Lee and Kwon *EURASIP Journal on Wireless Communications and Networking* (2016) 2016:250
DOI 10.1186/s13638-016-0746-6

EURASIP Journal on Wireless Communications and Networking

RESEARCH Open Access

Secure dissemination of software updates for intelligent mobility in future wireless networks

JongHyup Lee[1] and Taekyoung Kwon[2*]

Abstract

Wireless mobile networks frequently need remote software updates to add or adjust the tasks of mobile nodes. Software update traffic, particularly in the Internet of Things (IoT), should be carefully handled since attackers can easily compromise a number of unattended devices by modifying a piece of code in the software update routine. These attacks are quite realistic and harmful as seen in the real world. To protect lower-powered mobile devices, an in-network detection mechanism is preferred. However, due to the mobility of devices, it is difficult to set a network monitor with complete context of software updates. Moreover, even the conventional integrity checks can be fooled by a replaced binary code or minimized modification. In this paper, we tackle this problem and propose CodeDog, a new approach to check the integrity of software updates in mobile environments. CodeDog generates a binary code with semantics markers. A validation of those markers proves the control flow semantics was unchanged. It can be performed on program fragments for in-network monitoring to protect incapable devices. Our evaluation result shows that CodeDog can prevent attacks in the supply chain with 4.2 % storage overhead.

Keywords: Mobile networks, Software updates

1 Introduction

The Internet of Things (IoT) environments consist of heterogeneous devices and wireless, mobile networks. The versatile use of IoT devices causes frequent software updates by administrators to add or adjust the tasks of the devices. Due to the complexity of IoT environments, software updates are being delivered by a *deployment chain*, which is operated by outsourced authorities or network providers. However, the frequent updates expose the deployment chain to ever-growing attacks. Software updates are traditionally primary aims of attackers since a malformed software update can change the behavior of all victims. Hence, any point of the deployment chain can be compromised. A question then arises: can we always trust the deployment chain?

The deployment chain for the software updates starts from the administrator (and its developer) and ends at remotely located target devices. The developer writes a new source code and compiles it to an executable in the binary code form and then hands it over to the deployment chain. The deployment chain identifies the target devices and transports the binary code to the devices through its delivery networks. The recipient devices apply the delivered software updates to their system. In this way, the deployment chain automatically operates with frequent software updates for a number of devices. Due to these characteristics of the deployment chain, once any point of the deployment chain is compromised, the attack is not easily detected and affects a number of devices. For example, the attack on the deployment chain of SK Communications in South Korea, 2011 [1], could compromise a number of targets and have resulted in the theft of personal information of more than 35 million people.

In IoT environments, a large portion of low-end devices are operating. We cannot guarantee that all of them have security-aware update mechanisms, namely, it is hard to expect sophisticated software validation in the low-end devices, such as IoT light bulbs. To effectively protect deployment chain in IoT environments, the network itself should verify the integrity of software updates on behalf of

*Correspondence: taekyoung@yonsei.ac.kr
[2]Graduate School of Information, Yonsei University, 50 Yonseiro, Seoul, South Korea
Full list of author information is available at the end of the article

the low-end devices. For the purpose, network monitoring approaches such as intrusion detection system (IDS) or deep packet inspection (DPI) are being used to detect attacks by investigating packets with predefined rules or algorithms.

When applying the network-side protections to the IoT environments, we should also take into account the mobility of IoT devices. Conventional approaches, e.g., code signing and checksum, are based on the complete information of a software. However, the mobility of the devices hinders to get a complete program code at a single network position. A network entity has to decide whether an attack is happening or not with only a part of information. Even in that case, we need to protect mobile IoT devices from sophisticated attacks as an extended security service of mobility management.

Protections under limited environments due to mobility and capacity should care about effectiveness and concentrate on attacks customized to the environments. Generally, the network-support protections have focused on checking integrity of contents without any information. This content- or payload-agnostic approaches treat software in the same way as they check data. Code and data are different; the program code can be executed; thus, it has semantics. However, the approaches cannot ensure the semantics of a program code is intact. If the protections rely on syntactic methods only, adversaries can effortlessly hide the attacks by modifying a tiny portion of the program code. Furthermore, these attacks are more effective in mobile networks, since a network entity can investigate only a part of the program code. The binary code itself in a software update server can be replaced by a forged one before disseminating it [1]. During the delivery, like software cracking [2–4], the attackers can execute the limited modification attacks by minimizing the change within a few bytes. Just flipping a bit can complement the jump condition of the key branch from `jl` ($<$) to `jge` ($\geq$).

We can sum up the requirements to effectively check the integrity of software updates in mobile networks as follows: (1) the semantics changes of malformed software updates should be checked in the binary code and (2) the changes should be able to be detected by network monitors, i.e., watchdogs, as well as recipient devices to support all ranges of devices. Therefore, we propose CodeDog, a lightweight approach to check semantics integrity in an incomplete code. CodeDog can inspect and verify control flow semantics in all the software objects covering from a fragmented binary code to a whole program. To do so, we carefully implant *markers* in the binary code that statically shows valid control transfers. Thus, in-network watchdogs can early check the changes of software updates and support the low-end devices that cannot perform complicated integrity checks. The recipient device can employ the markers in the process of software attestation.

The contributions of this paper are:

1. We propose a lightweight method to check the integrity of control flow semantics for IoT and mobile environments. The proposed method transforms the binary code of program text into a verifiable form with semantics markers. Thus, it can prove the developers' intention is unchanged in software update with respect to control flows.
2. To effectively detect the attacks, CodeDog checks the integrity in mobile network as well as the recipient devices. The semantics markers also are valid in the fragments of the binary code. Hence, the watchdogs in mobile networks investigate the binary code of software updates from packets, and the recipient devices validate the received software updates and a whole program code for later software attestation.

2 Design

2.1 Attacks on software updates

The goal of an attacker is to alter the behavior of the software. To do so, the attacker can replace a whole code itself with a forged binary code or modify a portion of it. In order to keep the attacks undetectable and sustainable, they tried to minimize the changes. Traditionally, software cracking has a similar purpose, changing a target control flow while keeping other flows intact. For example, flipping one bit of key branch instructions can incur a completely different behavior.

2.2 Integrity of control flow semantics

The semantics of program consists of data abstractions and control flows. Checking complete semantics is the best for detecting changes in the software, but it needs computationally expensive methods, such as flawless symbolic executions. IoT devices are constrained in computation and power to support such expensive operations. Hence, we narrow our focus on more effective targets; CodeDog detects the change of control flow in the program binary code. To do so, it injects semantics markers to the binary code before and after control transfer statements. The semantics markers also bear the conditions of branches. Thus, unintended changes in control flows can be revealed with only semantics markers in the program code.

2.3 Process overview

A whole process of CodeDog is shown in Fig. 1. When initiating software updates, a developer or an administrator prepares the payload for software deployment chain. The developer writes down or updates the source code and then hands it over to the *payload generator* of CodeDog in order to build the payload of the marker-injected binary code. The payloads are transmitted as packets through the

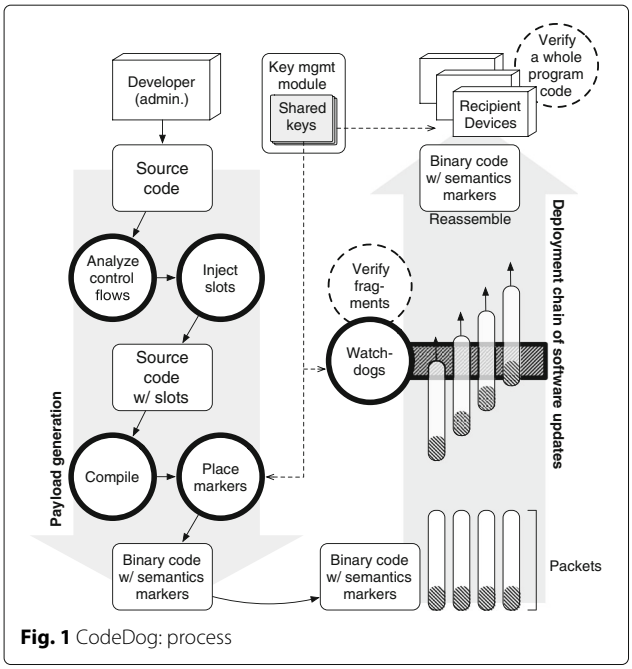

Fig. 1 CodeDog: process

mobile network of the deployment chain. While transmitting the packets, *code watchdogs* monitor the mobile network and detect the unintended modification in advance. The recipient devices then check semantics integrity in the software updates reassembled from the packets and perform further verification. When generating the markers in the *payload generator* and verifying them during the delivery process, CodeDog uses a secret key to authenticate the markers. For the purpose, a key management module handles the distribution of keys as shown in Fig. 1. The secret keys are shared with components in the whole process; thus, any group key management schemes can be applied to the key management module. The detailed process for the payload generator and the integrity checks are explained in the following sections, Section 3 and Section 4.

2.4 Features of CodeDog

Two-step marker injection Injecting markers into program binary code may alter the address of program text, and it skews the target address of control transfer instructions. Thus, binary rewriting requires a painful job to update all the address of control transfer instructions even in the position-independent code (PIC). This is the reason why binary rewriting on a code is being used in limited ways. In order to avoid the problem, we first inject empty slots to a source code to prepare space for the semantics markers and compile the source code. At the binary code, the slots will then occupy spaces for markers. We fill the slots by computing semantics markers for that position. Since space for markers already exists at compile time, no address of the program code needs to be changed; thus, no additional binary rewriting is required.

Semantics markers Semantics markers are placed before and after direct control transfers. In a binary code, all the direct control transfers will be transformed as conditional or unconditional jump instructions. In a source code, we have more control statements: `if`, `while`,[1] `goto`, `break`, `continue`, etc. Both `if` and `while` have a condition, true and false branches. Thus, we put a jump marker (slot) immediately above the control statement, and a true-branch marker at the beginning of the true branch and vice versa. Figure 2 shows the position of slots, which will be markers after compilation, for `if` and `while`. A jump marker is inserted before both `if` and `while`. For `if`, the branch for the then block has a true marker and the other branch has a false marker.[2] For `while`, a true marker is put in front of the loop body. When the `while` loop ends, a false marker is placed at the exit. The semantics markers for a control transfer instruction forms a group. A semantics marker has a validator as a hash value of an identification (ID) of the marker group, an indicator of branch (jump/true/false), the opcodes of the immediate previous and next instructions, and a shared key. Thus a marker is locked in the position and can be verified with the shared key.

Marker verification A group of markers are verified together for a control transfer instruction. Since they have complete information for the control transfer, the same verification process can be performed regardless of whether they come from fragments or a whole code. Thus, we can use the code watchdogs, which monitor the wireless network to detect attacks from binary code fragments. The recipient devices also have a whole updated program binary code. The program binary code with semantics

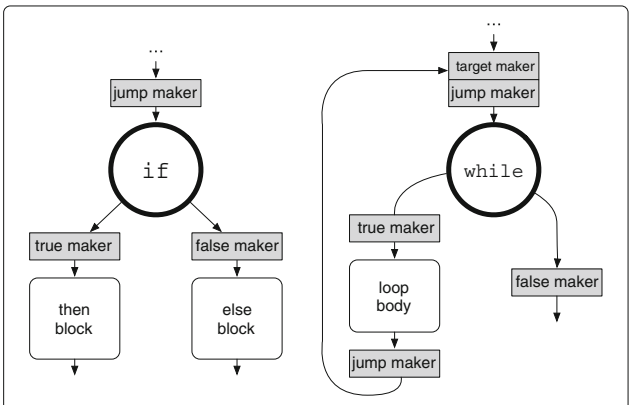

Fig. 2 Marker positions for `if` and `while`

markers can be repeatedly checked in the software attestation process. To prevent the attacker from illegally generating semantics markers, the marker is protected by a key. We assume the key is shared with the payload generator, code watchdogs, and the recipient devices.

3 Payload generator

In this section, we handle the detailed step for the payload generator. In the payload generator, CodeDog builds payload from the source code handed over by developers or administrators of the IoT devices.

3.1 Source code inspection

At the first step for making payload, CodeDog inspects the source code and discovers control transfer statements in it. Depending on the control transfer statements, we place slots of different types around them. Slots and markers have four types: $\mathcal{T}$ = jump, target, true, and false. In addition, every slot group has an indicator for marker calculation in the next step. To place a slot at a chosen location, we use an inline assembly code of the `prefetchnta` instruction. The `prefetchnta` is also used to set a label on the program code in CFI [5] since it does not incur side effects. Thus, the slot in an inline assembly form is shown as

`__asm__("prefetchnta` $\mathcal{T} \,||\, S_i$`");`

where S_i is a 30-bit slot indicator for the ith slot group and $\mathcal{T}$ is a 2-bit type indicator. $||$ is the operator for concatenation.

We put slots for unconditional and conditional control transfer statements. Figure 3 illustrates the position of slots.

- Unconditional control transfers: Like `goto`, `break`, and `continue`, for unconditional control transfer statements, we put a jump slot before the statements and a target slot at the target position.
- Conditional control transfers: Depending on a condition, the conditional control transfer statements select a jump target between a true and a false branch. As shown in Fig. 2, we put a jump slot immediately before the conditional control transfers, such as `if` and `while`, and place true and false slots at the beginning of the corresponding block. That is, for `if`, we put a true slot at the "then" block. If "else" block exists, we put a false slot at it. Otherwise, we put it immediately after the "then" block. For `while`, we place a true slot at the beginning of the loop body and a false immediately after the loop body.

3.2 Marker injection

Abstractions in the source code are removed during compilation, but the slots remain at the same position in the binary code as they are put in the source code. CodeDog then computes a marker value for each slot and replaces the operand of the slot with it. A marker consists of a group ID i and a validator, $\mathcal{V}$. The validator $\mathcal{V}$ is used to verify the marker is securely located. It is a hashed value of a group ID i, a type indicator $\mathcal{T}$, the first byte of the opcode of the immediate previous and next instructions of the marker ($\mathsf{op_p}$ and $\mathsf{op_n}$, respectively), and a key K, which is provided by the key management module:

$$\mathcal{V} = H(i \| \mathcal{T} \| \mathsf{op_p} \| \mathsf{op_n} \| K)$$

Due to the limited operand space, we truncate the least significant bits (LSBs) of the two values and concatenate them with its type. Thus, a marker value m is

$$m = (\mathcal{T} \parallel i|_{14} \parallel \mathcal{V}|_{16}),$$

where $|_x$ is the operation for x-bit truncation of the LSB. Since $\mathcal{V}$ has the information (opcode) of the previous and next instructions, m is valid at the current position; thus, the marker is locked to the position. However, before calculating $\mathcal{V}$ and m, CodeDog may change the type of markers. It depends on the compilers what assembly code will be emitted for a conditional control transfer statement. For example, the assembly code for `if` of $>$ (greater than) may be `jle` (jump if less/equal) of the complemented ($\neg$) condition, which jumps to the else block. For consistent results, we redefine $\mathcal{T}$ of the marker at the target address (taken branch) of the conditional jump as true.

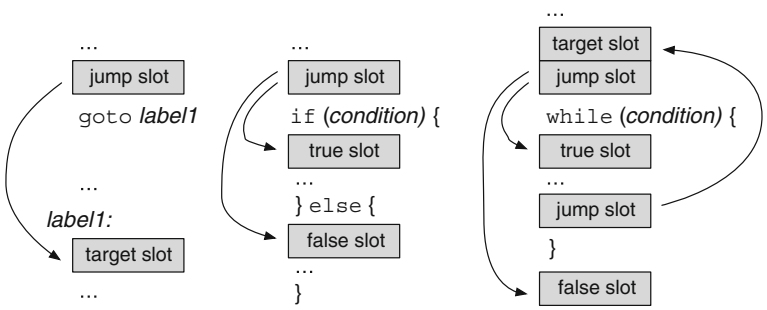

Fig. 3 Position of slots for unconditional/conditional control transfer statements

Likewise, we redefine $\mathcal{T}$ of the marker located next (not taken branch) to the conditional jump as false.

Figure 4 shows the process to CodeDog for `goto` and `if`. CodeDog first injects slots and sets the marker value at the binary code. For `goto` statement, only jump and target markers are used, but the conditional jump `if` statement uses true and false markers. Note that the `if` condition was $<$ in the source code but the binary code employs `jle`; thus, the types of slots and marker are exchanged.

4 Integrity checks

4.1 Different levels of fragmented software updates

A software update payload does not have to hold a complete program. In the mobile network of the deployment chain, we need to take into account the different levels of fragmented binary code: (1) diff'ed binary code and (2) packet fragments. Depending on the size of changes from the old version, we first choose either a whole program or a set of fragments of changed parts, called "diff," to transmit. The diffs are fragments of the binary code com from the different position of the program. The deployment chain then splits the binary object into packets to deliver them through mobile networks. Thus, when we check the integrity of software updates in the middle of networks, we need to inspect the binary fragments. Fortunately, CodeDog injects the semantics markers in the binary code itself. We can verify control flow semantics without knowledge of the whole program. It is incremental: as we collect more semantics markers, we can verify more control flows. When we have a whole program code, CodeDog can prevent attacks from all the modification of control flows.

4.2 Integrity checks in a code watchdog

CodeDog supports the integrity checks of software updates through network monitoring like surveillance watchdogs. We call the entities performing the in-network attack detection on software updates as "code watchdog." A code watchdog monitors (via overhearing wireless networks) the software update packets. In this way, even low-end devices that do not have the ability to check software updates can be protected by the code watchdogs since the code watchdogs are aware that undiscovered modification attacks are happening to mobile/IoT devices.

The code watchdog monitors network traffic through mobile networks and triggers software update inspection when it collects the software update packets. First, it recovers a fragmented part of the software updates from the collected packets. Packet monitoring of the code watchdog is a semi-stateful process. It cannot store all the session information of passing-by traffic. However, the internet data traffic is bursty so that a small group of packets in the same session are collected together. After recovering the fragment, to identify all the semantics markers in the fragments, the code watchdog searches the prefix of semantics markers, such as the opcode of `prefetchnta`. And then, the code watchdog validates every identified semantics marker:

1. Validate the position of the semantics marker. The code watchdog computes $\mathcal{V}'$ from i, $\mathcal{T}$ (given in the marker), $\mathrm{op_p}$, $\mathrm{op_n}$ (collected from the previous and next instructions in the fragment binary), and the shared key K (obtained from the key management module). If $\mathcal{V}'$ is the same as $\mathcal{V}$, the marker is valid.
2. If the type $\mathcal{T}$ of the semantics marker is jump, find the successive branch instruction.
   (a) If the branch is unconditional, find the target marker at the jump address and check whether it is valid and has the same group ID.
   (b) If the branch is conditional, find the true marker at the target address and false marker at the following instruction. And then, check the validity of both tags.

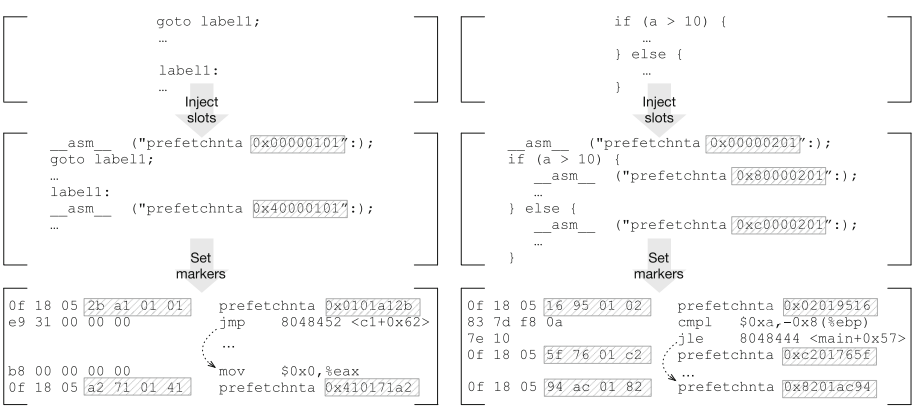

Fig. 4 Example of the CodeDog process to inject slots and set semantics markers

3. If the type $\mathcal{T}$ of the semantics marker is other than jump, remember the position for later checking triggered by the jump marker of its group.

When the code watchdog finds an invalid semantics marker, it notifies the administrator of malformed software updates and cancels the delivery to protect the recipient devices. Note that for some semantics markers, it may not find the corresponding markers of the same group in the same fragment. However, it can still be used to validate the position of the markers.

4.3 Integrity checks in the recipient devices

At the destination, a recipient device reassembles the packets and get full software updates. Before applying the software updates, it checks the semantics markers. The process is the same as that of the code watchdog in Section 4.2, but the device can check all the markers since it has a complete software update with the program code of the previous version. If the device can validate all the markers at an appropriate position, it applies the software update. Since the program code provided by CodeDog bears the semantics markers, it can be used to detect the modification of the program code in software attestation phases. Optionally, the markers can be rewritten at the recipient device with their own secret information for further protection.

5 Evaluations

5.1 Implementation

CodeDog can be applied to any programming language and platforms, but, for evaluations, we choose C programming language and x86 platform. We implemented CodeDog in two parts. First, we employ CIL [6] for the source code inspection and slot injection. And then, our binary rewriter module in python sets the marker values. In the slot injection, we define a new visitor in OCaml for CIL's control statements, such as `Goto`, `If`, and `Loop`.[3] The visitor adds slots to the visited control statements. Second, we implement the integrity checker in python. Both the binary rewriter and the integrity checker use `ndisasm` and `binutils` to get the disassembled code from the binary code fragments.

5.2 Storage overhead

We tested CodeDog on the Juliet test suite from the NIST Software Assurance Reference Dataset (SARD). From the test suite, we choose 30 source codes in the categories related to attacks on control flows, such as untrusted search path, trapdoor, unchecked loop, unchecked error condition, embedded malicious code, and logic bomb. We checked the size of the compiled binary code with CodeDog to see how much the injected semantics markers inflate the binary code. Figure 5 compares the size of the binary code when it generated with and without the semantics markers. The size of the binary code compiled with the semantics markers is 76.6 kB. Compared with the size without CodeDog which is 73.5 kB, we can see that CodeDog causes 4.2 % storage overhead in the compiled binary code.

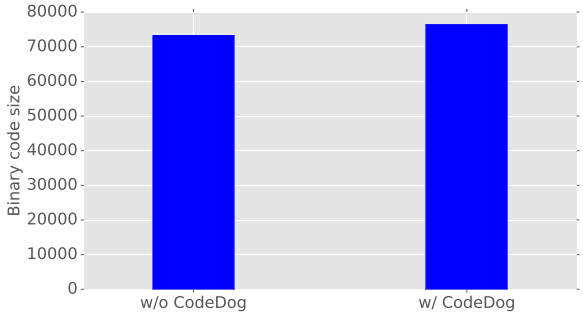

Fig. 5 Storage overhead of CodeDog

5.3 Effectiveness of semantics marker protection

In a whole program code, the semantics markers can detect all the illegitimate control transfers modified by attackers. On the other hand, in a fragment, a code watchdog can find the modification of a control transfer statement only if it can check all the markers that correspond tot the statement together. Therefore, we evaluate the rate of successful validation of markers in a fragmented binary code.

We define the probability of all the markers is included in the same fragment as the *inclusion rate*, $I(M)$:

$$I(M) = \begin{cases} \frac{M-s}{2s+M} & M > s \\ 0 & \text{otherwise} \end{cases},$$

where M is the size of the fragment and s is the *marker spread*, the gap between the lowest and highest addresses of markers in a group, namely, $I(M)$ is the probability that the range of a marker group is within a fragment.

We first check the distribution of the marker spread in our dataset. Figure 6a is a histogram of the marker spreads computed with our dataset. We can see in Fig. 6a that most of the semantics markers are closely located. Simple condition checking routines using `if` contribute the large population of small marker spreads.

Fragment size We investigate the inclusion rate with the marker spread of Fig. 6a by varying fragment sizes, M. Figure 6b shows the inclusion rate of our dataset. When the fragment size is 700, the inclusion rate is only 0.69. However, it grows to 0.82 and 0.898 when the fragment size is 1500 and 3000 bytes, respectively. Note that the maximum transmission unit (MTU) of Ethernet is around 1500 bytes. Thus, the inclusion rate for 1500 and

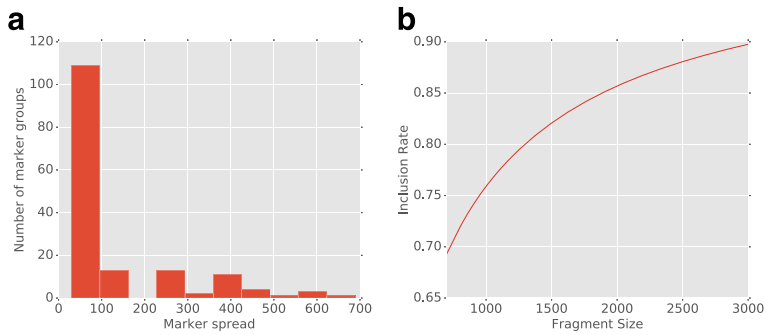

Fig. 6 Marker effectiveness in the binary code fragments. **a** Histogram of marker spreads. **b** Inclusion rate vs. fragment size

3000 means the probability of successful marker validation when the code watchdog inspects a single packet and two sequential Ethernet packets, respectively.

Mobility We then check the effect of mobility to CodeDog. We investigate how code watchdogs can successfully detect the modification attack for mobile devices of different mobilities. We set up a realistic ZigBee network simulation based on [7]. The packet size is 127 bytes, and software update traffic is generated in a constant bit rate (CBR) of 10 packets/s. According to the findings of [7], we assume the delivery ratio of packets is 0.4 with mesh routing. A mobile IoT device in a ZigBee network is randomly located within the communication range of a watchdog and then moves in the random waypoint model. It randomly selects the next waypoint and its speed under a given maximum speed. We assume the communication range of a code watchdog is the same as that of a mobile node. For simplicity, we also assume a code watchdog can overhear packets of mobile nodes within its communication range. Figure 7 depicts the inclusion rate of the fragments collected by a single code watchdog with different maximum speeds of mobile nodes. The result is averaged out from 100 simulation runs each. With low mobility, the code watchdog can successfully verify more than 92 % of fragments but the inclusion rate decreases inverse proportionally to mobility. However, even in a high mobility of 29.5 m/s (>100 km/h), CodeDog can verify more than 68 % packets with a single code watchdog. It can be improved with collaborative, multiple code watchdogs.

6 Related work and discussions

6.1 IDS and software updates

Traditional IDS systems discover the attacks based on well-defined rules. Khamphakdee et al. extend the Snort rules for network probe attacks [8]. Besides predefined rules, IDS researches focus on the statistics on packets to detect abnormality in networks. Onat and Miri identify attacks based on the statistics information from the last N packets [9]. Alam et al. also addressed the problems in secure dissemination of the software update [10]. However, it still detects attacks via examining the timing patterns in software update. Like our watchdogs, Mo et al. proposed the Area Agent System, which monitors nearby area to detect intrusions [11]. The coordinator discovers more collaborative attacks occurring in a wide area.

For software updates, a number of researches are proposed to efficiently and securely deliver software updates. Deluge is a protocol for software updates in wireless sensor networks (WSNs) [12]. It takes into account the reliable delivery of large objects in lossy wireless channel. Recently, for IoT devices, the methods for secure software updates are proposed in [13, 14].

6.2 Software attestation

Software attestation is a repeated process to check the modification of software in devices. It is well developed in wireless sensor network to identify compromised sensor nodes. SWATT [15] is a challenge-response protocol

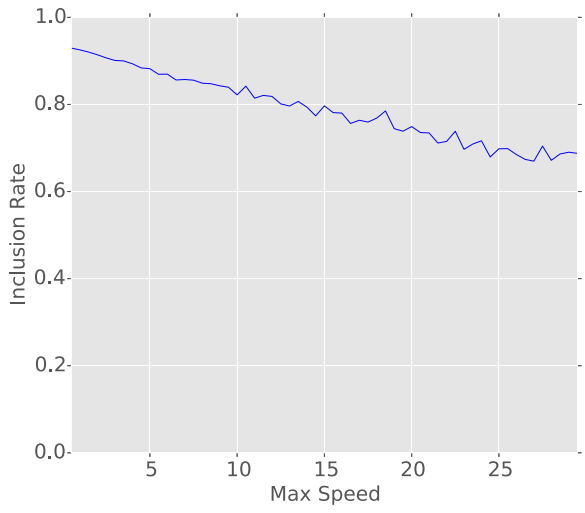

Fig. 7 Mobility impact on the inclusion rate

to attest devices. A device requests the attestation process by sending a challenge to a target device. The target device computes a checksum value from memory contents, which holds code and data, in a pseudo-random access pattern of the seed given by the request. If the testing device can generate the same checksum value, the target device is verified. To improve the security, Pioneer [16] is used for executing code and SCUBA [17] provides a trustworthy platform for executing code on it.

A distributed scheme for software attestation is also proposed in [18]. It calculates the checksum of the program in advance and distributes the partial results to other sensor nodes.

6.3 Discussions

The previous methods have mainly focused on protecting contents from unspecified attacks. Thus, they concerned the integrity of a whole content. Most of the previous approaches assume that they can inspect a whole content at once. However, for in-network monitoring, the assumption is not always true. Only IDS works on the fragments of contents (packets) to detect attempts of various attacks. It generally uses pattern matching with signatures generated from known attack models. This approach is effective for high-speed detection of wide-spectrum attacks, but the predefined signature patterns can be deceived in sophisticated attacks. In that sense, CodeDog is advantageous for securing software updates in mobile networks. It can detect the modification in semantics from the fragments of software objects.

Compared with the previous schemes, CodeDog has a different approach for protecting IoT networks. We focus on detecting mechanisms that can be applied in watchdogs. The detection of attacks may employ watchdogs deployed in networks as in [11], but it is about a whole framework as a holistic perspective rather than detecting method against specified attacks. CodeDog provides an effective method to the limited modification attack for watchdogs. The code watchdogs of CodeDog verifies the control semantics that reflects the actual behaviors of the software. The software update methods, such as [12–14], securely deliver special payloads, i.e., software objects; however, they still tackle syntactic integrity, not semantics. Even if we cannot keep safe the deployment chain, the software attestation schemes on a WSN can check the integrity of running programs after deploying software updates. The schemes of [15–18] assume the sensor nodes in a WSN are identical or to get the complete memory image of a target device in validating checksum. However, mobile IoT networks consist of heterogeneous IoT devices. The previous methods, in that case, have to handle a number of possibilities in combination of software objects and platforms. Thus, they are not suitable for our case. The distributed approach of [18] can provide improved security, but the checksum of software objects is hard to update once the checksum is distributed among sensor nodes. The quirk of IoT networks, frequent software update, does not go with the limitation.

7 Conclusions

Secure software dissemination is critical in future wireless networks. In-network monitoring is a preferred approach for protecting low-end devices from the attacks by checking the integrity of software update traffic. However, it is difficult to get complete information for in-network monitors on all software updates. Thus, with partial information, the in-network monitors can be fooled in verification of software updates. CodeDog presents a mechanism to solve this problem. We implant the semantics markers to the binary code. The semantics markers are locked in the position and indicate the control flows have been unchanged. Hence, they should be valid in binary code fragments, i.e., packets, as well as a whole program code. The code watchdog, the in-network monitor of CodeDog, inspects the software update packets to check the integrity of control flow semantics. CodeDog provides an effective protection even in a fragment of a single packet with 4.2 % storage overhead. In this paper, we focus on the integrity of control flow for preserving semantics in software updates of WSNs, but the integrity of data flow is also important for the software update protection. For our future work, we are working on a method to efficiently check the integrity of both control and data flows.

Endnotes

[1] We only use `while` to represent loops since `for` is a syntactic sugar of `while`.

[2] The type of markers can be exchanged at the marker injection step. See Section 3.2.

[3] In CIL, the `Loop` statement represents all kinds of loops in C.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF-2015-R1A2A2A01004792). This work was also supported in part by the Gachon University research fund of 2015 (GCU-2015-0054).

Competing interests

The authors declare that they have no competing interests.

Author details

[1] Department of Mathematical Finance, Gachon University, 1342 Seongnamdaero, Seongnam-si, South Korea. [2] Graduate School of Information, Yonsei University, 50 Yonseiro, Seoul, South Korea.

Received: 1 June 2016 Accepted: 2 October 2016
Published online: 19 October 2016

References

1. Command Five. SK hack by an advanced persistent threat (2011). http://www.commandfive.com/papers/C5_APT_SKHack.pdf
2. P Craig, *Software Piracy Exposed—Secrets from the Dark Side Revealed*, 1st edn. (Syngress Publishing, Rockland, 2005)

3. J Zhao, N Yao, S Cai, in *Computer Science and Information Engineering, 2009 WRI World Congress On*. A new method to protect software from cracking, vol. 2 (IEEE, New York, 2009), pp. 636–638
4. C Eagle, *The IDA Pro Book: the Unofficial Guide to the World's Most Popular Disassembler*. (No Starch Press, San Francisco, 2011)
5. M Abadi, M Budiu, Ú Erlingsson, J Ligatti, Control-flow integrity principles, implementations, and applications. Trans. Inform. Syst. Secur. (TISSEC). **13**(1), 4:1–4.40 (2009)
6. GC Necula, S McPeak, SP Rahul, W Weimer, in *International Conference on Compiler Construction*. CIL: intermediate language and tools for analysis and transformation of C programs (Springer Berlin Heidelberg, 2002), pp. 213–228
7. L-J Chen, T Sun, N-C Liang, An evaluation study of mobility support in ZigBee networks. J. Signal Process. Syst. **59**(1), 111–122 (2010)
8. N Khamphakdee, N Benjamas, S Saiyod, in *Information and Communication Technology (ICoICT), 2014 2nd International Conference On*. Improving intrusion detection system based on snort rules for network probe attack detection (IEEE, New York, 2014), pp. 69–74
9. I Onat, A Miri, in *Wireless And Mobile Computing, Networking And Communications, 2005.(WiMob'2005), IEEE International Conference On*. An intrusion detection system for wireless sensor networks, vol. 3 (IEEE, New York, 2005), pp. 253–259
10. A Ashraful Alam, D Eyers, Z Huang, in *Automation, Robotics and Applications (ICARA), 2015 6th International Conference On*. Helping secure robots in WSN environments by monitoring WSN software updates for intrusions (IEEE, New York, 2015), pp. 223–229
11. Y Mo, Y Ma, L Xu, in *IT in Medicine and Education, 2008. ITME 2008. IEEE International Symposium On*. Design and implementation of intrusion detection based on mobile agents (IEEE, New York, 2008), pp. 278–281
12. PK Dutta, JW Hui, DC Chu, DE Culler, in *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*. Securing the deluge network programming system (ACM, New York, 2006), pp. 326–333
13. J Liu, W Tong, in *Internet of Things (iThings/CPSCom), 2011 International Conference on and 4th International Conference on Cyber, Physical and Social Computing*. A framework for dynamic updating of service pack in the internet of things (IEEE, New York, 2011), pp. 33–42
14. SG Hong, NS Kim, T Heo, in *Consumer Electronics (ISCE), 2015 IEEE International Symposium On*. A smartphone connected software updating framework for IoT devices (IEEE, New York, 2015), pp. 1–2
15. A Seshadri, A Perrig, L van Doorn, P Khosla, in *SP '04: Proceedings of the 2004 IEEE Symposium on Security and Privacy*. SWATT: software-based attestation for embedded devices (IEEE, Oakland, 2004), pp. 272–282
16. A Seshadri, M Luk, E Shi, A Perrig, L van Doorn, P Khosla, Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. ACM SIGOPS Oper. Syst. Rev. **39**(5), 1–16 (2005)
17. A Seshadri, M Luk, A Perrig, L van Doorn, P Khosla, in *WiSe '06: Proceedings of the 5th ACM Workshop on Wireless Security*. SCUBA: Secure Code Update By Attestation in sensor networks (ACM, New York, 2006), pp. 85–94
18. S Kiyomoto, Y Miyake, Lightweight attestation scheme for wireless sensor network. International Journal of Security and Its Applications. **8**(2), 25–40 (2014)