ELECTRONICS AND ELECTRICAL ENGINEERING
ISSN 1392 – 1215 2011. No. 6(112)
ELEKTRONIKA IR ELEKTROTECHNIKA

SYSTEM ENGINEERING, COMPUTER TECHNOLOGY
T 120
SISTEMŲ INŽINERIJA, KOMPIUTERINĖS TECHNOLOGIJOS

Assessment of Dongle-based Software Copy Protection Combined with Additional Protection Methods

A. Liutkevicius, A. Vrubliauskas, E. Kazanavicius
Real Time Computing Systems Centre, Kaunas University of Technology,
Studentų str. 50, LT-51368, Kaunas, Lithuania, e-mails: agnius@ifko.ktu.lt, aras@ifko.ktu.lt, ekaza@ifko.ktu.lt

crossref http://dx.doi.org/10.5755/j01.eee.112.6.459

Introduction

There are a lot of methods to fight against software illegal use, which can be divided into two major groups: software-based protection and hardware-based protection. One of the popular hardware-based software copy protection methods is based on special hardware called "dongle". Dongle is a small USB, RS232 or LPT interface device usually, like USB flash pen, which protects applications from being illegally replicated from original copy. An application is not functional or looses major functionality without dongle plugged in host PC. Application is bound with particular dongle while dongle itself is almost impossible to clone or hack, hence illegally copied application is worthless. Many people and even some software developers and vendors think, that dongle based protection is very hard to break. But such method has its weak spot – communication between dongle and application logic. Communication security can be improved by using standard well known methods [1], [2] like AES, DES, 3DES, RC2, Rijndael, etc., however it only protects low level data transfer between dongle and application, and the attack can be performed at a higher level. Usually application protection is implemented by identifying dongle and checking some secret value, kept in dongle memory. But the problem is that modern software, independently from the programming languages and technologies, can be reverse engineered, disassembled or debugged. Attacker just needs to find code fragments, where applications asks dongle for some value, and place jump over those fragments, so hacked application completely ignores presence of dongle. We found some dongle vendors, which recommend having many calls to dongle, make them randomly and so on, but this adds just few additional minutes for attacker to spend without making protection really stronger.

Recently more advanced dongles appeared, like Rockey (rockey.com.my), Keylok (keylok.com) and some other, which are able to hide some application part inside of them, and execute that part directly in the dongle. This is quite new and promising technology we believe is very hard to break, but it requires more research regarding security strength measurements. On the other hand there are a lot of conventional dongle vendors and users, so our investigation and experiments were intended to show, if it was possible to implement good copy protection using dongles without code execution in combination with other well known software protection techniques, like software packers, anti-debugging, code obfuscation and so on.

Additional protection methods for Dongle-based protection

As described above, the main weak spot of dongle protection is communication between protected application and dongle itself. If attacker finds these calls using debugging or disassembling techniques, protection is breached completely despite communication complexity and communication ciphering. Attacker does not need to reverse engineer communication logic itself, but just place a jump command to avoid dongle checking at all.

There are two possibilities to improve dongle-based protection here. First of all, we need to make dongle-based protection in such a way, that attacker could not find code regions, where application is communicating with dongle, or at least to make this task very difficult and time consuming. Another option is to make protection in a way, that breaking communication with dongle (using jump or similar technique) would prevent program from normal functioning. Later option is included in new generation dongles like Rockey or Keylock, where dongle is no more like a secured external memory, but rather a micro computer with CPU, RAM, EEPROM or Flash memory, which is able to execute some part of the secured application. If the application developers use conventional dongles without code execution abilities (which are still very popular and widely used), then the only possibility to make such protection stronger is to use additional software based protection methods, to hide communication between application and dongle. These methods include anti-debugging, code obfuscation, software packers and few others, which are covered below.

"Anti-debugging encompasses the strategies, techniques, and tricks that protected software uses to attack debuggers and thwart reverse engineering" [3]. There are a number of the known methods of anti-debugging which are summarized in [4]. Also rootkits [5] and other virus like techniques can be used to strengthen anti-debugging protection. The general idea of anti-debugging is to detect or exploit specific debuggers, like most popular OllyDbg, IDA Pro or SoftICE. According to [4] the most commonly used tracking software is OllyDbg and significant portion of anti-dynamic tracking techniques are against it.

Software that employs anti-debugging techniques can determine if it's being debugged by identifying artifacts—side effects of the debugging process—whether from the hardware, software, or human layers [4]. There are several sources of identifying artifacts [3, 4]: API based detection; process and thread block detection; hardware and register based detection; timing based detection; modified code detection; exception based detection.

The main disadvantage of anti-debugging is that there are many plug-ins for popular debuggers and debugger versions, like IDA Stealth for example, which aim to hide the debugger from most common anti-debugging techniques. Such stealth debuggers are significant threat to any anti-debugging technique. But on the other hand, covert anti-debugging methods provide some software protection because reverse engineers can't manually circumvent anti-debugging techniques they don't see.

Code obfuscation is the technique, which makes program source code very hard to understand and is used widely to limit the possibility of malicious reverse engineering or attack activities on a software system [6]. Code obfuscation means original code transformation into new code, which is more difficult to understand, while having behavior identical to the original code. Code obfuscation technology is mainly used in software developed using .NET, Java and other interpretive platforms to protect the intermediate code. Obfuscation quality is defined by several factors: potency (level of obscurity); resilience (difficulty to be broken); cost (computational overhead); stealth (blending with the rest of the code). Obfuscation methods are classified into three groups [6]: layout obfuscations, data obfuscations, control-flow obfuscations.

The main disadvantage of this protection method is that obfuscators are good to hide custom code details and logic, but they cannot obfuscate third party external library calls. Hence it is relatively easy to find such calls to the dongle even in obfuscated program, because obfuscator cannot change names of libraries or their methods.

Packing compresses and/or encrypts the program code in such way, that actual code stays hidden till runtime (when the executable is unpacked) making it immune to static analysis [7]. Packed program contains additional code, which dynamically unpacks or generates original program code in memory and then transfer control to it.

Since every packer has its associated unpacker to undo packing, a successful generic unpacker is difficult to come by [7]. Packing is considered as one of the best protection against reverse engineering, because it can combine other protection methods mentioned above: anti-debugging, code obfuscation, etc. Kim et al. [8] concludes, that there are no well-developed widespread secure binary code packing tools for Linux-based embedded systems, and propose their own packing methods for Linux platform. On the other hand, there are a many widely used packers for the Windows platform including ASPack, ASProtect, PECompact, MoleBox, Armadillo, etc.

Though originally packers were created to minimize size of executables, today they are primary used for software protection, because packed code cannot be analyzed statically. Dynamic analysis using debuggers is also quite complicated, because it is sometimes difficult to identify regions of original unpacked code.

The main disadvantage of this protection method is the fact, that almost every commercial and widely used packer has its own third party unpacker, which can remove additional packer's code leaving only original program (preserving functionality while the unpacked code can differ from original one). Every new version of commercial packer sooner or later (usually sooner) has its own unpacker, which can be found on Internet easily.

Related work

Dongle protection evaluation is quite rare topic among researchers. Piazzalunga et al. [9] proposed general model for dongle based security evaluation and identified both "*attack pattern catalog*" and "*defense pattern catalog*". Authors of [9] developed attack tree model and experimentally proved, that model based protection measurements are quite similar to the field validation results. On the other hand, the details of the field testing are omitted, just presenting defense patterns used and time spent for cracking.

Jozwiak et al. [10] proposed a special hardware device, which is intended to make dongle cracking easier, showing exact moments, when program is accessing dongle. Though such device is good to crack simple and short dongle protected programs, additional protection methods like code packing, can make reverse engineering process much more complicated and not so straightforward. Also instead of proposed hardware module simple USB packet sniffer or similar tools can be used, making such kind of reverse engineering even much easier.

In their next paper Jozwiak et al. [11] present hypothetical hardware protection device, based on ATMega128 MCU with real-time clock. Jozwiak et al. show two methods to crack such protection: switching off cycle checking of device's presence and simple RTC emulator. No additional protection methods like obfuscation, anti-reverse engineering or packing are used. The important conclusion can be found in [11], saying that "primary strength of hardware-based protection centers on a tight binding between protected software and hardware key". In our case the tight binding between dongle and application is achieved using additional protection techniques including anti-debugging, code obfuscation and program packing.

Regarding related work, the aim of our investigation was not only the assessment of dongle-based protection combined with additional protection methods, but also evaluation of possibility to break protection by inexperienced attackers, using widely available tools and methods found on Internet.

Experimental Setup

Security Dongle Hardware, Software and Documentation. For our experimental evaluation we used DLP-D USB-based security dongle manufactured by DLP Design Inc. The dongle is pre-programmed with a unique identification number and additionally has 128 bytes of EEPROM user area to store custom data. API libraries for accessing dongle were downloaded from http://www.ftdichip.com/FTSupport.htm including .NET, Java and C++. We also used application programming interface (API) for the FTD2XX DLL function library programmer's guide, found at the same address provided above. This guide was useful to find out the exact names of API functions, called from the application during authentication with dongle.

Program to Protect. We wrote experimental command line program, which accesses dongle through dongle API library and read the dongle ID. If dongle is present, then dongle ID is read using dongle API and compared to hardcoded ID value to confirm, that correct dongle is inserted. If dongle is not present (dongle API function returns false) program stays locked.

Software for Additional Protection. SoftwarePassport version 8 (trial) based on well known and widely used Armadillo engine from the Silicon Realms Toolworks (http://www.siliconrealms.com) was used for program packing.

Dotfuscator Community Edition was used to obfuscate C# implementation of experimental program (http://www.preemptive.com/products). This tool is a part of MS Visual Studio 2008 distribution.

Another commercial tool we used for C# code obfuscation was Crypto Obfuscator For .Net (v2011) (http://www.ssware.com).

ProGuard (http://proguard.sourceforge.net/) free Java class files shrinker, optimizer, obfuscator, and pre-verifier was used to obfuscate Java implementation of experimental program.

Reverse Engineering Tools. For the protected program debugging and cracking any of above mentioned debuggers is suitable. We used OllyDbg version 1.10, which can be downloaded from http://www.ollydbg.de/. Our OllyDbg version had additional "Olly Advanced" plug-in for the anti-anti-debugging, making this debugger "stealth" and undetectable by known anti-reverse engineering methods and protections.

For the protected program decompiling .NET Reflector version 6.6 (http://www.reflector.net), Dis# .NET Decompiler version 3.1.4, JAD decompiler for Java version 1.5.8 (http://www.varaneckas.com/jad), JD-GUI Java Decompiler (http://java.decompiler.free.fr/) version 0.3.3 with JD-Core version 0.6.0, JODE java decompiler and optimizer (http://jode.sourceforge.net/) version 1.1.2-pre1 were used.

We used ArmStripper v0.1 beta 6 for the unpacking purpose (http://www.woodmann.com/crackz/Packers.htm).

Programming Languages. Protected and later cracked program was implemented using three popular programming languages including C++ (Microsoft compiler used), .NET C# and Java. All implementations had the same functionality and calls to the same dongle API functions from the different API libraries available for different languages. Java and .NET are high level interpreted languages, which are very suitable for the obfuscation protection evaluation, while C++ is compiled into the machine code directly, allowing to evaluate anti-debugging and code packing techniques. Also it was interesting to compare which languages (interpretive or not) are more resistive to the reverse engineering attacks.

Operating system. All experiments there performed on Windows Server 2003 R2 platform.

Protection Methods to Evaluate. The protection methods we used for the experimental evaluation can be divided into few groups: dongle only protection; dongle protection with code obfuscation; dongle protection with program (code) packing.

Dongle only protection was used to evaluate how strong is pure dongle-based protection without any additional protection techniques. It was tested using all three above mentioned programming languages.

Additional code obfuscation was used for the .NET and Java implementations of protected program. Obfuscation is good for interpretive languages, because such languages are quite easily decompiled into original code. On the other hand, debugging is rather useless for such languages, because they are compiled during runtime. Additional anti-debugging protection does not make sense in this case.

The anti-debugging methods can be used separately from other protection methods, but today they are usually combined with code packing techniques. We used commercial SoftwarePassport packing software with enabled anti-reverse engineering protection to evaluate protection of program, written in C++. Since cracking of C++ program was done at the assembly level using debugger, we did not add additional obfuscation protection to this implementation of protected program. At the assembly level obfuscation is useless, because calls to external libraries are leaved as they are in original program.

Evaluation of protection

Dongle Only Protection. Dongle only protection evaluation started with breaking program written in C++. We used OllyDbg for reverse engineering purposes. The main idea was to find calls to the dongle API library, which are used to retrieve dongle ID and patch experimental program to skip dongle checking. Actually this task is extremely easy, because dongle vendors usually give a programming guide with all API function descriptions. OllyDbg environment allows making simple search within assembly to find exact function names. The result of this search is presented in Fig. 1, where call to dongle API library can be seen. Even multiple calls to dongle can be found in several minutes. The next step of changing assembly code to ignore dongle calls was performed using simple JMP command making patched program successfully continue execution ignoring dongle absence completely.

```
MOV DWORD PTR SS:[ESP+10],ECX
MOV BYTE PTR SS:[ESP+4],DL
MOV DWORD PTR SS:[ESP+8],0
CALL DWORD PTR DS:[<&ftchipid.FTID_GetDeviceChipID>]
TEST EAX, EAX
JNZ SHORT _test_c_004010CA
MOV ECX,DWORD PTR SS:[ESP]
```

Fig. 1. Dongle API call found in OllyDbg environment

Even without knowing exact API function names, it is quite easy to check all dongle library calls to find those functions. If dongle has no special library, reverse engineering is almost identical if not more simple. Just instead of finding dongle library calls, attacker searches for Windows API functions to communicate with I/O devices. MSDN Windows API Reference contains all needed documentation for this purpose.

Cracking of experimental program written in high level interpreted Java and .NET languages is quite different. OllyDbg and other assembly level reverse engineering tools are not very suitable, because virtual machine and interpreter hide execution logic at user level and cracking should be performed at kernel level. Though it is still possible using special software tools or plug-ins for debuggers, more simple approach can be used. The main drawback of interpretive programming languages is that programs written in Java or .NET can be easily decompiled into original code. We used .NET Reflector software for C# and JD-GUI decompiler for Java to reverse engineer binaries into source code. Next step was the same like in case with C++ binaries. We made simple API call search and in matter of few minutes found all calls to the dongle.

This group of experiments proved that having commonly available tools and some technical knowledge given by dongle vendors or OS vendors, the dongle protection breaking takes from several minutes to few hours in extreme cases.

Dongle Combined with Code Obfuscation. First group of experiments proved, that without additional protection, dongles without code execution are easily avoided, because there is no big deal to find exact places where dongle is called from the protected program. Logically next step was to see, how additional protection methods like obfuscation will help to improve protection. Obfuscation is useful to resist decompilation, because even decompiled code looks like a mess. We obfuscated both C# and Java implementations of protected program and then decompiled them.

```
DateTime now = DateTime.Now;
int num2 = checkKey();
TimeSpan span = (TimeSpan)(DateTime.Now - now);
if (num2 == 0)
{
    Console.WriteLine("checkKey()=PASSED, execution time="
    + span);
```

Fig. 2. Obfuscated C# program after decompilation with .NET reflector

Actually obfuscation is very good method to hide business logic of program, but in our case we do not care about this. Attacker just wants to find calls to the dongle and avoid them. Obfuscation is applicable to custom code, but not to the third party library calls. It cannot change names of dongle API functions and consequently these calls can be found with the same ease as with non-obfuscated program. C# implementation of experimental program was obfuscated using standard Dotfuscator utility found in Visual Studio 2008 IDE. Then program was decompiled with .NET Reflector. Fig. 2 depicts obfuscated program after decompilation with clearly seen code portion calling dongle API.

```
call    DateTime DateTime.get_Now()
dup
pop
stloc.1
call    int Program.checkKey()
dup
pop
stloc.2
call    DateTime DateTime.get_Now()
dup
pop
```

Fig. 3. Obfuscated C# program after decompilation with Dis# decompiler

Next obfuscator we tried for C# program, was commercial Crypto Obfuscator For .Net (v2011). It allows not only obfuscation, but also packing and anti-reverse engineering protection. In this case only obfuscation protection was enabled. For decompiling we used both .NET Reflector and Dis# tools. Reflector was not able to disassemble code correctly (showing error message), while Dis# was able to show mixed assembler and C# code, which again contained clear logic of how dongle API library functions are called, as depicted in Fig. 3.

Java implementation of experimental program was obfuscated using ProGuard obfuscator. This is quite sophisticated tool, which allows many obfuscation options to make obfuscated code very hard to understand. The bad thing is that protected program must call native dongle API library functions and these calls cannot be obfuscated. This restriction makes reverse engineering straightforward. We decompiled obfuscated java binaries and jar files and simply found native method calls (Fig. 4). Then we traced all calls to these methods (after obfuscation classes and methods were renamed like "a", "b", "a.b", etc.) and easily found exact places, were experimental program checks the dongle. Program was patched and compiled back into binary code.

This group of experiments showed, that code obfuscation almost does not add any significant additional protection for the dongle-based protection solutions. Places where dongle API library is called can be found quickly leading to the simple program patching to ignore dongle presence.

Dongle Combined with Program Packing and Anti-debugging. Last group of experiments included evaluation of dongle protection improved with packing and anti-debugging. For this purpose trial version of SoftwarePassport commercial software was used. C++ implementation of experimental program was packed using additional settings like license, anti-debugging and few others.

```
final a a()
{
        if (this.a == null)
        {
                EEPROM localEEPROM = this;
                 localEEPROM = this;
                 if (!null.e())
                        throw new
                        IllegalStateException(a.a.a.a(EEPROM.class,
                        "error.notOpen"));
                this.a = readDeviceDescriptor(null.b());
        }
        return this.a;
}


private static native a readDeviceDescriptor(long paramLong);
```

Fig. 4. Obfuscated Java program after decompilation with JD-GUI decompiler

Our first step of breaking packer protection started with avoiding additional anti-reverse engineering techniques applied by packer. Packed program did not allow debugging at all, throwing warning message and terminating. We used Olly Advanced plug-in for the OllyDbg debugger. This plug-in is very easy to use simply selecting few checkboxes with anti-reverse engineering methods attacker would like to avoid. Checking all checkboxes allowed analyzing program without any inconvenience ignoring all anti-debugging protections.

In contrast with dongle only protection, on the first look packing hides original program code details, because original code is unpacked during runtime. Most simple "brute force" solution is to step over assembly code using debugger until original program is unpacked and loaded into memory. Next phase is the same as in case with dongle only protection: make search for dongle API calls and patch them with JMP or similar techniques. Such approach though simple, can require a lot of time. More sophisticated protections can include time-based checking and automatic program shutdown not allowing reaching point, when program is fully loaded into memory. This would increase time needed for attacker to break the protection as well. Having this in mind, we used another attack method, which can be applied even by inexperienced attacker. We simply searched on Internet for the "armadillo unpacker" and tried to use few first matches. We found that even quite old unpacker ArmStripper v0.1 beta 6 (designed for older versions of Armadillo) unpacked our experimental program without any problems. It was noticed though, that unpacked code differs comparing with original code before packing, but the functionality is equal. Unpacked code was analyzed and calls to dongle API were found using debugger's search engine. These calls were patched like in previous test cases.

Java packers usually include wrapping Java bytecode with C family program which leads to the same reverse engineering process, like with C++ program implementation (non-interpreted language).

C# implementation of program was obfuscated and packed with Crypto Obfuscator. This was the only time, when we were unable to decompile it using general tools. However from the experience we got before, it is just a matter of having the right knowledge and finding the right tools for successful attack.

Our experiments emphasized problem associated with packing-based protections: almost every known packer has its own third party unpacker. The general recommendation we can give is to use custom (unknown) packer, though we understand that it is quite unrealistic that software developers would spend additional time and money to make protection mechanisms more expensive than program they want to protect.

The summarized results of experimental evaluation are presented in Table 1, showing the time spent to break particular software protection method.

Conclusions and Future Work

In this paper assessment of dongle-based software copy protection combined with additional protection methods is presented. The security dongle without code execution ability was used for experimental evaluation combining it with additional protection methods including code obfuscation, anti-reverse engineering and code packing.

The experimental results show that even occasional attacker can quite easily break dongle (without code execution) protection using widely known tools and information found on Internet.

Experiments proved that having commonly available tools and some technical knowledge given by dongle vendors or OS vendors, the dongle only protection breaking takes from several minutes till few hours in extreme cases.

Code obfuscation almost does not add any significant additional protection for the dongle-based protection solutions. Places where dongle API library is called can be found quickly leading to the simple program patching to ignore dongle presence.

Table 1. Time spent to break software protection

	Dongle Only Protection			**Dongle with Obfuscation Protection**			**Dongle with Packing Protection**	
Programming language	C++	.NET	Java	.NET		Java	C++	.NET
Additional protection	Not used	Not used	Not used	Dotfuscator	Crypto Obfuscator	Pro Guard	SoftwarePassport (Armadillo)	Crypto Obfuscator
Time (minutes)	~10	~5	~5	~3	~8	~7	~15	N/A

Packed versions of dongle protected programs are more difficult to crack, comparing with obfuscated and dongle protected or protected only with dongle programs. On the other hand, we showed that with use of open source and freely downloadable tools even packed programs can be reverse engineered in matter of minutes.

In future works we are planning to evaluate the dongles with code execution ability using well known reverse engineering attack methods.

References

1. **Toldinas J., Štuikys V., Ziberkas G., Naunikas D.** Power Awareness Experiment for Crypto Service–Based Algorithms // Electronics and Electrical Engineering. – Kaunas: Technologija, 2010. – No. 5(101). – P. 57–62.
2. **Toldinas J., Stuikys V., Damasevicius R., Ziberkas G., Banionis M.** Energy Efficiency Comparison with Cipher Strength of AES and Rijndael Cryptographic Algorithms in Mobile Devices // Electronics and Electrical Engineering. – Kaunas: Technologija, 2011. – No. 2(108). – P. 11–14.
3. **Gagnon M.N., Taylor S., Ghosh A.K.** Software Protection through Anti–Debugging // IEEE Security & Privacy, 2007. – Vol. 5. – Iss. 3. –P. 82–84.
4. **Tang Jiutao, Lin Guoyuan.** Research of Software Protection // International Conference on Educational and Network Technology (ICENT'2010), 2010. –P. 410–413.
5. **Toldinas J., Rudzika D., Štuikys V., Ziberkas G.** Rootkit Detection Experiment within a Virtual Environment // Electronics and Electrical Engineering – Kaunas: Technologija, 2009. – No. 8(104). – P. 63–68.
6. **Ceccato M., Di Penta M., Nagra J., Falcarin P., Ricca F., Torchiano M., Tonella P.** The Effectiveness of Source Code Obfuscation: an Experimental Assessment // IEEE 17th International Conference on Program Comprehension (ICPC'2009). – Vancouver, Canada, 2009. – P. 178–187.
7. **Babar K., Khalid F.** Generic Unpacking Techniques // 2nd International Conference on Computer, Control and Communication (IC4'2009), 2009. – P. 1–6.
8. **Min–Jae Kim, Jin–Young Lee, Hye–Young Chang, SeongJe Cho, Yongsu Park, Minkyu Park, Wilsey P. A.** Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering // 2010 13th IEEE International Symposium on Object/Component/Service–Oriented Real–Time Distributed Computing (ISORC), 2010. – P. 80–86.
9. **Piazzalunga U., Salvaneschi P., Balducci F., Jacomuzzi P., Moroncelli C.** Security Strength Measurement for Dongle–Protected Software // IEEE Security & Privacy, 2007. – Vol. 5. – Iss. 6. – P. 32–40.
10. **Jozwiak I. J., Liber A., Marczak K.** A Hardware–Based Software Protection Systems–Analysis of Security Dongles with Memory // International Multi–Conference on Computing in the Global Information Technology (ICCGI'2007), 2007.
11. **Jozwiak I. J., Marczak K.** A Hardware–Based Software Protection Systems – Analysis of Security Dongles with Time Meters // 2nd International Conference on Dependability of Computer Systems (DepCoS–RELCOMEX '07), 2007. – P. 254–261.

Received 2011 03 29

A. Liutkevicius, A. Vrubliauskas, E. Kazanavicius. Assessment of Dongle-based Software Copy Protection Combined with Additional Protection Methods // Electronics and Electrical Engineering. – Kaunas: Technologija, 2011. – No. 6(112). – P. 111–116.

Dongle is a hardware device which is bound with software application in such way, that application functions only if dongle is plugged in. The most modern dongles are able to hide some parts of application's code and execute this code directly inside the dongle, but today's market has a lot of dongle types, which are not able to execute code. This paper presents our investigation regarding evaluation of software protection using dongles without code execution ability. Commercial dongle is used for the case study, combining it with well known software protection methods to hide application communication with dongle. The experimental results show that even inexperienced attackers can quite easily break dongle without code execution protection using widely known tools and information found on Internet. Ill. 4, bibl. 11, tabl. 1 (in English; abstracts in English and Lithuanian).

A. Liutkevičius, A. Vrubliauskas, E. Kazanavičius. Aparatiniais raktais su papildomais apsaugos metodais pagrįstos programinės įrangos apsaugos įvertinimas // Elektronika ir elektrotechnika. – Kaunas: Technologija, 2011. – Nr. 6(112). – P. 111–116.

Apsaugos raktas – tai specialus aparatinis įrenginys, susietas su taikomąja programa taip, kad be rakto programa neveikia. Nors patys naujausi apsaugos raktų modeliai gali paslėpti dalį taikomosios programos funkcijų ir vykdyti jas rakto viduje, rinkoje vis dar plačiai siūlomi raktai, negalintys vykdyti kodo. Šiame straipsnyje eksperimentiškai įvertinamas negalinčių vykdyti programos kodo apsaugos raktų apsaugos lygis. Tam naudojamas komercinis apsaugos raktas, kurio komunikavimui su taikomąja programa paslėpti naudojami gerai žinomi papildomi apsaugos metodai. Eksperimentų rezultatai rodo, kad net nepatyrę programinės įrangos piratai gali apeiti apsaugos rakto užtikrinamą apsaugą, panaudodami plačiai žinomus programinius įrankius bei informaciją, laisvai prieinamą internete. Il. 4, bibl. 11, lent. 1 (anglų kalba; santraukos anglų ir lietuvių k.).