International Journal of Security and Its Applications
Vol. 10, No. 10 (2016) pp.169-180
http://dx.doi.org/10.14257/ijsia.2016.10.10.16

Sensitive Data Hiding Scheme for Internet of Things using Function Call Obfuscation Techniques

Jeongmi Shin and Yeonseung Ryu*

Department of Security and Management Engineering, Myongji University
116 Myongji-ro, Cheoin-gu, Yongin, Gyeonggi-do, Korea
shinblly@gmail.com, ysryu@mju.ac.kr

Abstract

A new era of Internet of Things (IoT) will emerge in near future due to the rapid innovations and advancements in computing and communication technologies. In order to make IoT era become possible, sensitive data in IoT devices such as cryptographic keys should be protected from malicious attackers. In particular, cryptographic keys are vital for securing systems and communication. In this paper, we study a sensitive data hiding scheme for IoT devices using software obfuscation techniques. Proposed scheme exploits function call obfuscation and creates a hidden area between functions to secretly store sensitive data. The proposed scheme outperforms the legacy hardware-based schemes which impose additional hardware cost to IoT devices.

Keywords*: Internet of Things, sensitive data protection, cryptography, obfuscation*

1. Introduction

The Internet of Things (IoT) refers to an internet connectivity of physical objects beyond traditional devices like desktop computers and smartphones to a various range of devices and things that utilize embedded technology and wireless technology. Examples of objects that can fall into the scope of Internet of Things include thermostats, cars, electronic appliances, lights in household, alarm clocks, speaker systems, vending machines and more. Today there are already more IoT devices than persons in the world [1]. According to Gartner, there will be nearly 20.8 billion devices on the Internet of Things by 2020 [2].

In the IoT environment, smart objects will be able to communicate with each other in an autonomous manner. The Institute of Electrical and Electronics Engineers (IEEE) and the Internet Engineering Task Force (IETF) are designing new communications protocols for low-energy communication of IoT devices. Examples of IoT communication protocols include IEEE 802.15.4 [3], 6LoWPAN [4], Routing Protocol for Low-power and Lossy Networks (RPL) [5], Constrained Application Protocol (CoAP) [6]. The communications with these protocols should be protected from those who have intension of stealing sensitive information [7-9]. To do so, the security mechanisms must provide appropriate assurance in terms of confidentiality, integrity, authentication and non-repudiation. In order to provide secure communications to IoT devices, cryptography is essential mechanism and cryptographic keys are the fundamental component of this protection mechanism [10]. Keys are the critical component for securing systems, communication and applications, and therefore must be protected at all times. Once the keys are exposed to malicious attackers, fatal exploits can be easily created.

There are two types of solutions to protect the sensitive data: hardware-based and software-based solutions. In case of hardware-based solutions, Trusted Platform Module (TPM) is well-known and widely used for high-performance computing platforms [11-

* Corresponding Author

ISSN: 1738-9976 IJSIA

12]. TPM is a hardware chip that can securely store artifacts used to authenticate the computing platform such as PC, mobile phones and network devices. These artifacts can include passwords, certificates, or encryption keys. TPM is also used to store platform measurements that help ensure that the platform remains trustworthy. The nature of hardware-based solution ensures that the information stored in hardware is better protected from external software attacks. However, hardware-based solutions impose additional cost to the product and thus cannot be used appropriately for IoT products that usually require low cost. The alternative cheaper solution is software-based protection. However, it cannot be trustworthy because it is very difficult to hide sensitive data from external attacks.

In this paper, we study a software-based sensitive data hiding scheme for IoT devices using function call obfuscation technique. Software obfuscation is the technique of making the software code hard to read and understand by changing it [13-16]. In order to protect intellectual property of software or disturb tampering the source code, it is vital to obfuscate software source code before deployment. There are various obfuscation techniques including control flow obfuscation, layout obfuscation, data obfuscation, and so on. Examples of control flow obfuscation are function call obfuscation and jump obfuscation. If IoT devices are deployed without software obfuscation, they are easily exposed from harmful accesses. In order to protect intellectual property or disturb tampering the source code, it is vital to obfuscate software source code before deployment.

The proposed scheme assumes that there is a compiler which performs software obfuscation. When the compiler translates source code into machine code, it performs function call obfuscation. We make the compiler to input sensitive data information from user and to create a hidden area between obfuscated functions to hide sensitive data. If the binary code is disassembled, hidden area represents the meaningless machine instructions. Therefore, when performing the reverse engineering, the attackers have difficulties in understanding the execution flow and hidden area. In terms of cost, the proposed software-based hiding scheme outperforms the legacy hardware-based schemes which store data in additional hardware chip.

The rest of the paper is organized as follows: In Section 2, we describe the related works including cryptographic key management and function calling conventions. In order to understand the function call obfuscation, we introduce the function calling conventions in x86 and ARM architecture in detail. Section 3, we present our proposed scheme for sensitive data hiding using the function call obfuscation techniques. We describe our scheme by illustrating the source code examples in 16-bit x86 and ARM. Finally, Section 4 concludes the paper.

2. Related Works

2.1. Cryptographic Keys

Cryptography is a method of storing and transmitting data so that only those for whom it is intended can read and process it [10]. There are five primary functions of cryptography: privacy/confidentiality, authentication, integrity, non-repudiation, and key exchange. In cryptography, a key is a piece of information used in cryptographic algorithm. For encryption algorithms, a key specifies the transformation of plaintext into cipher text, and vice versa for decryption algorithms. According to NIST SP 800-57, there are many types of cryptographic key such as private authentication key, public authentication key, symmetric data encryption key, symmetric master key and so on [17]. The key management is one of the most important aspects of cryptographic systems for securing computing and communication systems. Keying material should be available as long as the associated cryptographic service is required.

Keys may be maintained within a cryptographic module while they are being actively used, or they may be stored externally and recalled as needed [17]. Some keys may need to be archived if required beyond the key's originator usage period.

According to NIST SP 800-57, the following protections and assurances may be required for the keying material.

- *Integrity protection (also called assurance of integrity)* shall be provided for all keying material. Integrity protection always involves checking the source and format of received keying material. Integrity protection can be provided by cryptographic integrity mechanisms (*e.g.*, cryptographic checksums, cryptographic hash functions, MACs, and signatures), non-cryptographic integrity mechanisms (*e.g.*, CRCs, parity checks, *etc*.), or physical protection mechanisms.
- *Confidentiality protection* for all symmetric and private keys shall be provided. Public keys generally do not require confidentiality protection. When the symmetric or private key exists internal to a validated cryptographic module, confidentiality protection is provided by the cryptographic module in accordance with [18], level 2 or higher. When the symmetric or private key exists external to the cryptographic module, confidentiality protection shall be provided either by encryption (*e.g.*, key wrapping) or by controlling access to the key via physical means (*e.g.*, storing the keying material in a safe with limited access).
- *Association protection* shall be provided for a cryptographic security service by ensuring that the correct keying material is used with the correct data in the correct application or equipment.
- *Assurance of domain-parameter and public-key validity* provides confidence that the parameters and keys are arithmetically correct.
- *Assurance of private key possession* provides assurance that the owner of a public key actually possesses the corresponding private key.
- *The period of protection* for cryptographic keys, associated key information, and cryptographic parameters (*e.g.*, initialization vectors) depends on the type of key, the associated cryptographic service, and the length of time for which the cryptographic service is required. The period of protection is not necessarily the same for integrity as it is for confidentiality.

2.2. Function Calling Conventions

A function calling convention is defined as an implementation scheme for how functions receive parameters from their caller and how they return a result [19]. Calling conventions may differ in:

- Where parameters, return values and return addresses are placed (in registers, on the call stack, a mix of both, or in other memory structure)
- The order in which actual arguments for formal parameters are passed
- How a return value is delivered from the callee back to the caller (on the stack, in a register, or within the head)
- How the task of setting up for and cleaning up after function call is divided between caller and the callee
- How local variables are allocated

Calling conventions are usually related to a particular programming language's evaluation mechanism as well as CPU architecture.

2.2.1. x86 architecture: There are some types of function calling conventions in x86 architecture: cdecl, stdcall, fastcall [20-21].

```
#include <stdio.h>
int __calling convention convention_test (int a, int b)
{
        return a + b;
}

int main (int argc, char* argv[]) {
        int x;
        x = convention_test(1,2);
        return 0;
}
```

Figure 1. C Code for Testing the Function Calling Convention

The cdecl (which stands for C declaration) originates from the C programming language and is used by many C/C++ compilers. The C language uses this calling convention by default. So if you define nothing about function calling convention, it is considered as cdecl. In cdecl, function arguments are passed on the stack in Right-to-Left order. And return value s are passed in EAX register [20]. Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved. In the Figure 2, 'ADD ESP, 8'means that the caller cleans the stack after the function call returns.

```
main :
PUSH EBP
MOV EBP, ESP
PUSH EAX
PUSH 2
PUSH 1
CALL _cdecl_test
ADD ESP, 8
XOR EAX, EAX
POP EBP
RETN
```

Figure 2. Assembly Code of Main Function

```
_cdecl_test :
PUSH EBP
MOV EBP, ESP
MOV EAX, [EBP+8]
ADD EAX, [EBP+C]
MOV ESP, EBP
POP EBP
RETN
```

Figure 3. Assembly Code of cdecl Test Function

The stdcall calling convention is a variation on the Pascal calling convention in which the callee is responsible for cleaning up the stack, but the parameters are pushed onto the stack in right-to-left order, as in the cdecl calling convention. Registers EAX, ECX, and EDX are designated for use within the function. Return values are stored in the EAX

register. The arguments on the stack are cleaned up by the callee in contrast with the cdecl convention. We can see this by 'RET 8' line in the Figure 6. Also, stdcall is the default for system calls including Windows API calls.

```
main :
PUSH EBP
MOV EBP, ESP
PUSH EAX
PUSH 2
PUSH 1
CALL _stdcall_test
XOR EAX, EAX
POP EBP
RETN
```

Figure 4. Assembly Code of Main Function

```
_stdcall_test :
PUSH EBP
MOV EBP, ESP
MOV EAX, [EBP+8]
ADD EAX, [EBP+C]
MOV ESP, EBP
POP EBP
RETN 8
```

Figure 5. Assembly Code of stdcall Test Function

As you can see in the Figure 8, we can't find 'PUSH' instruction and there are 'MOV' instructions. That's because the fastcall is a register based calling convention which passes the first two arguments in registers, with the most commonly used registers being EAX, ECX and EDX. It can reduce the number of memory accesses required for the call and thus make calling usually faster. Also, it can directly add the value of EDX register to it of EAX register as the Figure 9. And when the number of parameters is over the 3rd parameter, they are stored on the stack.

```
main :
PUSH EBP
MOV EBP, ESP
PUSH EAX
MOV EDX, 2
MOV EAX, 1
CALL _fastcall_test
XOR EAX, EAX
POP EBP
RETN
```

Figure 6. Assembly Code of Main Function

```
_fastcall_test :
PUSH EBP
MOV EBP, ESP
ADD EAX, EDX
MOV ESP, EBP
POP EBP
RETN
```

Figure 7. Assembly Code of Fastcall Test Function

2.2.2. ARM Architecture: There are registers which are generally used to hold either data or an address in ARM architecture. They are identified with the letter *r* prefixed to the register number [22].

The registers are visible to the programmer as *r0* to *r15*. The ARM processor has three registers assigned to a particular task or special function: *r13*, *r14* and *r15*:

- Register *r13* is traditionally used as the stack pointer (*sp*) and stores the head of the stack in the current processor mode.
- Register *r14* is called the link register (*lr*) and is where the core puts the return address whenever it calls a subroutine.
- Register *r15* is the program counter (*pc*) and contains the address of the next instruction to be fetched by the processor.

The calling convention being usually used in ARM architecture is similar to the *fastcall* in x86 architecture. As it is above-mentioned, the *fastcall* in x86 passes one or two arguments in registers. In ARM architecture, on the other hand, the first four integer arguments are passed in the first four ARM registers: *r*0, *r1*, *r2* and *r3*. And subsequent integer arguments are placed on the full descending stack, ascending in memory as in Figure 2. But, if arguments are over 32-bit length such as long long or double they split in two and are passed in a pair of consecutive argument registers and returned in *r0*, *r1*. Function return values are passed in *r0*.

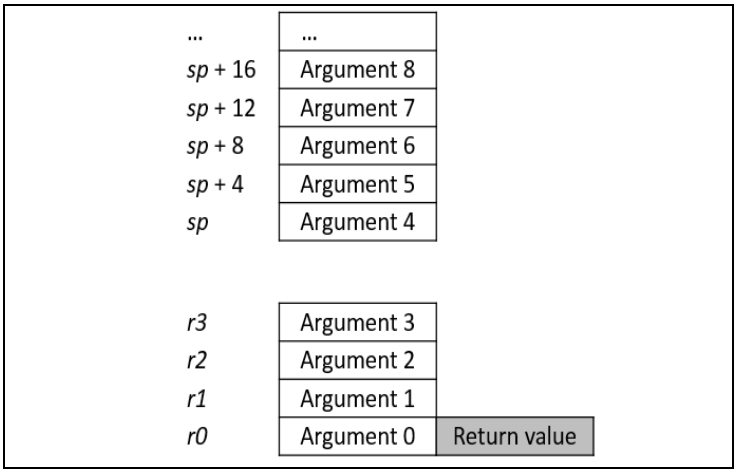

Figure 8. Argument Passing of Function call in ARM

With regard to function calling and return, there is a difference too. BL instruction and MOV instruction is used instead of CALL and RET instructions in x86 architecture. BL instruction means to branch with link. It allows program to have subroutines and store a return address in the link register *lr* (same as *r14*). In another words, *lr* has the address of the next instruction after the BL instruction. At the end of the function, the address in *lr* is

copied to the *pc* (same as *r15*) by performing 'MOV pc, lr' in the Figure 9. And it is same as RET instruction is implemented. In the case of a function frequent called, it has advantage in terms of speed. That's because it doesn't have to store return address in the stack each time. For this reason, there doesn't exist the RET instruction in ARM architecture.

```
        BL      subroutine      ; branch to subroutine
        CMP     R1, #3          ; compare r1 with 3
        MOVEQ   R1, #0          ; if (r1==3) then r1 = 0
        :
subroutine
        <subroutine code>
        MOV     PC, LR          ; return by moving pc = lr
```

Figure 9. An Example of a Function Call in ARM

3. Sensitive Data Hiding Scheme using Function Call Obfuscation

In this section, we study a sensitive data hiding scheme using obfuscation techniques and function calling conventions. The proposed scheme assumes that there is a compiler which performs software obfuscation by calling the obfuscator. When the compiler translates source code into machine code, it calls obfuscator to perform function call obfuscation. The compiler inputs sensitive data information from user and creates a hidden area between obfuscated functions to hide sensitive data.

The obfuscation techniques are categorized as call obfuscation, return obfuscation, false return obfuscation, jump obfuscation and so forth [14-16, 23-24]. Among them, our scheme is related to the function call obfuscation. Usually the next instruction to be executed after a function end is the instruction located after its respective function call instruction. By using this convention, the disassembler commonly disassembly the bytes stored after call instruction. The obfuscator exploits this disassembler's behavior and obfuscates a program, redirecting the return of a function to another location. As a result, it can create a hidden area between functions to store some information which we want to hide. If the binary code is disassembled, hidden area represents the meaningless machine instructions. Therefore, when performing the reverse engineering, the attackers have difficulties in understanding the execution flow and hidden data.

3.1. x86 Architecture

The proposed scheme modifies the obfuscator to make a function call using other set of instructions and the fastcall calling convention. Without loss of generality, we explain the proposed scheme. This scheme scan be adopted to various processor architectures with minor modification. In this proposed scheme, to obfuscate a function call, we create an intermediate function which uses the fastcall calling convention. Using the intermediate function, we push the memory addresses on the stack: the address of the function to be executed and the return address of the function. And the RETN instruction makes the control flow to be unchangeably run.

For example, Figure 10 and 11 illustrate a non-obfuscated and an obfuscation function call in the order named. In Figure 10, the 'Function 1' is called at address 0x1020. The RETN at the address 0x1080 lets the control flow go back to the address 0x1024 (MOV AX, 10) which is next to the CALL instruction of the 'Function 1'. In Figure 11, the 'MOV AX, 10' instruction is relocated to 0x104C, but it doesn't affect the original program execution flow in the proposed scheme. In Figure 11, the 'Function 2' is an intermediate function between a caller and a callee.

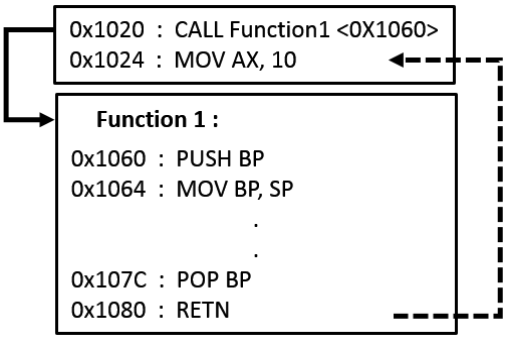

Figure 10. Non-obfuscated Function Call in x86

In this work, we assume that the memory address is 16-bits length. So we utilize 8-bits registers. Firstly, the address 0x104C is stored at BH and BL registers using two MOV instructions, and then two PUSH instructions load the memory address on the stack. The memory address of the 'Function 1' is called by the 'Function 2'. We pass the address 0x1060 to parameters of the 'Function 2'. As the 'Function 2' uses the fastcall calling convention, it is to use CX and DX registers instead of PUSH instruction.

When the control flow is transferred to the 'Function 2', we move SP register to next to where the value of the BX register is loaded on the stack. Then, the address 0x1060 stored at CX and DX registers is loaded on the stack. When the 'Function 2' encounters RETN instruction at the address 0x2014, it gets the memory address of the 'Function 1' (the address 0x1060) on the top of the stack. When the 'Function 1' finishes, the RETN of the 'Function 1' gets the address 0x104C on the stack. Finally the program execution flow is the same as the original code.

It is important that there is the memory interval between 0x103C to 0x1048. It is called a hidden area. We can use this to secretly store data which must be hidden.

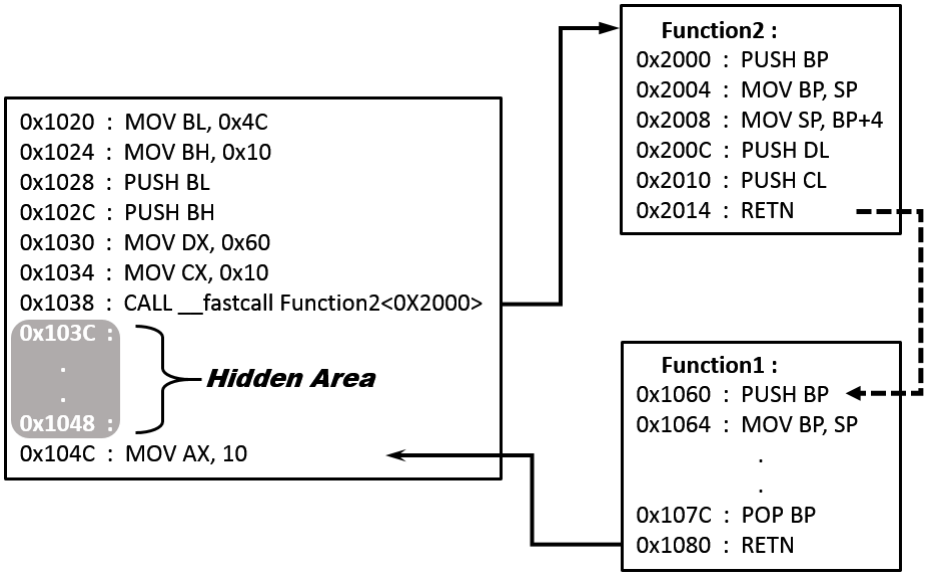

Figure 11. An Obfuscated Function Call Creates a Hidden Area in x86

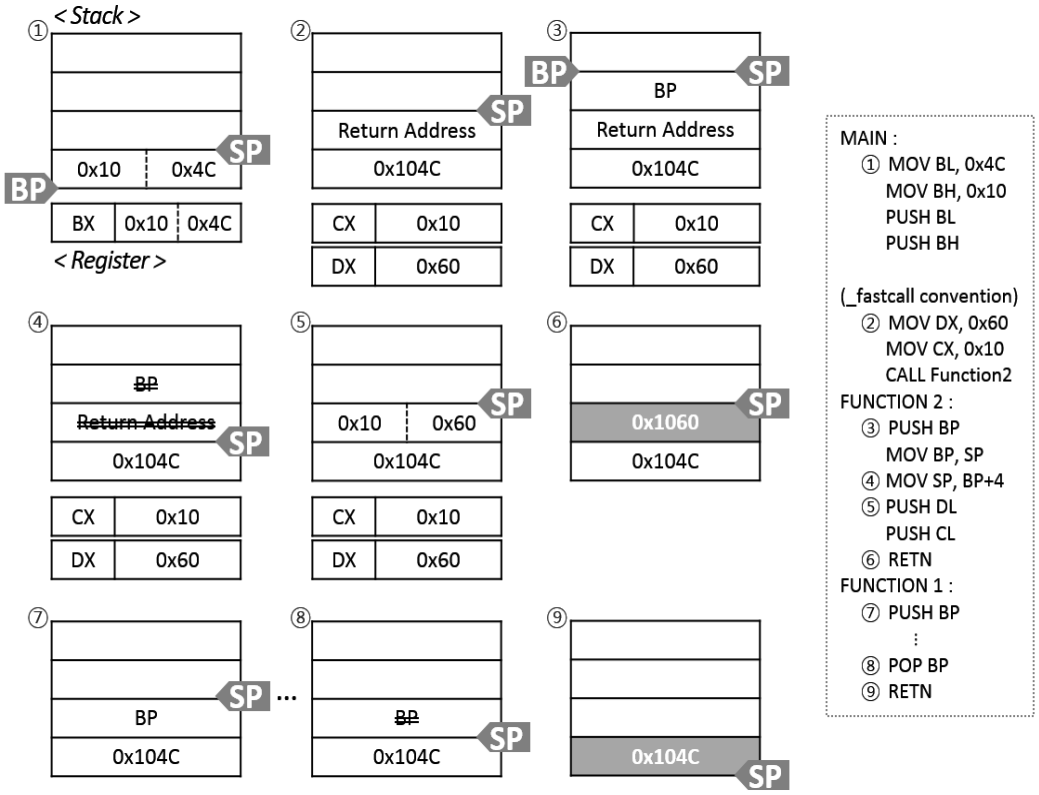

Figure 12. Changes of Stack and Registers during the Execution of Code in Figure 11

3.2. ARM Architecture

The proposed scheme's concept in ARM architecture is similar to that of x86 architecture. We once explained that CALL and RETN instruction in x86 system is substituted with BL and MOV instruction in ARM architecture, in Section 2.2. Since ARM architecture doesn't have the fastcall convention, we cannot obfuscate function call using the fastcall convention. Instead, this proposed scheme modifies the obfuscator to make a function call using the *four-register rule* and the link register, *lr*.

Figure 13 shows an example of non-obfuscated function call in ARM architecture. In Figure 13, the 'Function 1' is called by BL instruction. In ARM architecture, BL instruction performs a branch with link operation [E]. BL instruction stores the next value of the program counter – the return address – into the link register (*lr*) and the destination address into the program counter (*pc*). Hence, it immediately transfers execution control to the destination address, passing the return address in *lr* as an additional parameter to the called subroutine. Later, execution control is returned to the instruction following BL instruction when the return address (which is stored in *lr*) is loaded back into the *pc*. In Figure 13, at the address 0x1020, *pc* is changed to the address of 0x1060. And the return address 0x1024 is in *lr*. It lets the flow of execution go to the 'Function 1' (0x1060). At the end of the 'Function 1', execution flow goes back to the address 0x1024 by performing 'MOV pc, lr'.

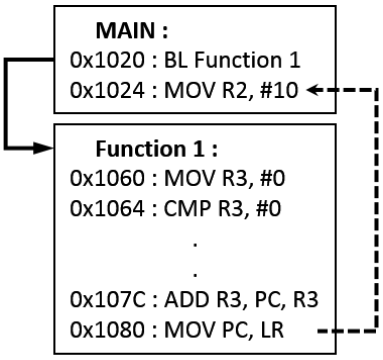

Figure 13. Non-obfuscated Function Call in ARM

Now, we describe how we obfuscate functions and create the hidden area. Figure 14 illustrates an example. We create an intermediate function called 'Function 2'. The intermediate function's role is to obfuscate a function call by changing the return address. As the 'Function 2' is branched, the arguments of the function are passed in the first four registers: from *r0* to *r3*. Then at 0x2000, the 'Function 2' adds the value of *r1* to that of *r0* and adds *r3* to *r2* (*i.e., r0 = r0 + r1, r2 = r2 + r3*). Consequently the value of *r0* becomes 0x1060 and the value of *r2* becomes 0x104C. And then at 0x2008, 'BL R0' lets the control flow be transferred to the 'Function 1'. At the end of the 'Function 1', 'MOV PC, LR' transfers execution flow back to the next instruction of the 'BL R0' in the 'Function 2'. After that, we directly change the value of *lr* to *r2* by performing 'MOV LR, R0' at 0x2004. The link register of the 'Function 2' held 0x1038 which is next to the 'BL Function 2', but it is turned into 0x104C. By doing that, the value of the link register is copied to the program counter when returning from the 'Function 2'.

Note that there is no change in the program execution flow between Figure 13 and Figure 14. Furthermore, we can create a hidden area from the address 0x1038 to 0x1048. That is, we can secretly store a cryptographic key or important data in this area.

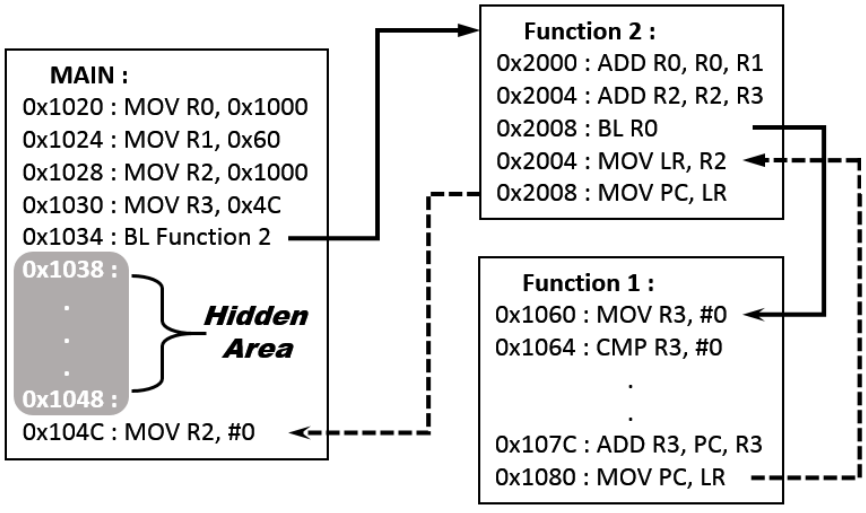

Figure 14. An Obfuscated Function Call Creates a Hidden Area in ARM

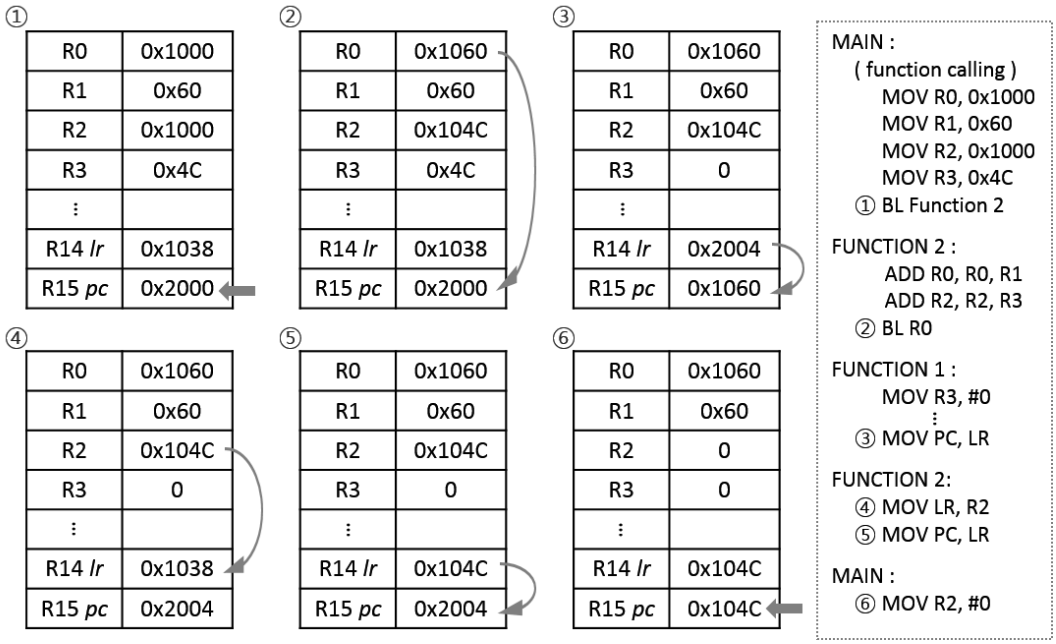

Figure 15. Changes of Registers during the Execution of Code in Figure 14

4. Conclusion

For secure Internet of Things, cryptography and software obfuscation are indispensable for protecting sensitive information and software intellectual property. In this paper, a sensitive data hiding scheme has been proposed for IoT devices. The proposed scheme exploits software obfuscation technique and conceals sensitive data in the hidden area of obfuscated software code. In terms of cost, the proposed software-based hiding scheme outperforms the legacy hardware-based schemes which store data in additional hardware chip. For the future work, we will implement the proposed scheme in small-scale IoT devices to evaluate the performance.

Acknowledgments

This work was supported by 2016 Research Fund of Myongji University. This paper is a revised and expanded version of a paper entitled "Key Protection Scheme for Secure Internet of Things" presented at "The 5th International Conference on Information Science and Industrial Applications, Harbin, China, August 19-20".

References

[1] http://www.ericsson.com/mobility-report
[2] Gartner News, http://www.gartner.com/newsroom/id/3165317, **(2015)** November.
[3] IEEE Standard for Local and Metropolitan Area Networks. Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs) Amendment 1: MAC Sublayer, IEEE Std. 802.15.4e-2012 (Amendment to IEEE Std. 802.15.4-2011), **(2011)**.
[4] N. Kushalnagar, G. Montenegro and C. Schumacher, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, Goals", RFC 4919, **(2007)**.
[5] P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP.Vasseur and R. Alexanders, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks", RFC 6550, **(2012)**.
[6] C. Bormann, A. Castellani and Z. Shelby, "CoAP: An Application Protocol for Billions of Tiny Internet Nodes", IEEE Internet Computing, vol. 1, no. 2, **(2012)**, pp. 62-67.
[7] S. Sicari, A. Rizzardi, L.A. Grieco and A. Coen-Porisini, "Security, Privacy and Trust in Internet of Things: The road ahead", Computer Networks, vol. 76, **(2015)**, pp. 146-164.

[8] G. Jorge, M. Edmundo and S. Jorge, "Security for the Internet of Things: A Survey of Existing Protocols and Open Research Issues", IEEE Communication Surveys and Tutorials, vol. 17, no. 3, **(2015)**.
[9] F. Hu, "Security and Privacy in Internet of Things (IoTs): Models, Algorithms, and Implementations", CRC Press, **(2016)**.
[10] N. Ferguson, B. Schneier and T. Kohno, "Cryptography Engineering: Design Principles and Practical Applications", Willey, **(2010)**.
[11] S. Kinney, "Trusted Platform Module Basics: Using TPM in Embedded Systems", Newnes, **(2006)**.
[12] W. Arthur and D. Challener, "A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security", Apress, **(2015)**.
[13] K. Wilson, "An Introduction to Software Protection Concepts", Intellectual Property Today, **(2007)**, August.
[14] A. Lakhotia and E. U. Kumar, "Abstract Stack Graph to Detect Obfuscated Calls in Binaries", Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation, **(2004)** September.
[15] C. Kruegel, W. Robertson, F. Valeur and G. Vigna, "Static Disassembly of Obfuscated Binaries", Proceedings of the 13th USENIX Security Symposium, **(2004)** August.
[16] A. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly", Proceedings of the 10th ACM conference on Computer and communication security - CCS'03, **(2003)**.
[17] NIST Special Publication 800-5, Part 1 Revision 3, "Recommendation for Key Management", **(2012)**.
[18] Federal Information Processing Standard 140-2, "Security Requirements for Cryptographic Modules", **(2001)** May 25.
[19] Wikipedia, https://en.wikipedia.org/wiki/Calling_convention.
[20] J. Cavanagh, "X86 Assembly Language and C Fundamentals", CRC Press, **(2013)**.
[21] B. Dang, A. Gazet, E. Bachaalany and S. Josse, "Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation", Wiley, **(2014)**.
[22] W. Hohl and C. Hinds, "ARM Assembly Language: Fundamentals and Techniques", CRC Press, **(2014)**.
[23] E. U. Kumar, A. Kapoor, and A. Lakhotia, "DOC-Answering the Hidden 'Call' of a Virus", Virus Bulletin, **(2005)**.
[24] A. Lakhotia, D. Boccardo, A. Singh and A. Manacero Jr, "Context-sensitive Analysis without Calling Context", Higher-Order and Symboic Computation, vol. 23, no. 3, **(2010)**, pp. 275-313.

Authors

Jeongmi Shin, she received her BS degree in Computer Engineering from Myongji University, Korea, in 2015, and she is currently studying a master's degree in Security and Management Engineering from Myongji University in 2015. Her research interests include information security and convergence security.

Yeonseung Ryu, he received his BS degree in Computer Science and Statistics from Seoul National University, Korea, in 1990, and his MS and PhD degrees in Computer Science from Seoul National University in 1992 and 1996, respectively. In 1996 he joined Samsung Electronics, Co. as a senior researcher. Since 2003, he has been with Myongji University, Korea, where he is currently a full professor in the Computer Engineering Department and Security and Management Engineering Department. From March 2009 to February 2010, he was a visiting scholar at the University of Minnesota, USA. His research interests include operating systems and convergence security.

