalitelná při statické inspekci kódu. Měla by tedy obsahovat co nejvíce závislostí na aktuálních hodnotách proměnných s co největším stupněm zanoření.

**c) Konverze do ireducibilního diagramu toku dat**

V případě, že jazyk do kterého je zdrojový kód překládán obsahuje konstrukce, které nemají přímou korespondenci v původním jazyku, je nutno při dekompilaci provádět syntetizaci původního kódu z konstrukcí, které původní jazyk nabízí a vzniká tak možnost ztížit nebo zmást tuto syntetizaci tak, aby výsledný dekompilovaný kód poskytoval nepřesné a těžko interpretovatelné výrazy.

**d) Nahrazení volání knihovních funkcí přímou funkcionalitou kódu**

Pro zabránění útoku pomocí podvržení návratových hodnot knihovních se kód, který byl normálně umístěn do knihovny vloží přímo do kódu programu. Výsledkem je vyšší náročnost vytvoření mentální modelu daného kódu, neboť je ztížena identifikace pozice, na které je vložený kód prováděn. Negativním dopadem je nárůst velikosti kódu programu.

**e) Odstranění standardně používaných programovacích vzorů**

Mezi dobrý programátorský zvyk patří používání co nejstandardnějšího způsobu prováděný často se vyskytujících akcí, neboť výrazně zvyšují pochopitelnost celého kódu pro další programátory a po jisté době i pro původního tvůrce. Tento zvyk je ale bohužel v přímém rozporu s potřebami obfuskace, tj. snahy o co nejobtížnější pochopitelnost výsledného kódu. Je tedy vhodné nepoužívat standardní programátorské vzory a principiálně stejné operace by měly být prováděny pokud možno pokaždé jiným způsobem.

**f) Nahrazení standardního interpretu mezikódu vlastním interpretem**
Části kódu programu nejsou prováděny standardně (nativní kód, standardní interpret), ale jejich obsah je interpretován pomocí vlastního interpretu obsaženého také v kódu programu. Dochází k výraznému zvýšení obtížnosti pochopení, neboť útočník musí provést analýzu neznámého interpretu často bez možnosti použít standardních prostředků určených pro běžné interprety či nativní kód, zároveň ale dojde ke zvýšení výkonnostních požadavků.

**g) Zavedení redundantních operací**
Do kódu programu jsou přidány operace, jejichž výkon pouze duplikuje výsledky již provedených operací nebo výsledná hodnota operace je shodná se vstupní hodnotou. Pokud útočník nemůže předem odhadnout, zda daná operace provádí konstruktivní změnu vstupních dat, je nucen ji během dynamické inspekce procházet a ztěžuje mu tvorbu mentálního modelu.

**h) Paralelizace kódu**
Pro ztížení dynamické inspekce lze výkon sekvenčních, datově nezávislých částí kódu programu rozdělit do paralelně vykonávaných procesů, v případě částečné datové závislosti pak provádět synchronizaci pomocí synchronizačních primitiv. Zároveň lze zavádět irelevantní procesy, jejichž obsahem je irelevantní kód se stejným významem jako v případě vkládání irelevantního kódu.

**i) Agregační transformace**
Kód, který k sobě logicky přísluší a je standardně umístěn do jednoho funkčního bloku je rozdělen na části a rozmístěn způsobem, který zachová sémantiku původního kódu. Části kódu které k sobě logicky nepřísluší jsou naopak uměle agregovány do jednoho funkčního bloku.

Využitím „inline“ metod lze vkládat těla metod přímo do kódu namísto volání a ztížit tak jejich lokalizaci. „Outline“ metody poskytují možnost umělého umístění části sekvenčního kódu do nově vytvořené metody, která však nemá logickou spojitost s diagramem toku dat. Pro ztížení lokalizace volání citlivé metody lze použít vícevýznamové metody spojením kódu více metod do jedné, v níž je výkon příslušného kusu původního kód rozlišen na základě vstupních parametrů metody. Volání takové vícevýznamové metody je pak obsaženo i na mnoha místech kódu, které nesouvisí s voláním citlivé části metody. Další transformací pro ztížení lokalizace volání citlivé metody mohou být klonované metody, které uměle zavedení více metod vykonávajících stejnou operaci jako metoda původní. Irelevantní výběr konkrétní metody může být prováděn dynamicky za běhu.

Pro kód obsahující programové cykly lze použít několik možností jejich transformace na obtížněji pochopitelná části kódu. Pomocí blokování cyklů dosáhneme rozdělení původního cyklu na sérii menších, vnořených cyklů. Opačnou transformací je rozbalení („unrolling“) cyklů, následkem čehož je původní cyklus zcela nebo částečné nahrazen sekvenčním kódem obsahujícím příslušná opakování těla cyklu. Pokud jsou

v těle cyklu vykonávány nezávislé operace, lze původní cyklus štěpením cyklů nahradit sérii cyklů provádějících vždy jen jednu z nezávislých operací
Pomocí transformace řazení operací v kódu lze dosáhnout distribuce logicky souvisejících operací co nejdále od sebe při zachování původní funkčnosti.

### C.2.2 Datové transformace

**a) Změna kódování dat**

Zavedením vhodného kódování uchovávaných dat, které je před jejich použitím odstraněno, lze ztížit vyhledávání pozice těchto dat a jejich nahrazování. Zavedení dynamického kódování předchozí operací, které je následující operací odstraněno lze dosáhnout obtížnější dynamické inspekce dat předávaných dat. Kódování může být závislé na parametrech okolního prostředí.

**b) Rozdělení resp. spojení proměnných**

Rozdělením resp. spojením původních proměnných do jiných paměťových uskupení a příslušným upravením původních operací nad těmito proměnnými lze dosáhnout obtížnější automatické tvorby diagramu toku a zanést do kódu uměle vyšší stupeň vzájemné závislosti proměnných.

**c) Konverze statických struktur do dynamicky generovaných**

Namísto přítomnosti statických dat je do kódu vložena operace, která generování jejich hodnoty provede dynamicky za běhu, čímž ztěžuje statické inspekci kódu.

**d) Restrukturalizace polí**

Pro úpravu paměťových polí do jiných paměťových uskupení lze použít několik transformací spolu s příslušnou úpravou operací pracujících nad těmito poli. Lze provést spojení více polí do jednoho celku, rozdělení jednoho pole na více částí ve formě polí nebo jednoduchých typů proměnných. Dimenzi pole, která logicky odpovídá oblasti použití pole, lze zvýšit resp. snížit a pole příslušně reorganizovat. Prvky pole mohou být permutovány předchozí sekvencí operací, následující operace zavádí před jejich použitím inverzní permutaci.

**e) Modifikace vztahů dědičnosti mezi třídami**

Pro ztížení pochopení modelu OOP lze zavést rozdělení původní třídy na více tříd, které obsahující vždy jen část funkcionality původní třídy nebo zavedení nové rodičovské třídy pro dvě třídy, které nemají společnou funkcionalitu dosáhneme

**f) Preventivní transformace**

Cílem těchto transformací je využití konkrétních vědomostí o používaných obsfukačních technikách a známých obsfukačních programech pro konkrétní ovlivnění jejich chování nebo ztížení procesu automatického rozlišování relevantního kódu od irelevantního. Cílem tedy není primárně lidský útočník, ale známé vlastnosti používaných automatických nástrojů.

**g) Transformace zaměřené proti známým deobsfukačním technikám**

Přidání formálních parametrů referencujících stejný paměťový prostor slouží ke zvýšení paměťových nároků pro automatické určení kusů kódu, které se podílejí na modifikaci hodnoty proměnné umístěné na daném paměťovém prostoru. Stejného efektu lze dosáhnout umělým zvýšením počtu možných operací, které by mohly ovlivnit hodnotu proměnné, ale nikdy tak neučiní.

## C.3 Sebekontrolující kód

### C.3.1 Požadovaná funkcionalita

**a) Vyčerpávající a včasná detekce změny**

Mechanismus by měl detekovat změnu i jediného bitu kódu a co nejdříve po provedení změny. Cílem je zamezení útoků využívajících dočasnou změnu kódu jen po dobu vykonání škodlivé činnosti (dočasné odstranění ochrany), po které je kód vrácen do původního stavu.

**b) Oddělená a flexibilní reakce na detekci**

Oddělení reakčního mechanismu od detekčního umožní vhodnější přizpůsobení reakce aktuální situaci. Zároveň činí detekční a reakční mechanismus obtížněji rozpoznatelný i po nalezení jedné z částí.

**c) Modulární komponenty**

Komponenty mechanismu jsou modulární a nezávislé, díky čemuž mohou být snadno nahrazeny a modifikovány při budoucím vývoji a přechodu na nové formáty dat.

**d) Platformová nezávislost**

Mechanismus by neměl využívat principy specifické pro konkrétní výpočetní platformu a poskytovat tak možnost rozšíření i na další platformy.

**e) Akceptovatelné snížení výkonu**

Mechanismus by neměl způsobovat znatelné výkonnostní zpomalení ani výrazné zvětšení původního kódu.

**f) Snadné začlenění**

Mechanismus by měl být schopen pracovat i při přítomnosti dalších ochranných mechanismů (statický watermarking, customization).

**g) Použitelnost pro volitelnou velikost kódu**

Mechanismus by měl být použitelný pro zabezpečení jak malých (kB), tak velkých (MB) programů.

### C.3.2 Testery

Úkolem testerů je vypočítat integritní součet z přiřazené testovací oblasti, porovnat se správnou hodnotou a v případě rozdílnosti hodnot spustit reakční mechanismus. Je potřeba učinit kompromis mezi vlivem testerů na výkon, bezpečností a utajením

provádění testerů. Experimentálně bylo zjištěno [HMST01], že doba výpočtu funkce integritního součtu z daného intervalu je relativně neměnná do doby, dokud velikost testované oblasti nepřesáhne velikost L2 cache. Velikost testované oblasti je tedy vhodné stanovit na několik stovek kB.

Každý tester může počítat integritní součet buď celého přiděleného úseku [HMST01], nebo pouze jeho části a výsledná hodnota je pak získána složením mezivýsledků [ChaAt01]. Druhá možnost vyžaduje navíc uchovávání mezivýsledků, a celkově zvyšuje velikost kódu testerů (a tím usnadňuje nalezení), takže se jeví jako méně vhodná.

Pro snadný výpočet hodnot korektorů během instalace je důležitá linearita funkce, použití více typů funkcí navíc zajistí, že z pouhá znalost intervalu a pozice korektoru se ještě neposkytuje dostatek informace pro úspěšnou změnu korektoru kompenzující útočníkovu změnu. Navíc musí být funkce integritního součtu invertovatelná, neboť je třeba odvodit hodnotu korektoru ze znalosti testovací oblasti a požadované výsledné hodnoty integritního součtu a měla by být i co nejjednodušší z důvodu požadavku na co nejmenší velikost testeru.

Pro tvorbu malých a nenápadných testerů je vhodné testery napsat v jazyku C, přeložit do assembleru, následně upravit obfuskací adres čtení z kódové oblasti a pokud možno zjednodušit. Pro zajištění odolnosti proti srovnávacímu útoku je třeba vytvořit co nejvíce odlišných implementací testerů, např. použití různých funkcí pro výpočet integritního součtu, permutace základních bloků testeru, inverze logiky podmíněných skoků, permutace použitých registrů apod.

Testery mohou být vkládání do zdrojového kódu [HMST01] nebo objektového kódu [ChaAt01]. Při vkládání do zdrojového kódu umožňuje vložení do míst s odhadnutelnou výkonnostní charakteristikou a je snáze zajistitelná bezkonfliktnost použití registrů původním kódem a kódem testeru. Vkládání do objektového kódu však umožnilo lepší statistickou distribuci testerů.

### C.3.2.1 Útoky na tester

**a) Detekce čtení z oblasti kódu**

Pro výpočet integritních součtů je třeba přistupovat do oblastí s kódem programu, což je netypická a detekovatelná operace (lze prohlížet obsah standardních registrů na výskyt hodnoty adresy kódového segmentu programu, statická inspekce kódu). Obranou může být obfuskace čtecích instrukcí tak, aby se hodnota cílové adresy nikdy nevyskytla v jednom registru.

**b) Generalizace**

Pokud je v kódu umístěno více podobných kontrolních mechanismů, může útočník po odkrytí jednoho hledat ostatní pomocí podobné kódové sekvence. Mechanismy by tedy měli mít rozdílnou reprezentaci, čehož lze docílit využitím rozdílných technik použitých jednotlivými kontrolními mechanismy (různé funkce použité pro výpočet integritních součtů…) a personalizací jejich spustitelných representací.

**c) Srovnávací útok**

Pro odhalení polohy ochranných mechanismů lze využít více kopií daného programu pro nalezení odlišných částí kódu.

**d) Inspekce instalačních záplat**
Při použití statického watermarkingu a sebekontrolujícího kódu je třeba použít během instalace záplatu, která učiní z nefunkčního spustitelného kódu po provedení watermarkingu kód funkční. Inspekce této záplaty by neměla vést k odkrytí kontrolních a reakčních mechanismů.

### C.3.2.2 Uchování korektních hodnot integritních součtů testovaných oblastí

Tester provede výpočet hodnoty integritního součtu a porovná s očekávanou korektní hodnotou. Pro uchovávání korektní hodnoty integritního součtu pro daný tester a jemu příslušnému intervalu kódu lze zvolit několik možných přístupů, lišící se co do obtížnosti umístění a modifikace hodnot a vztahu k poloze testeru. Je vhodné, aby hodnota poskytovala co nejméně informací o poloze testeru.

**a) Kód testeru**
Tato varianta skýtá nebezpečí odhalení polohy testerů pomocí nalezení uchovávaných hodnot integritních součtů srovnávacím útokem (pokud je použit watermarking, příslušné hodnoty se budou u různých kopií lišit). Navíc může být obtížné vypočítat korektní hodnoty v případě, že se oblasti testerů překrývají a vložením jedné hodnoty se kruhově poruší korektnost hodnota vypočtené dříve pro jiný tester. Tato varianta je použita v [ChaAt01].

**b) Datová sekce**
Tato varianta sice řeší oba předešlé problémy, ale neposkytuje ochranu proti modifikaci, neboť datová sekce spustitelného kódu není kontrolována.

**c) Tester si korektní hodnotu neuchovává**
Spustitelný kód je během instalace upraven pomocí *korektoru* tak, aby integritní dával předem známou hodnotu (nula) – do každé testovací oblasti se vloží hodnota (např. 32 b dvojslovo), které je během instalace nastaveno na korektní hodnotu. Pomocí srovnávacího útoku lze sice odhalit polohu korektorů, ale umístění korektorů nemá přímý vztah k testeru. Korektory se navíc nachází v testované oblasti a jejich změna je tudíž obtížná. Korektory jsou vkládány pomocí post-kompilační manipulace mezi bloky základního kódu (mrtvý kód, kam nikdy nebude předáno řízení programu). Další možností je korektory vkládat místo NOP operací.

### C.3.3 Reakční mechanismus

Úkolem reakčního mechanismu je provést vhodnou akci při detekci změny kódu programu. Může se jednat o ukončení programu spolu s hlášením uživateli, ale vhodnější z hlediska utajení polohy reakčního mechanismu je provést sérii operací vedoucí k chybě a pádu programu. Program však může sám sebe opravit do původní podoby pomocí zálohy, jak je navrhováno v [HMST01].

Vlastní zpuštění reakčního mechanismu může být realizovat různými způsoby, např. normálním během programu poté, co tester nastaví hodnoty proměnných ovlivňující tok programu, funkčním voláním, nezávislým testovacím vláknem nebo vyvoláním výjimky. Důraz by měl být kladen především na nenápadnost takové akce, neboť může prozradit polohu testeru a reakčního mechanismu.

### C.3.3.1 Útoky na reakční mechanismus

**a) Modifikace testerů**

Cílem útoku je modifikace jednoho nebo více testerů takovým způsobem, aby nereagoval správně na detekci změny integritního součtu. Pokud se testované oblasti testerů redundantně překrývají, je změna jednoho z nich detekována nejméně jedním dalším. Pro aplikaci takového útoku je potom nutné nalézt a modifikovat většinu testerů.

**b) Modifikace reakčního mechanismu**

Cílem útoku je modifikace reakčního mechanismu způsobem, který dovolí pokračování běhu programu i přes detekci změny. Reakční mechanismy jsou ale opět redundantně chráněny testovacími oblastmi testerů. Při použití většího množství reakčních mechanismů (jeden tester jeden mechanismus) je opět třeba nalézt a modifikovat většinu reakčních testerů.

**c) Modifikace korektorů**

Cílem útoku je modifikace kódu takovým způsobem, který nezpůsobí detekci změny testerem (integritní součet se nezmění). Díky redundantnímu překryvu testovaných oblastí toho opět lze docílit až odkrytím většiny kontrolních mechanismů, čímž se problém redukuje na problém modifikace testerů.

**d) Dočasná modifikace**

Cílem útoku je provést změnu kódu jenom dočasně na dobu nutnou pro vykonání nežádoucího chování a poté obnovit původní kód a zabránit tak detekci změny. Dostatečně časté vykonávání výpočtu integritních součtů brání tomuto útoku. Pro dostatečně časté vykonávaní testerů je potřeba nejen jejich dostatečné množství v kódu, ale zároveň i jejich rovnoměrné rozložení. Rovnoměrné rozložení lze zajistit s použitím profilovacích nástrojů a náhodnou permutací základních bloků s již umístěnými testery.

## C.4 White-Box Attack Resistant AES

Praktická implementace vychází ze způsobu urychlení výkonu AES pomocí předpočtených tabulek, jak je navrženo v [RiDa99]. Po sobě jdoucí operace v rámci jedné rundy jsou realizovány pomocí předpočtené tabulky, která obsahuje v závislosti na hodnotě vstupního bloku dat výsledné hodnoty po provedení operací SubByte() a části operace MixColumn(). Po provedení tohoto náhledu již zbývá pouze provést operace odpovídající použití funkcí AddRoundKey(), ShiftRow() a zbývající části operace MixColumn(). Výsledkem je snížení počtu operací nutných pro realizaci jedné rundy na 4 náhledy do předpočtené tabulky a 4 operace XOR (8bitové) na jeden sloupec vstupního bloku dat. Tuto optimalizaci budu nazývat $AES_{OPT.}$

WBACR AES využívá a rozpracovává výše uvedenou optimalizaci $AES_{OPT}$. Vhodnou úpravou hranic jednotlivých rund lze dosáhnout toho, že všechny rundu kromě finální se skládají ze sekvence operací AddRoundKey(), ByteSub(), ShiftRow() a MixColumn(). Finální runda pak má tvar AddRoundKey(), ByteSub(), ShiftRow() a AddRoundKey(). Tabulky WBACR AES jsou na rozdíl od výše uvedené optimalizace vytvářeny pouze pro konkrétní klíč, a lze tedy operaci ByteSub() pro danou rundu transformovat tak, aby výsledná hodnota již obsahovala i XOR s odpovídající hodnotou expandovaného šifrovacího klíče, tedy operaci AddRoundKey().

Získané tabulky však zatím neposkytují žádnou ochranu klíči, který je v nich uschován, neboť z existujících tabulek lze inverzním postupem k jejich tvorbě hodnotu klíče snadno extrahovat. Útočníkovi pro extrakci stačí zvolit náhodný vstupní blok a opatřit si všeobecně známou tabulku S-box, která definuje substituci prováděnou operací SubByte(). Pomocí inverzní tabulky k S-box a hodnoty získané náhledem do předpočtených tabulek pro zvolený vstupní blok pro první rundu, lze XOR operací získat hodnotu expandovaného klíče pro první rundu. Opakováním tohoto postupu lze rekonstruovat celý expandovaný klíč[1].

Je dobré si povšimnout, že optimalizace $AES_{OPT}$ vytváří pouze jedinou tabulku společnou pro všechny rundy, naproti tomu WBACR AES musí vytvořit zvláštní sadu tabulek pro každou rundu, neboť každá tabulka obsahuje expandovaný klíč odpovídající konkrétní rundě, díky čemuž ji nelze použít pro rundu jinou, neboť používá během operace AddRoundKey() jiný klíč.

Klíčovým místem, které umožní útočníkovi získat klíč je možnost pozorovat vstupy a výstupy už na úrovni tabulky pro jednu rundu. Spolu se znalostí konstrukce této jednotlivé tabulky je snadné klíč získat. V případě, že by útočník odkázán pouze na pozorování vstupů a výstupů celého šifrovacího algoritmu, tak by byl ve stejné pozici, jako kdyby mu někdo poskytl převodní tabulku, která by prováděla šifrování daným klíčem na jediný náhled. Tato tabulka reprezentující bijekci mezi vstupním, a výstupním blokem dat by měla velikost $2^{128}$ x 128 bitů. Pokud by útočník neodhalil nějakou slabinu v konstrukci této tabulky[2], byl by zisk hodnoty klíče výpočetně neproveditelný. Velikost takové tabulky ale brání jejímu praktickému použití. Pokud je však šifrování realizováno více náhledy, útočník má možnost získat výstupní hodnoty těchto náhledů a použít je pro útok na oddělené tabulky. Optimalizace $AES_{OPT}$ se navíc skládá ze střídání náhledů do tabulek a standardních aplikací operace XOR a jako taková vyžaduje otevřený vystup na úrovni každé rundy, aby bylo možno provést operaci XOR.

WBACR AES přidává dvě vlastnosti, jejichž cílem je odstranit výše uvedený nedostatek, tedy možnost pozorovat otevřený výstup z dané tabulky:

**a) Operace XOR je nahrazena předpočtenou tabulkou**

Díky této úpravě se šifrování pomocí WBACR AES skládá pouze z náhledů do předpočtených tabulek. Odpadá tak nutno poskytnout standardní operaci XOR vstup v otevřené podobě. Jak zabezpečit výsledky náhledů mezi jednotlivými tabulkami popisuje druhá vlastnost. Utajení klíče v tabulkách dosáhneme teprve její aplikací.

[1] Pro jednoduchost zde není uveden vliv části operace MixColumn(), která je v předpočtené tabulce též zahrnuta, neboť na utajení klíče nemá vliv.

[2] Což by byla zároveň slabina v algoritmu AES

**b) Výsledky náhledů do tabulek jsou chráněny interním kódováním**

Pro zajištění, že výstup náhledu do tabulky nebude útočníkovi dostupný v otevřené podobě, provádí tabulka $T_x$ kromě výše uvedených operací je i výstupní kódování. Toto výstupní kódování může být realizováno například náhodnou bijekcí G pro zvolenou velikost výstupních dat. Tabulka $T_{x+1}$, jejíž použití při šifrování následuje bezprostředně po tabulce $T_x$ provede jako první operaci nad vstupem aplikaci vstupního kódování $G^{-1}$, tedy inverzi k bijekci G. Výstup z tabulky $T_{x+1}$ je opět zakódován pomocí výstupního kódování H, které se může lišit od kódování G a které je opět odstraněno vstupním kódováním $H^{-1}$ tabulky $T_{x+2}$. Pokud je vstupní kódování pro první tabulku a výstupní pro poslední rovno identické bijekci, je vstupem nezašifrovaný blok dat a výstupem jeho zašifrovaná podoba pomocí klíče obsaženého v tabulkách. Útočníkovi tedy již pro extrakci klíče nestačí získat vstup a výstup dané tabulky, musí navíc získat podobu náhodné bijekce G resp. H. Z konstrukce kódování je zřejmé, že jeho podoba nemá vliv na výsledek šifrování, kódování se vždy v následujícím kroku odstraní.

Pro praktickou konstrukci WBACR AES tabulek je nejprve třeba vytvořit tzv. *mapu vstupně/výstupních kódování*, tedy pro každou tabulku T zjistit, jaké u ní má být použito vstupní a jaké výstupní kódování. Do množiny tabulek $T_{\{x\}}$ jsou zařazeny všechny tabulky, které mohou být použity pro náhled bezprostředně po tabulce $T_x$. Výstupní kódování tabulky $T_x$ pak musí být vstupním kódováním všech tabulek z množiny $T_{(x)}$ a naopak pokud náhled do tabulky $T_y$ bude bezprostředně předcházet náhledu do kterékoli tabulky z množiny $T_{\{x\}}$, musí pak mít i výstupní kódování tabulky $T_y$ shodné se vstupním kódováním tabulek $T_{\{x\}}$. Velikost kódování nemusí být nutně rovna velikosti výstupní hodnoty náhledové tabulky, výstup lze rozdělit na části a ty kódovat nezávislými bijekcemi menší velikosti. Musí být ale stále zajištěno, že na odpovídající část vstupu bude použito vstupní kódování odpovídající výstupnímu kódování, jaké bylo použito u předchozí tabulky. Počet kódování tedy bude roven nejvýše počtu tabulek násobeno podílem mezi velikostí vstupu/výstupu každé tabulky a velikostí kódování[3], většinou však bude různých kódování méně.

Po určení *mapy kódování* se provede generování tabulek způsobem uvedeným výše a navíc se přidá pro každou tabulkovou hodnotu výstupní resp. vstupní kódování pro danou tabulku dle *mapy kódování*. Z důvodů praktického omezení velikosti výsledných tabulek je nutno provést úpravy realizací některých operací, čímž se za cenu většího počtu náhledů dosáhne snížení celkové velikosti tabulek. Dále je nutno zvolit kompromis mezi velikostí vstupu do jednotlivých tabulek a bezpečností interního kódování. Zvyšování velikosti vstupu do tabulky zvyšuje exponenciálně prostor možných kódování, ale zároveň narůstá i celková velikost výsledné tabulky. Pro tabulku se vstupem o velikosti m bitů a výstupem o velikosti n bitů lze použít vstupní kódování až do velikosti $2^m$, výsledná velikost tabulky bude $2^m$ x n bitů. Tabulka se vstupem o poloviční velikosti bude moci vstupní kódování pouze o velikosti $2^{m/2}$, její celková velikost se ale sníží $2^{m/2}$ krát. Konkrétně pro realizaci operace, která přijímá vstup o velikosti 16 bitů a produkuje výstup o stejné velikosti, můžeme použít buď jeden náhled do tabulky o celkové velikosti 131 kB, u které bude muset útočník při útoku hrubou silou vyzkoušet nejhůře 65536

[3] Jen v případě, že by každá tabulka měla pouze jedinou možnou předchůdkyni.

možných kódování, nebo můžeme provést dva náhledy do tabulek o celkové velikosti 512 B[4], počet možných kódování se však sníží na 2 x 256. Nutným předpokladem pro takovou transformaci je rozložitelnost původní operace na více operací s menšími velikostmi vstupu, jejichž složením získáme výsledek původní operace.

### C.4.1 Konstrukce tabulek

Operace SubByte() a AddRoundKey() pracují nativně s prvky matice o velikosti 8 bitů, výsledná tabulka pro náhled pro prvek na konkrétní pozici bude mít tedy 8 bitový vstup a 8 bitový výstup a rozumnou velikost 256 bajtů každá.

Operace ShiftRow() nemění hodnotu prvků matice, mění pouze jejich polohu a není tedy nutné vytvářet zvláštní pro její aplikaci tabulky. Pouze je potřeba při konstrukci mapy kódování zajistit, aby vstupní kódování operace nad prvkem s konkrétní polohou v matici, která bude následovat po operaci ShiftRow() odpovídalo výstupnímu kódování aplikovanému na tento prvek před jeho posunem operací, která bezprostředně předchází operaci ShiftRow().

Operace MixColumn() pracuje s celým sloupec matice, tedy 32 bitovým vstupem i výstupem. Realizace jednou tabulkou by znamenala tabulku o velikosti přes 17 GB. Maticové násobení lze však v tomto případě rozepsat jako sérii násobení jednotlivých prvků sloupce a operací XOR. Pro každý prvek sloupce je tedy generována tabulka s 8 bitovým vstupem i výstupem a tabulky provádějící složení mezivýsledků pomocí operace XOR. Tabulky provádějící násobení lze složit s tabulkami provádějícími SubByte() a AddRoundKey(), neboť obě pracují nad 8 bitovým vstupem i výstupem a pozice prvku nad kterým bude pracovat operace MixColumn() po provedení ShiftRow() lze určit.

Z paměťových důvodů[5] provádí XOR tabulky pouze 4 bitovou operaci XOR, takže je nutno operaci XOR dvou 8 bitových argumentů rozložit na náhled do tabulky pro horní 4 bity obou argumentů a dolní 4 bity obou argumentů. Výsledek je zřetězen do výsledného 8 bitového výstupu. Tabulka pro provádění XOR operací neobsahuje interně žádnou závislost na klíči nebo poloze prvku v matici a mohla by být pouze jedna pro všechny potřebné operace XOR. Výstupní kódování všech ostatních tabulek předcházejících operaci XOR by však muselo být shodné se vstupním kódováním této tabulky, čímž by se razantně snížil celkový počet interních kódování na pouhá čtyři různá. Pro útočníka by pak nebyl problém všechna možná kódování projít a odstranit. Proto je vhodné generovat XOR tabulku pro každé možné výstupní kódování předchozích tabulek a udržet tak počet použitých kódování na maximu. Při konstrukci tabulek je přidáno příslušné vstupní a výstupní interní kódování dle *mapy kódování.*

### C.4.2 Velikost interního kódování

Z nutnosti použití XOR tabulek, které provádí pouze 4 bitový XOR a jejichž 8 bitový vstup se skládá ze dvou 4 bitových částí s rozdílným vstupním kódování, vyplývá požadavek, že i výstupní kódování operace předcházející použití XOR tabulek musí být 4 bitové. Pokud má některá tabulka výstup větší než 4 bity, je tento výstup rozdělen na 4 bitové části a pro každou z nich použito jiné 4 bitové kódování

[4] 512 B = 2 x $2^8$ x 1; dvě tabulky se vstupem a výstupem o velikosti 8 bitů
[5] Operace XOR pro dva 8 bitové argumenty vyžaduje tabulku o velikosti $2^{16}$ x 1 bajtů

### C.4.3 Celková velikost WBACR AES tabulek

Tím je generování tabulek hotovo. Pro režim šifrování pomocí WBACR AES v režimu 128 bitů klíč a 128 bitů blok jsou vytvořeny tabulky tří typů o celkové velikosti 262 144 bajtů. Tabulky pro režim dešifrování mají obdobnou strukturu a stejnou velikost. Prvním typem je 144 tabulek[6] obsahující předpočtený výsledek operací AddRoundKey, SubByte a části operace MixColumn (ASMT tabulky) s celkovou velikostí 147 456 bajtů[7]. Druhým typem je 864 tabulek[8] obsahující předpočtený výsledek operací XOR (XT tabulky) nutné pro dokončení operace MixColumn o celkové velikosti 110 592 bajtů[9]. Třetím typem je 16 tabulek[10] pro operace závěrečné rundy obsahující předpočtený výsledek operací AddRoundKey, SubByte a závěrečný AddRoundKey (ASAT tabulky) s celkovou velikostí 4096 bajtů[11].

Celková velikost tabulek pro šifrování je 262 144 bajtů a pro konstrukci šifrovacích tabulek je použito 2016 interních kódování realizovaných formou náhodné 4 bitové bijekce. Analogicky pro operaci dešifrování.

## C.5 Neinformovaný agent

### C.5.1 Možnosti konstrukce šifrovacích klíčů

Nechť N je hodnota odpovídající aktuální charakteristice prostředí, H je jednocestná hash funkce, M je hodnota hashe H z data N nutná pro aktivaci, K je šifrovací klíč použitý pro zabezpečení citlivých informací a R je keksík. Hodnota M je nesena agentem.
Možné podmínky:

- pokud H(N) == M pak K := N
- pokud H(H(N)) == M pak K := H(N)
- pokud $H(N_i) == M_i$ pak K := $H(N_1, \ldots, N_i)$
- pokud H(N) == K := $H(R_1, N)$ XOR $R_2$

### C.5.2 Agent pro neinformované hledání

Cílem je prověřit výskyt určitých dat v prohledávaném prostoru bez vyjevení hodnoty hledaných dat.
Příprava:

1. N := náhodný keksík
2. K := H(“hledaná data D“)
3. M := $E_K$(“operace provedená při nalezení data D“)
4. O := H(N XOR “hledaná data D“)

Metoda prohledávání charakteristik:
**for** (data (x) o délce ”hledaná data D”) **in** (prohledávaný prostor) **do**

[6] Počet běžných run x Počet řádků *State* matice x Počet sloupců *State* matice = 9 x 4 x 4 = 144
[7] Počet tabulek x Velikost tabulky s 8 bitovým vstupem a 32 bitovým výstupem = 144 x $2^8$ x 4 = 147 456 B
[8] Počet běžných run x Počet sloupců *State* matice x Počet operací XOR nutných pro sečtení čtyř 32 bitových sloupců po 4 bitových částech = 9 x 4 x (3 x 2 x 4) = 864
[9] Počet tabulek x Velikost tabulky s 8 bitovým vstupem a 4 bitovým výstupem = 864 x $2^8$ / 2 = 110 592 B
[10] Počet řádků *State* matice x Počet sloupců *State* matice = 4 x 4 = 16
[11] Počet tabulek x Velikost tabulky s 8 bitovým vstupem a 8 bitovým výstupem = 16 x $2^8$ = 4096 B

**if** H(N XOR (x)) == O **then execute** ($D_{H(x)}$(M))

### C.5.3 Agent využívající časové charakteristiky poskytované důvěryhodným serverem

1. Programátor zašle cílový čas $T^*$ a keksík R časovému serveru.
2. Server nastaví T na aktuální čas a vrátí programátorovi T a H(H(S, $T^*$)), H(R, T).
3. Programátor nastaví P := H(R, T) a K := H(H(S, $T^*$)), H(R, T).
4. Klíč K je použit pro šifrování citlivých informací agenta, agent nese hodnotu P.
5. Agent pravidelně žádá aktuální tajemství závislé na aktuálním čase od serveru.
6. Server vrací $S_i$ := H(S, $T_i$), případně tuto informaci veřejně šíří a agent ji pravidelně získává.
7. Agent se pokouší odvodit klíč K := H($S_i$, P) pro dešifrování instrukcí. Uspěje ve chvíli, kdy $S_i$ == H(S, $T^*$) což je tehdy když platí $T_i$ == $T^*$.

### C.5.4 Agent využívající privátního klíče odpovídajícího časové charakteristice poskytované důvěryhodným serverem

1. Programátor zašle cílový čas $T^*$ serveru.
2. Server vrátí veřejný klíč $D^*$ pro tento čas.
3. Programátor použije $D^*$ pro zašifrování citlivých informací agenta.
4. Agent pravidelně požaduje privátní klíč odpovídající aktuálnímu času od serveru.
5. Server vrací $E_i$.
6. Agent zkouší dešifrovat citlivé informace pomocí $E_i$. Uspěje ve chvíli, kdy $T_i$ == $T^*$.

# Příloha D

## D.1 Zdrojové kódy (CD\SourceCodes)

**\LightApp** … Ukázkové využití ochranného rozhranní (softwarový agent). Microsoft Visual Studio 6.0 projekt. Programovací jazyk C++.

**\SecureAlg** … JavaCard applet realizující hardwarový token. Borland JBuilder 6.0 projekt, GemXpresso RAD 3.2 plugin. Programovací jazyk Java.

**\SecureFW** … Implementace ochranného rozhranní. Microsoft Visual Studio 6.0 projekt. Programovací jazyk C++.

**\SecureFW\Tables** … Hlavičkové soubory WBACR AES pro autentizační a transportní šifrovací klíče.

**\Shared** … Sdílené hlavičkové soubory a knihovny.

**\WBACR_AES** … Implementace generátoru WBACR AES tabulek. Microsoft Visual Studio 6.0 projekt. Programovací jazyk C++.