Budapesti Műszaki és Gazdaságtudományi Egyetem

Static Driver Verifier
Fakultatív feladat
Operációs rendszerek (vimia219)

Ferencz Endre

Konzulens: Micskei Zoltán

Budapest, 2010.

1. fejezet

Motiváció

Fontos, hogy egy operációs rendszer eszközmeghajtói jól legyenek megírva. A Windowshoz elérhető egy statikus ellenőrző eszköz (SDV - Static Driver Verifier), ami a C-ben megírt forrásfájlokat vizsgálja, és tipikus hibákat keres az eszközmeghajtókban. A feladat az SDV és az általa használt módszer megismerése, és egy-két egyszerűbb példa eszközmeghajtó elkészítése, amin be lehet mutatni az SDV funkcióit.

A választásom azért esett erre a fakultatív feladatra, mert betekintést nyújt az operációs rendszer és az alatta lévő hardver kapcsolatába. Fontos, hogy jó minőségű kódot írjunk, ugyanis egy driver hiba végzetes lehet az operációs rendszer számára. A Static Driver Verifier egy eszköz, mely felhívja a programozó figyelmét a gyakran előforduló súlyos hibákra, ezáltal csökkentve a driverben található hibák számát. Számomra nagyon fontos, hogy lehetőleg hibamentes kódot írjak és ebben segítségemre lehet ennek a feladatnak a kidolgozása.

Az én elképzelésem a feladatról az, hogy először áttekintem, értelmezem az eszközhöz adott példa driverek hibáit (melyek felderítését a verifier elvégzi), majd én is írok néhány egyszerű meghajtót és ezekre is lefuttatom az ellenőrzést.

A feladat megoldásához hátteret nyújt az egyetem keretein belül elvégzett tantárgyak sokasága, melyek közül kiemelném a C-ben való programozást, ami elengedhetetlen a driverek írásához, megértéséhez.

Ferencz Endre

2. fejezet

Bevezető

„Bugs in kernel-level device drivers cause
85% of the system crashes in the
Windows XP operating system."[1]

Programozás során a hibák egyik elsőszámú forrása az elkerülhetetlen komplexitás. Ez alól nem kivétel a Windows driver API sem: a programozóknak nagy számú szabályt kell ismerni, illetve betartani.

A Static Driver Verifier, röviden SDV, egy a driverek (illesztőprogramok) fordítási idejében használt eszköz, amely segítségével átfogó ellenőrzéseket végezhetünk a C nyelven írt forráskódokon. Az ellenőrzéshez az SDV felhasználja az előre megírt interfészek szabályait (rule), valamint az operációs rendszer egy modelljét. A végső cél az, hogy eldöntsük, a driver megfelelően használja-e az operációs rendszer által nyújtott szolgáltatásokat, vagy sem.

Az SDV egy 2000 környékén induló projekt eredménye (SLAM project), melynek eredeti célja az volt, hogy megvizsgálja a szoftver biztonsági jellemzőit különböző modell ellenőrző technikákkal. Végeredményben a SLAM egy „counterexample-guided abstraction refinement" elnevezésű technikát használ, amely lépésről lépésre egyre jobb modelleket alkot a tesztelni kívánt programról [2].

Az SDV elsősorban olyan eszközmeghajtók ellenőrzésére készült, amelyek a Windows Driver Model (WDM[3]), a Kernel-Mode Device Framework (KMDF[4]), vagy a Network Driver Interface Specification (NDIS) alapján készültek. Úgy tervezték, hogy a teljes fejlesztési ciklus során hasznos legyen a driver alap struktúrájának elkészültétől egészen a végső változat kiadásáig, de az SDV használata a fejlesztés végén a legfontosabb. Itt megemlíteném, hogy rendelkezésre áll egy másik eszköz is a driverek ellenőrzésére, még pedig a PFD (PREfast for Drivers), melynek segítségével az ellenőrzés elvégezhető egy szűkebb tartományra (célszerűen egy metódusra).

Az ellenőrzés során az SDV egy összetett eljárással felderíti a program lefutása során lehetséges útvonalakat és ezekben megkeresi a szabályokban leírt összefüggéseket. Nagy driverek esetén az SDV futási ideje a sok lehetséges végrehajtási vonal miatt akár több óra is lehet.

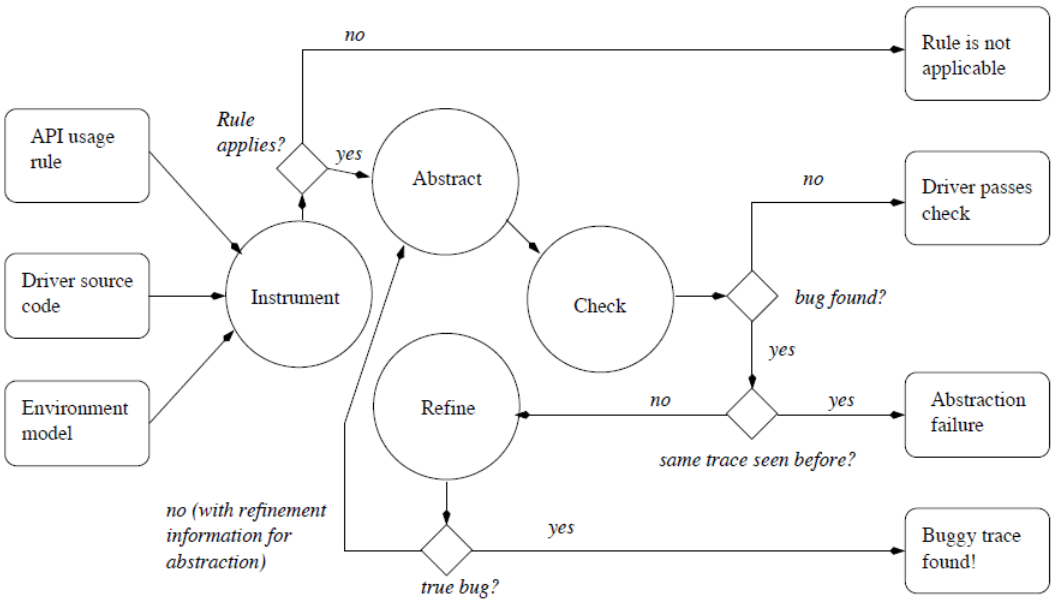
Az SDV[5] mindenki számár elérhető a Microsoft Windows Driver Kit (WDK) letöltésével és feltelepítésével. Az eszköz a \tools\sdv mappába települ és tartalmaz néhány példa drivert is, melyeken tetszőlegesen tanulmányozható a működése.

3. fejezet

SDV

3.1. Rövid bevezető az SDV használatához

A eszköz a 3.1-es vázlat szerint működik: a bemeneti oldalon megkapja a szabályt, a driver forráskódját, valamint az operáció rendszer (környezet) modelljét, melyek alapján előáll a négy lehetséges kimenetből valamelyik. A vizsgálat eredményei lehetnek: a szabály nem alkalmazható, a driver teljesíti a szabályban leírtakat, driver hiba (ebben az esetben megkapjuk a hibához vezető útvonalat is) vagy előfordulhat absztrakciós hiba is.



3.1. ábra. SDV működése

Az SDV használata könnyen elsajátítható a szabadon rendelkezésre álló források [6] alapján: A WDK telepítése után a Windows Start menüjéből elérhetőek a fordítási környezetek. Ezek közül a Checked Build Environment-et célszerű kiválasztani, mivel így plusz funkciók érhetők el a Free Build Environment-hez képest. A környezet felépülése után a forráskódot tartalmazó mappához navigálva először a felhasznált library-ket kell lefordítani a *staticdv /lib* parancs segítségével. Ezek után a driver átvizsgálása következik,

melynek során a *staticdv /scan* parancs hatására az SDV megpróbálja felkutatni a szükséges belépési pontokat. Ezek nélkülözhetetlenek a tényleges ellenőrzés végrehajtásához.

Az ellenőrzés kétféleképpen indítható el: készíthetünk előre egy konfigurációs fájlt, vagy parancssorból megjelöljük azokat a szabályokat, melyekre az ellenőrzést futtani szeretnénk. Ahhoz, hogy minden szabályt felhasználjunk a *staticdv /rule:** parancsot célszerű futtatni.

A parancs sikeres végrehajtása után az SDV felhívja a figyelmünket arra, hogy a *staticdv /view*[7] utasítással megnézhetjük az elkészült részletes jelentést. Itt könnyen felderíthetjük a hiba forrását.

A következőkben a WDM alapokra épülő példa driverek közül az elsőn fogom bemutatni az SDV használatát.

```
cd .\tools\sdv\samples\fail_drivers\wdm\fail_driver1
staticdv /scan
staticdv /rule:CancelSpinLock
staticdv /view
```

Az SDV sikeresen megtalálja a szándékosan implementált hibát. A felugró ablakban rögtön fellelhetők azok az utasítások, amelyek nem megfelelő használata hibás futást eredményezhet. A CancelSpinLock elnevezésű szabály azt ellenőrzi, hogy az IoAcquireCancelSpinLock és az IoReleaseCancelSpinLock utasítások megfelelően követik-e egymást, azaz egy lock-al lefoglalt erőforrást minden esetben felszabadítunk-e. Az első példában a driver által implementált DispatchSystemControl nevű metódus úgy foglalja le az erőforrást, hogy a felszabadítás egyáltalán nem jelenik meg a kódban. Ezt a részt az SDV Report piros színnel jeleníti meg (ld. 1. ábra).

```
158: NTSTATUS
159: DispatchSystemControl (
160:     IN  PDEVICE_OBJECT  DeviceObject,
161:     IN  PIRP            Irp
162:     )
163: {
164:     /*
165:        This defect is injected for the "CancelSpinLock" rule.
166:     */
167:     KIRQL oldIrql;
168:
169:     UNREFERENCED_PARAMETER(DeviceObject);
170:     UNREFERENCED_PARAMETER(Irp);
171:
172:     IoAcquireCancelSpinLock(&oldIrql);
173:
174:     return STATUS_SUCCESS;
175: }
176:
```

3.2. ábra.

Ezek után célszerű megnézni a szabályt is, hogy belelássunk az SDV működésébe, valamint, hogy jobban átláthassuk a CancelSpinLock helyes használatát. A szabály megfogalmaz két belső állapotot: *unlocked, locked.*

```
state{
   enum {unlocked,locked} s = unlocked;
}
```

A kezdetben unlocked állapot bizonyos események hatásánál nyer értelmet. Például az IoAcquireCancelSpinLock meghívásánál. A kódból könnyen kivehető, hogy amennyiben meghívjuk ezt az operációs rendszer által szolgáltatott metódust, két úton ágazhat el a szabályunk: amennyiben már lefoglaltuk az erőforrást, akkor hibásan hívjuk meg újra; ha még nem volt lefoglalva akkor állapotváltás következik be, hogy a későbbi metódushívásokat is megfelelőképpen tudjuk kezelni.

```
IoAcquireCancelSpinLock.exit
{
    if(s==locked) {
        abort "The driver is calling IoAcquireCancelSpinLock
after already acquiring the spinlock.";
    } else {
        s=locked;
    }
}
```

A szabály átvizsgálása után levonhatjuk a következtetést, hogy az SDV úgy vizsgálja meg a végrehajtás útvonalait, hogy az operációs rendszer függvényhívásait „kicseréli" a szabályokban megfogalmazott metódusokra. Ezáltal minden, az operációs rendszer által értelmezett metódushívás ellenőrizhető. Itt felhívnám a figyelmet arra, hogy az SDV csak a driver és az operációs rendszer közötti interakció helyességét vizsgálja, tehát az ellenőrzés nem terjed ki arra, hogy a driver megfelelően elvégzi-e a feladatait.

A driver működése során, az operációs rendszer különböző események bekövetkezténél meghívja az eszközmeghajtó különböző metódusait. Ezeket a metódusokat általában az SDV könnyen felismeri. A felismerés helyességéről az *sdv-map.h* fájl tartalma alapján győződhetünk meg. Az első példa esetében ez a következő képpen alakul:

```
//Approved=false
#define fun_IO_DPC_ROUTINE_1 DpcForIsrRoutine
#define fun_IO_COMPLETION_ROUTINE_1 CompletionRoutine
#define fun_DriverUnload DriverUnload
#define fun_IRP_MJ_SYSTEM_CONTROL DispatchSystemControl
#define fun_IRP_MJ_PNP DispatchPnp
#define fun_IRP_MJ_POWER DispatchPower
#define fun_DriverEntry DriverEntry
#define fun_IRP_MJ_READ DispatchRead
#define fun_AddDevice DriverAddDevice
#define fun_KSERVICE_ROUTINE_1 InterruptServiceRoutine
#define fun_IRP_MJ_CREATE DispatchCreate
```

Fontos, hogy a felismerés helyesen történjen meg, mivel egy rosszul felismert metódus miatt előfordulhat, hogy másképp alakul a program futása, így bizonyos hibákat nem találunk meg, vagy nem létező hibákat jelez az eszköz.

Ezek után javaslom a többi példa átnézését is, mivel az SDV ennél jóval bonyolultabb szabályok ellenőrzésére is képes.

3.2. Írjunk drivert!

Ebben a fejezetben lépésről lépésre fogom bemutatni az SDV hasznosságát egy WDM alapú példa driver formálásával párhuzamosan [8, 9]. Induljunk ki a következő alap driverből, mely csak egy belépési pontot tartalmaz.

```
#include <wdm.h>

DRIVER_INITIALIZE DriverEntry;

#pragma alloc_text (INIT, DriverEntry)

NTSTATUS DriverEntry (IN PDRIVER_OBJECT pDriverObject,
IN PUNICODE_STRING strRegistryPath )
{
    UNREFERENCED_PARAMETER(pDriverObject);
    UNREFERENCED_PARAMETER(strRegistryPath);
    DbgPrint("Hello World!\n");
    return STATUS_SUCCESS;
}
```

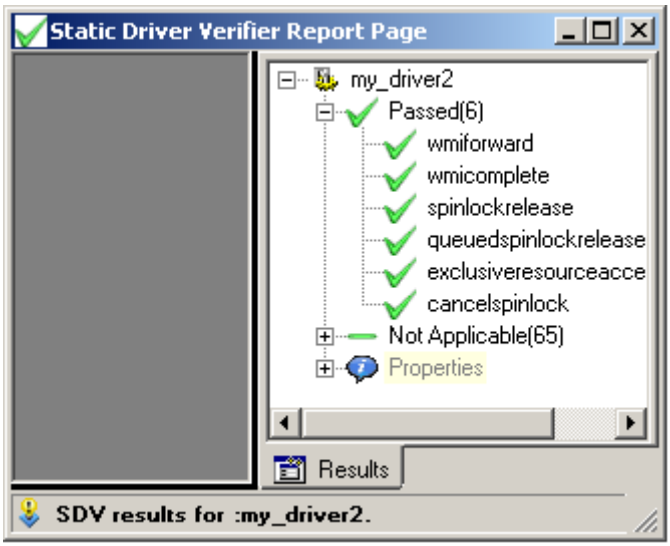
Minden WDM driver kell tartalmazzon egy belépési pontot DriverEntry néven. Ahhoz, hogy ezt a függvényt az SDV felismerje, szükség van arra, hogy az előre definiált típus (DRIVER_INITIALIZE) alapján megadjuk a metódus prototípusát. Ez a függvény két bemeneti paramétert is kap, de ezeket még nem használjuk fel, így megjelöljük őket a későbbi figyelmeztetések elkerülése végett. A legtöbb függvény NTSTATUS visszatérési értéket ad vissza, mely a végrehajtott művelet sikerességéről (sikertelenségéről) tájékoztat minket. A kód harmadik sora arra szolgál, hogy tájékoztassuk az operációs rendszert arról, hogy mikor írható ki a driver a memóriából a háttértárba (pagefile). A legtöbb esetben bármikor nyugodtan kiírható, de bizonyos esetekben (pl. megszakítások) nem szabad laphibának keletkeznie.

Ezt a hello-world drivert az SDV segítségével leellenőrizve láthatjuk, hogy nem szeg meg egyetlen szabályt sem. Az SDV megmutatja nekünk azokat a szabályokat is, amiket nem sikerült alkalmaznia a legelső driveren. Ezek mellett látható, hogy két ellenőrzés sikeresen lefutott.

Sokszor van szükség a lefoglalt erőforrások felszabadítására. Egészítsük ki a hello-world drivert:

```
#include <wdm.h>

DRIVER_INITIALIZE DriverEntry;
DRIVER_UNLOAD DriverUnload;

#pragma alloc_text (INIT, DriverEntry)
#pragma alloc_text (PAGE, DriverUnload)

NTSTATUS DriverEntry (IN PDRIVER_OBJECT pDriverObject,
```

```
IN PUNICODE_STRING strRegistryPath )
{
    UNREFERENCED_PARAMETER(strRegistryPath);
    pDriverObject->DriverUnload = DriverUnload;
    DbgPrint("Hello World!\n");
    return STATUS_SUCCESS;
}

void DriverUnload(IN PDRIVER_OBJECT pDriverObject)
{
    UNREFERENCED_PARAMETER(pDriverObject);
    DbgPrint("Bye..\n");
}
```



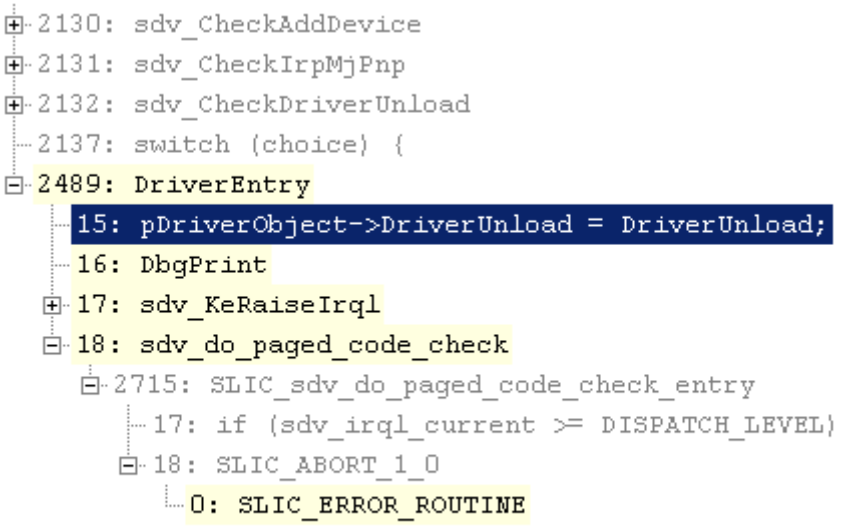
3.3. ábra. Passed

Ez a driver szintaktikailag helyes és amint ez az ábrából is kiderül az SDV sem jelez hibát. Innentől kezdve már sok szabállyal lehet játszadozni. Számomra a legérdekesebb a megszakítás kérések szintjeivel (IRQL - Interrupt ReQuest Levels) való kísérletezés volt. A DriverEntry függvényt a következőre cseréltem ki:

```
NTSTATUS DriverEntry (IN PDRIVER_OBJECT pDriverObject,
IN PUNICODE_STRING strRegistryPath )
{
    UNREFERENCED_PARAMETER(strRegistryPath);
    pDriverObject->DriverUnload = DriverUnload;
    DbgPrint("Hello World!\n");
    PAGED_CODE();
    return STATUS_SUCCESS;
}
```

A PAGED_CODE() makró arra szolgál, hogy leellenőrizhessük, az adott helyen az IRQL nem nagyobb-e az APC_LEVEL szintnél. Ez a feltétel jelen esetben teljesül is. Emeljük meg a szintet magasabbra, hogy lássuk megfelelően működik-e az SDV ellenőrző mechanizmusa. Az eredmény az ábrán látható, a hiba felismerése megtörtént.

```
NTSTATUS DriverEntry (IN PDRIVER_OBJECT pDriverObject,
IN PUNICODE_STRING strRegistryPath )
{
    KIRQL old;

    UNREFERENCED_PARAMETER(strRegistryPath);
    pDriverObject->DriverUnload = DriverUnload;
    DbgPrint("Hello World!\n");
    KeRaiseIrql(DISPATCH_LEVEL, &old);
    PAGED_CODE();
    KeLowerIrql(old);
    return STATUS_SUCCESS;
}
```



3.4. ábra. Failed

Ez csak egy egyszerű példa volt az IRQL-el kapcsolatos ellenőrzéskre. A valóságban rengeteg hibalehetőséget hordoz magában ezeknek a használata (pl. a jelenlegi szintnél alacsonyabb szintre próbáljuk emelni, stb.), ezért ezen a területen rendkívűl hasznos az SDV.

Ezek után érdemes még néhány szabályt kipróbálni, mivel így remekül átlátható az SDV működése.

3.3. KMDF

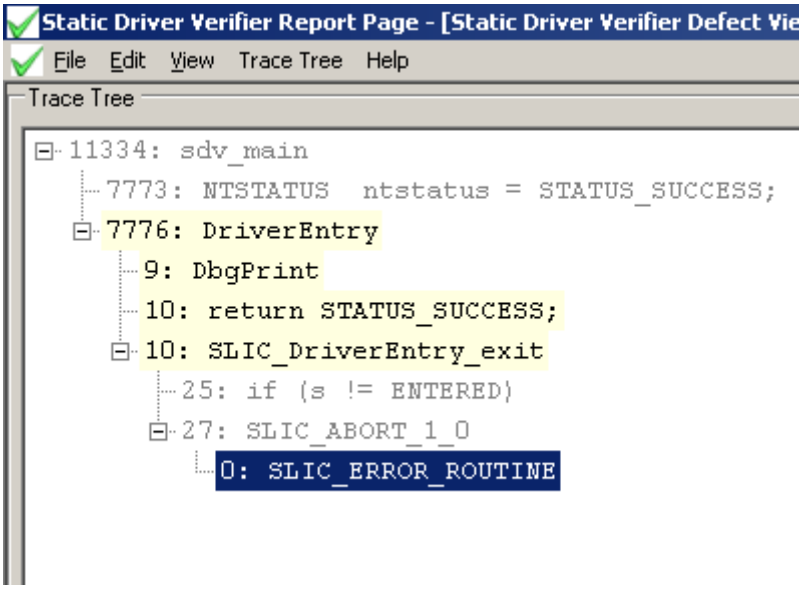
A WDM driver modell alapjaira épül a Kernel Mode Device Framework, mellyel a funkciókat egyszerűbben és biztonságosabban lehet megvalósítani. Ezzel kapcsolatos a következő példa:

```
#include <ntddk.h>

DRIVER_INITIALIZE DriverEntry;

NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
    UNREFERENCED_PARAMETER(DriverObject);
    UNREFERENCED_PARAMETER(RegistryPath);
    DbgPrint("Hello World\n");
    return STATUS_SUCCESS;
}
```

A sikeres fordítás után az SDV a DriverCreate szabályban hibát jelez:

```
#include "ntddk_slic.h"

state { enum {INIT, ENTERED} s = INIT;}


fun_DriverEntry.exit
{
        if (s != ENTERED)
{
            abort "The driver is not calling WdfDriverCreate in its
DriverEntry routine. This is not a WDF driver and no WDF API can be
called.";
}
else
{
halt;
}
}

sdv_WdfDriverCreate.entry
{
    s = ENTERED;
}
```

A szabályt megnézve rögtön láthatjuk, hogy a WDF driverek működésük előtt meg kell hívják a WdfDriverCreate metódust a szükséges paraméterekkel.



3.5. ábra. Failed

3.4. Az eszköz határai

Néhány a driverek szempontjából fontos aspektust nem tud megvizsgálni a statikus ellenőrző:

Memóriakezelés biztonsága Az SDV működése során feltételezi, hogy a mutatott objektumok megfelelően inicializálásra kerültek (nincs wild pointer).

Osztott memória konkurrens kezelése Annak ellenére, hogy sok szabály van a konkurrencia ellenőrzésére, az SDV szekvenciálisan teszteli a lehetséges futási vonalakat, tehát nem ellenőrzi a szálak közötti késleltetésekből adódó hibákat.

Egész számok és bitműveletek Az egész számokat felső határ nélküliként kezeli a statikus ellenőrző, tehát a túlcsordulásból adódó hibákra nem figyelmeztet. A bitműveleteket jelenleg nem értelmezi az eszköz, de a jövőben várható ennek a funkciónak a kiegészítése.

Driver feladatok elvégzése Mint ahogy az egyszerű példákból is láthattuk, az SDV nem ellenőrzi le azt, hogy a driver elvégzi-e a feladatait (belátható időn belül).

3.5. Összefoglalás

Néhány egyszerűbb driver és szabály átnézése után, levonhatjuk azt a következtetést, hogy az SDV segítségével nem csak csökkenteni tudjuk a hibák számát, hanem a fejlesztés során segít minket abban is, hogy sokkal mélyebben át tudjuk tekinteni az operációs rendszer által nyújtott lehetőségeket.

Irodalomjegyzék

[1] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In POPL 03: Principles of programming languages, pages 97-105, 2003.

[2] SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft http://research.microsoft.com/pubs/70038/tr-2004-08.pdf

[3] Windows Driver Model From Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Windows_Driver_Model

[4] Kernel-Mode Driver Framework From Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Kernel-Mode_Driver_Framework

[5] Windows Hardware Developer Central, Static Driver Verifier homepage
http://www.microsoft.com/whdc/devtools/tools/sdv.mspx

[6] Introducing Static Driver Verifier Compile-time verification for WDM kernel-mode drivers
http://www.microsoft.com/whdc/devtools/tools/sdvintro.mspx

[7] Using the Static Driver Verifier Report
http://msdn.microsoft.com/en-us/library/ff556050.aspx

[8] A simple demo for WDM Driver development By mjtsai
http://www.codeproject.com/KB/system/WDM_Driver_development.aspx

[9] Writing a device driver for Windows
http://www.adp-gmbh.ch/win/misc/writing_devicedriver.html