Data Exfiltration from Air-Gapped Computers based on ARM CPU

Kenta Yamamoto, Miyuki Hirose, and Taiichi Saito
Tokyo Denki University
5 Senju-Asahi-Cho, Adachi-Ku, Tokyo 120-8551, Japan
Tokyo, Japan

***Abstract*—Air-gapped Network is a network isolated from public networks. Several techniques of data exfiltration from air-gapped networks have been recently proposed. Air-gap malware is a malware that breaks the isolation of an air-gapped computer using air-gap covert channels, which extract information from air-gapped computers running on air-gap networks. Guri et al. presented an air-gap malware "GSMem", which can exfiltrate data from air-gapped computers over GSM frequencies, 850 MHz to 900MHz. GSMem makes it possible to send data using the radio waves leaked out from the system bus between CPU and RAM. It generates binary amplitude shift keying (B-ASK) modulated waves with x86 SIMD instruction. In order to efficiently emit electromagnetic waves from the system-bus, it is necessary to access the RAM without being affected by the CPU caches. GSMem adopts an instruction that writes data without accessing CPU cache in Intel CPU. This paper proposes an air-gap covert channel for computers based on ARM CPU, which includes a software algorithm that can effectively cause cache misses. It is also a technique to use NEON instructions and transmit B-ASK modulated data by radio waves radiated from ARM based computer (e.g. Raspberry Pi 3). The experiment shows that the proposed program sends binary data using radio waves (about 1000kHz ~ 1700kHz) leaked out from system-bus between ARM CPU and RAM. The program can also run on Android machines based on ARM CPU (e.g. ASUS Zenpad 3S 10 and OnePlus 3).**

Keywords*—Air-Gapped Network; ARM CPU; data exfiltration; SIMD; NEON; GSMem

I. Introduction

Air-gapped network is a network physically separated from other unsecured networks, and *air-gapped computer* is a computer in an air-gap network. Industrial control systems and security protection systems are often constructed in the air-gapped networks in which data leakage is prevented by restricting the use of Wi-Fi and Bluetooth and the access to removable storage such as USB flash drive.

Air-gap malware is a malware that breaks the isolation of an air-gapped computer using *air-gap covert channels*, which extract information from air-gapped computers running on air-gap networks.

EMSEC (Emission Security) [1] is an approach against attack using electromagnetic waves leaked from the computer. Usually, a computer emits various energy by data communication such as Wi-Fi or Bluetooth. However, more energy emissions are generated than what the user aware. For example, the fact that an electric current flows in a base board or a wiring may itself be an antenna. TEMPEST [2] is a specification of the defense technology against an attacker to exploit and techniques to steal information by using the emitted energy, such as electromagnetic waves or sound by the National Security Agency. In 1985, Van Eck [3] showed that electromagnetic radiation of monitor can be captured and image can be reconstructed using inexpensive devices as concrete exploit method of TEMPEST. Kuhn and Anderson demonstrated that the emission of electromagnetic radiation emitted from desktop computers can be controlled by software [4], [5]. Several methods of air-gap covert channels have been recently proposed. Mordechai et al. [6] proposed a method of transmitting data through an air gap using electromagnetic waves from a display cable. Hanspach et al. [7] proposed a method of transmitting data through an air gap using ultrasonic waves from a speaker.

Guri et al. a research team at Ben-Gurion University, presented an air-gap malware GSMem [8]. It generates electromagnetic waves from the memory bus between CPU and RAM to transmit B-ASK modulated signals in Intel architecture. GSMem executes an x86 SIMD instruction that exhausts the memory bus bandwidth to accelerate leakage of electromagnetic waves from the memory bus.

This paper presents an air-gapped covert channel on air-gapped computers based on ARM architecture CPU through which B-ASK modulated signals are transmitted over the AM frequency band (1,000 kHz - 1,600 kHz). GSMem adopts the particular instruction in x86 SIMD instruction set which can manipulate data without going through the CPU cache to efficiently access the memory bus, in order to transmit the modulated signal on Intel-based computer. On the other hand, since the CPU of the ARM architecture does not support instructions of application level to bypass the CPU cache, this paper proposes an algorithm that directly accesses memory avoiding CPU cache hit as much as possible and uses a NEON instruction that efficiently occupies the bandwidth of memory bus. The experiment executes the program that adopts the algorithm and shows the electromagnetic waves leaked from ARM computer with a spectrum analyzer.

A poster presentation and preliminary version of this paper were presented at IWSEC 2017 and CSS 2017, respectively.
The authors declare that there is no conflict of interest regarding the publication of this paper.

The remainder of this paper is organized as follows: Section 2 describes the basic technology to understand the proposed method in this paper. Next, Section 3 presents assorted related works. Section 4 describes a main method. Section 5 presents the result of measurement. Finally, Section 6 concludes this paper.

II. Basic Technical Outline

This section provides a basic technical information to make it easier for readers to understand technical parts in the algorithm proposed by the previous researches and the proposed method.

A. B-ASK Modulation

Binary amplitude shift keying (B-ASK) modulation is one of the modulation techniques for transmitting digital signals. It changes the amplitude corresponding to the transmitting binary data. B-ASK uses only amplitude modulation, and the frequency and phase are fixed. When the bit to be transmitted is "1", the amplitude of the carrier wave becomes large, and when the bit to be transmitted is "0", the amplitude of the carrier wave becomes small. Fig. 1 shows an example of an amplitude representation of bit string to be transmitted. Fig. 2 is an illustration of a signal to be sent (upper) and a signal modulated with B-ASK (lower).

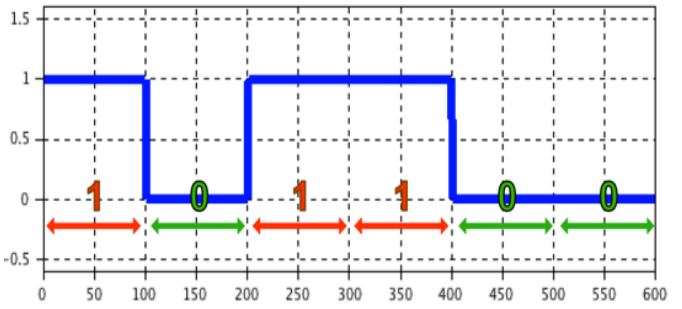

Fig. 1. Example of signal change according to bit string by B-ASK.

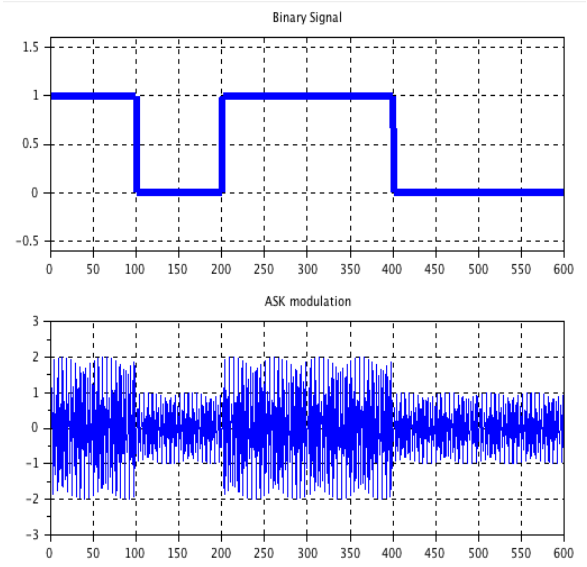

Fig. 2. The signal to be transmitted (upper) and the modulated signal (lower).

B. CPU Cache

CPU cache is a small capacity memory installed in the CPU to hide the low-speed memory bus when the CPU transfers data to the memory. The bandwidth of the memory bus cannot catch up with the processing capability of the CPU, which becomes a bottleneck. CPU cache works as a very high-speed memory that holds the data and addresses of the memory accessed by the CPU in order to complement the performance difference between the CPU and RAM. In the case that the CPU accesses the data stored in the cache memory, it does not need to go through the memory bus, and therefore it is possible to avoid the bottleneck. CPU cache has various features: capacity, speed, data update method, etc. depending on the architecture. However, since the hardware automatically operates the cache, an application software needs not to control it. Although it is a high-speed memory, CPU cannot have a large-capacity cache memory like RAM. In many cases, the CPU cache has a multistage structure in order to support with increasing the capacity of the RAM and multi-core. They are called Level 1 (L1), Level 2 (L2), Level 3 (L3) in the order closest to the CPU core. Most of the L1 caches are installed for each CPU core, and the L1 cache is the fastest cache memory with the smallest capacity. L2 and L3 caches are cache memories with larger capacity, and they are shared with all CPU cores.

In Intel-based CPU for individual products, they have L1 and L2 caches about 32KB-256KB in each cores, and L3 as shared cache of up to 8MB. Cortex-A53, one of the ARM architectures, has up to 64KiB of L1 cache and 2MiB of L2 cache in each core.

In either architecture, in the L1 cache, it is divided into an instruction cache and a data cache. The instruction cache stores CPU instruction groups included in the program, and the data cache is an area for storing data processed by the program.

The CPU cache is stored by a unit called *line* and associated with the tag generated from the memory address. The line length in the ARM architecture is eight words [9], [23].

C. SIMD Instructions

Single Instruction Multiple Data (SIMD) is an instruction set that can manipulate multiple data with a single instruction. In an algorithm capable of parallelization, it is possible to increase the effective speed of a program by processing a plurality of data with one clock by using SIMD. It is effective in an algorithm that parallelly calculates data equal to or larger than the data width supported by the processor in the general-purpose instruction set. Usually, the data width supported by the instruction set is divided and used for parallel calculation. For example, when the SIMD instruction set supports a calculation width of up to 128-bit, it is common to use a method of calculating four 32-bit floating point operations in parallel.

The Intel CPU can use the SSE instruction set, AVX, etc. the SSE-based instruction set supports to use up to 128-bit and AVX supports up to 256-bit.

Some ARM architectures support SIMD instructions called NEON. NEON supports up to 128-bit and is available in the Cortex-A family.

III. Related Work

A. GSMem

GSMem presented by Guri et al., is a malware that exfiltrates data from an air-gapped computer based on x86 CPU architecture. It utilizes the phenomenon that electromagnetic waves over the GSM frequency band are radiated when the CPU accesses RAM via the memory bus. GSMem malware infects a computer and performs it as a transmitter that sends a modulated signal by B-ASK over the GSM frequency band. The B-ASK modulation is a method of sending digital data by changing the amplitude of carrier wave according to binary symbols "1" and "0". GSMem involves much RAM access to generate large amplitude and less access to generate small amplitude. The amplitude of leaked wave becomes significantly large at GSM frequencies when memory access occurs. The receiver tries to demodulate leaked wave at GSM frequencies measuring the amplitude of it.

Guru et al. use the following technique for amplifying radiation of electromagnetic wave when accessing to RAM. They require the algorithm to involve direct RAM access avoiding CPU cache. Their technique uses an SSE2 instruction *"MOVNTDQ"* [10] on x86 architecture CPU in order to produce efficient access to RAM. MOVNTDQ is an instruction that stores data to RAM bypassing CPU caches. This is named _mm_stream_si128 [11] in C++ library.

Algorithm 1 is a concept code which sends B-ASK modulated data by GSMem. This algorithm reads a binary data from an element of array ***data*** at index ***bit_index***, and if it is '1', the algorithm repeats executing MOVNTDQ for ***tx_time*** nanoseconds. If the bit is '0', the algorithm sleeps for the same period.

Algorithm 1. Transmit data by the GSMem

```
1: buffer ← ALIGNED_ALLOCATE(16,4096)
2: tx_time ← 500000
3: for bit_index ← 0 to 32 do
4:     if (data[bit_index] == 1) then
5:         start_time ← CURRENT_TIME()
6:         while (tx_time > CURRENT_TIME () - start_time) do
7:             buffer_ptr ← buffer
8:             for i ← 0 to buffer_size do
9:                 MOVNTDQ (buffer_ptr, 128bit_register)
10:                buffer_ptr ← buffer_ptr + 16
11:            end for
12:        end while
13:    else
14:        SLEEP (tx_time)
15:    end if
16: end for
```

B. System-Bus-Radio

William Entriken (github.com: fulldecent) presented the program *"System-bus-radio"*[1] that generates an AM frequency carrier wave amplification-modulated with sound data to be reproduced by a loudspeaker. System-bus-radio uses an algorithm similar to GSMem. It calculates "*period*" from audio frequency. It produces audio wave of frequency *f* by generating alternately "1" and "0". Fig. 3 shows an example of generating audio wave by calculated "*period*" from frequency *f*.

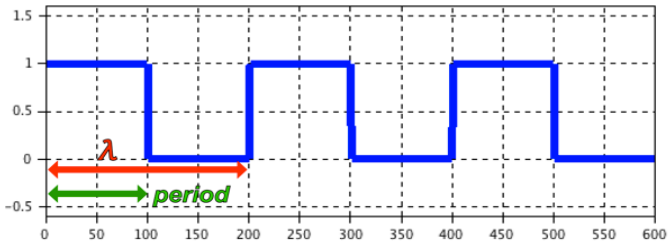

Fig. 3. System-bus-radio calculates "*period*" from $f = 1/\lambda$ to obtain audio wave with frequency *f*. For example, if it wants sound of 2000 Hz, it calculates $period = 1/2\lambda = f/2 = 0.00025$ seconds.

Even a common AM radio can be used to receive a signal as a receiver in System-bus-radio. Since the signal is amplitude-modulated, the AM radio can receive and demodulate it to reproduce the sound. System-bus-radio uses SSE2 instruction MOVNTDQ available on x86 architecture CPU.

IV. Proposed Method

This section proposes an algorithm that transmits B-ASK modulated signal in ARM-architecture-based computers. In GSMem, Guri et al. adopted the SIMD instruction, MOVNTDQ in x86 architecture CPU. They showed that the instruction can efficiently and massively send data to the memory bus. Since ARM, however, has a different microarchitecture from x86, the instruction cannot be used. The proposed algorithm adopts a SIMD instruction that efficiently accesses the memory bus in ARM CPU based computer.

A. ARM Architecture

Low Power Double Data Rate (LPDDR) [12], [13] is a power saving standard of DDR memory. LPDDR memory is often adopted for many devices with ARM architecture CPU for power saving. Since the LPDDR memory has a 32-bit bandwidth bus, it is possible to occupy the memory bus even by using a 32-bit or 64-bit general-purpose instruction set. However, if the computer is in a multi-channel environment, any instruction in the general purpose instruction set cannot occupy the memory bus bandwidth.

In this paper, NEON instruction set is adopted for occupying the memory bus more certainly in a computer based on ARM CPU. NEON is a SIMD instruction set that is available on ARM architecture CPU. It can be used on ARMv7 and above, and operate 64-bit or 128-bit data.

[1] William Entriken (fulldecent). (2017) System-bus-radio, [Online]. [Accessed 25 October 2017]. https://github.com/fulldecent/system-bus-radio/.

Therefore, it is suitable for bandwidth occupation of the memory bus.

The instructions for operating data without using the CPU cache are equipped in x86 SIMD. However, in the NEON instruction set, there is no instruction to operate data explicitly without using the CPU cache. The ARM CPU has an instruction cache and a data cache, also has CPU modes disabling each cache. However, since the CPU's privileged mode is required to switch modes, invalidating the CPU cache is not a practical way for malware running in user application level. Therefore, it is needed to build an algorithm that generates access to RAM even if the CPU cache is valid, and select the optimal instruction from the NEON instruction set. The next section proposes an algorithm to avoid CPU data cache hit.

B. Algorithm

In order to avoid reading cached data, it is only necessary that the data is not stored in data cache when the CPU refers to it. Here a concept code realizing an algorithm to avoid CPU data cache in ARM CPU is presented in **Algorithm 2**. This code modulates the electromagnetic waves from memory bus by B-ASK and transmits 4-bit data "1010". In order to transmit '1', it is necessary to repeat accessing the memory bus for a predetermined time to radiate electromagnetic waves. The algorithm repeats loading data on memory and releases electromagnetic radiation from the memory bus.

Algorithm 2 is a simple algorithm for loading data from memory, but it changes the address of data to be loaded every time the load instruction is executed. CPU operates different data on different location each time. This trick makes it difficult to hit the CPU data cache.

Algorithm 2. A new algorithm for ARM computers

```
 1: p = (int32_t *)malloc(size)
 2: for (int i=0; i<=n; i++):
 3:   p[i] = i;
 4:
 5: data_bits[] = {1, 0, 1, 0,}
 6: period = 500000
 7: for data in data_bits :
 8:   if (data == 1):
 9:     i = 0
10:       start = now()
11:       while (period > now() - start):
12:         va = vld1q_s32(p+i)
13:         i+=8
14:         if(i==limit) i=0
15:   if (data == 0):
16:     sleep(period)
```

As a SIMD instruction executed for loading data, the proposed algorithm adopts *"VLD1.32"* [14] which allows more data to be transferred than a general-purpose instruction set. It is implemented as the function *"vld1q_s32"* in C ++ language library. *VLD1.32* is an instruction to load four 32-bit data stored in the RAM as a single vector into a register. It allows loading 128-bits data at a time.

This algorithm separates into a part of allocating a large array in memory and the other part of controlling the radiation electromagnetic waves from memory bus. When it executes RAM access, the CPU repeats loading the data allocated in the memory.

In Algorithm 2, the first line allocates the memory. It reserves *"size"* bytes of 32-bit data array when working on a 32-bit system. The size is needed to be at least larger than the L1 cache for the algorithm to effectively work. For example, the Cortex-A53 architecture, which is one of the ARM Cortex-A series installed in Raspberry pi 3, can have an L1 cache up to 64 KiB, so the size is needed to be larger than 64 KiB.

The lines 2-3 initialize each element of the array with each distinct value. *"data_bits"* is an array of binary data to be transmitted. In this case, the data *"1010"* is sent.

The "*period*" means 500 microseconds. According to GSMem, if the *period* decreased, a higher bit rate is obtained, but the error rate is increased. The algorithm sets period to 500 microseconds in our algorithm like GSMem's one.

In the line 7, when a data in *data_bits* is 1, the block starts ***while*** loop that performs memory operations and generates electromagnetic wave.

In the line 12, it loads the data into the registers as a single vector from the four addresses using the *"vld1q_32"* instruction. Then it adds 8 to the variable *"i"*. It is the index to select memory address. That is, the index of the memory address next to be loaded is shifted by 8. This integer 8 means the size of the cache line in the ARM CPU. If *"i"* reaches the overflow value, set it to 0. The program repeatedly executes the code written on the lines 12-14 for the *"period"* time, where electromagnetic waves are generated from the memory bus.

In the last line, the algorithm sleeps for "period" microseconds when outputting "0". The amplitude of electromagnetic waves from memory bus becomes small.

C. Details of Avoiding Cache

Once the CPU accesses a location in main memory, the data around the location is stored in the CPU cache in units of 8 words. Even when the CPU reads data of index 0 to 3 in an array, there is a possibility that data of index 0 to 7 is stored in the cache. Accordingly, on every memory access, the algorithm needs to read data at least 8 words far from the previously accessed data.

Fig. 4 shows the first action of loading four elements from an array, and Fig. 5 shows the next action of loading the four elements located 8 words far from the first ones in the array. Because the algorithm allocates an array larger than the total size of the CPU caches, the data in elements initialized in early stage have already been removed from the caches and then the actions success in causing cache miss and loading data directly from RAM. Also, by loading, CPU generates new data caches. To prevent loading from the same cache line, the algorithm shifts 8 words the position to be loaded each time.

The algorithm is described based on virtual memory terms and it is assumed that the virtual memory is almost not

fragmented. However, even though the virtual memory is fragmented and mapped to the physical memory, there is little influence for the purpose of outputting electromagnetic waves.

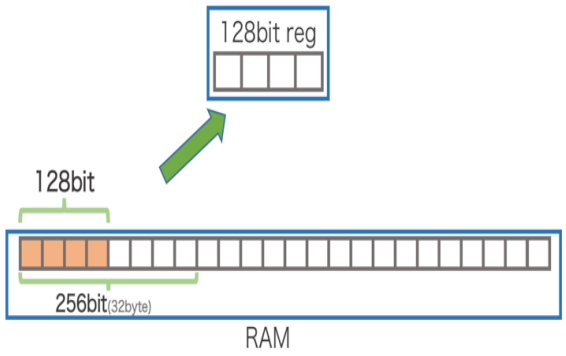

Fig. 4. The action of the first loading elements from memory to 128-bit register. Algorithm selects four elements in index 0-3.

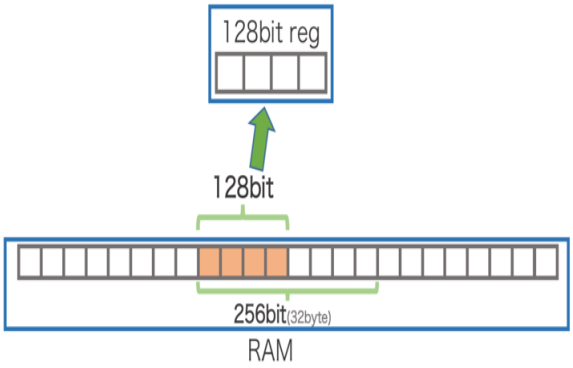

Fig. 5. The next action of loading elements. Algorithm selects four elements in index 8-11.

V. Measurements

In order to verify effects of the proposed algorithm in this section, frequency and time characteristics of the leaky electromagnetic wave from devices, based on ARM CPU, implemented the program described in the previous section, were measured by a spectrum analyzer and an oscilloscope.

A. Frequency Domain

1) Experimental arrangement

Based on the algorithm in the previous section, a data transmission program was created. It was forked from System-bus-radio made by William Entriken. This section shows the frequency characteristics when the program is executed on the computer based ARM CPU. Raspberry Pi 3 and ASUS Zenpad 3S 10 Z500M (Z500M) were used for the measurements.

Raspberry Pi 3 is running with Broadcom BCM2837[2], which is one in a series of ARM architecture. The BCM2837 has four ARM Cortex-A53[3] CPU cores. Also Raspberry Pi 3 has 1 GB of LPDDR2 SDRAM. Raspbian Linux for experiment OS is used in this experiment.

Z500M uses Android 7.0 as operating system. Z500M is running on MediaTek MT8176[4]. Internally, the MT8176 has two cores of ARM Cortex-A72[5] and four Cortex-A53. Z500M is installed with 4 GB RAM. The RAM is connected to CPU by dual channel.

The measurement campaigns were conducted in a radio anechoic chamber. The receiving antenna was an omnidirectional, vertically polarized mono-pole antenna. Frequency-domain propagation gains were measured with a spectrum analyzer, as shown in Fig. 6. The DUT (Device Under Test) were Raspberry Pi3 and Z500M.

2) Results and analysis

The frequency-domain gains of Raspberry pi 3 and Z500M are shown in Fig. 7 and 8, respectively. The blue lines stand for ON state, the proposed program is executing. The black lines stand for OFF state, program is not executing. While a signal was found in ON state, no one was yielded practically in OFF state. From the measurement data, the peak (the highest amplitude value) of signals at approximately 1.5 MHz was observed in case of Raspberry Pi 3, and the peaks ranged 1.2 – 1.4 MHz in case of Z500M. In both cases, peak gains were approximately 4 dB.

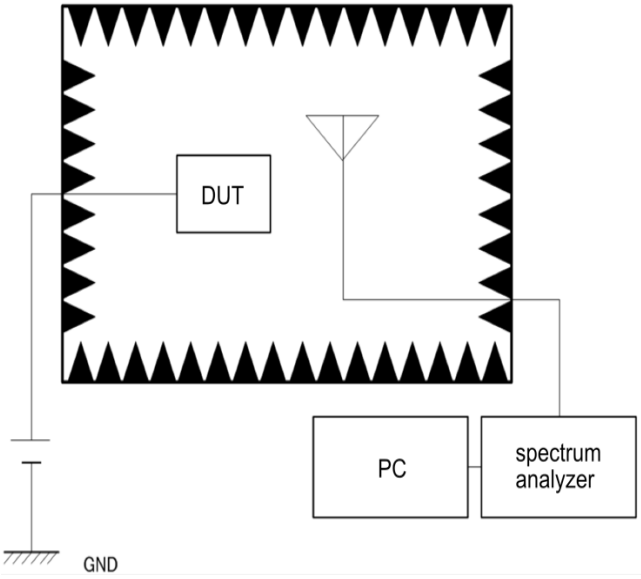

Fig. 6. Experiment setup.

[2] RASPBERRY PI FOUNDATION (2017), Raspberry Pi 3 Model B - Raspberry Pi, [Online]. [Accessed 25 October 2017] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[3] ARM Ltd., "ARM® Cortex®-A53 MPCore Processor Revision: r0p4 Technical Reference Manual", [Online]. [Accessed 15 November 2017], http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500g/DDI0500G_cortex_a53_trm.pdf

[4] MediaTek Inc. (2016), MediaTek MT8176 for Tablets | MediaTek, [Online]. [Accessed 27 October 2017] https://www.mediatek.com/products/tablets/mt8176

[5] ARM Ltd., "ARM® Cortex®-A72 MPCore Processor Revision: r0p1 Technical Reference Manual", [Online]. [Accessed 15 November 2017], http://infocenter.arm.com/help/topic/com.arm.doc.100095_0001_02_en/cortex_a72_mpcore_trm_100095_0001_02_en.pdf

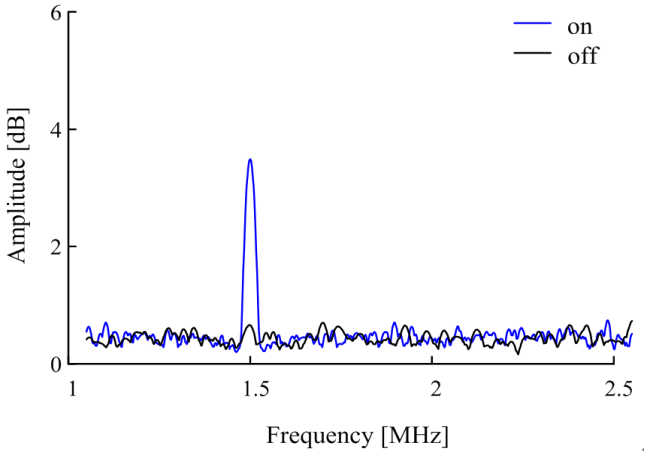

Fig. 7. An example of frequency-domain gains.(Raspberry Pi 3). *"on"* is in the program execution state. *"off"* is the state in which the program is not running.

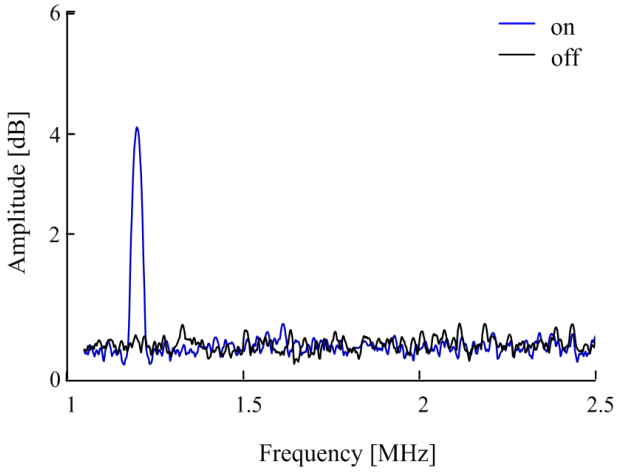

Fig. 8. An example of measured frequency spectrum.(ASUS Zenpad 3S 10 Z 500 M). *"on"* is in the program execution state. *"off"* is the state in which the program is not running.

B. Time Domain

1) Experimental arrangement

The waveform in the time domain of the received signal with the same configuration was measured as in Fig. 6. The experiment was set up to analyze the time domain of the leaky waves, as show in Fig. 9. During the measurement, the oscilloscope was placed outside the radio anechoic chamber. The receiving antenna was an omnidirectional, vertically polarized mono-pole antenna. The DUT was Raspberry Pi3.

2) Results and analysis

Fig. 10 shows the waveform in the time domain of the received signal, and Fig. 11 shows the signal and the envelope of the received signal calculated from the measurement data. From the measured data, the received signal was performed the amplitude modulated. Also, the power of the envelope was distorted because of the noise from the DUT.

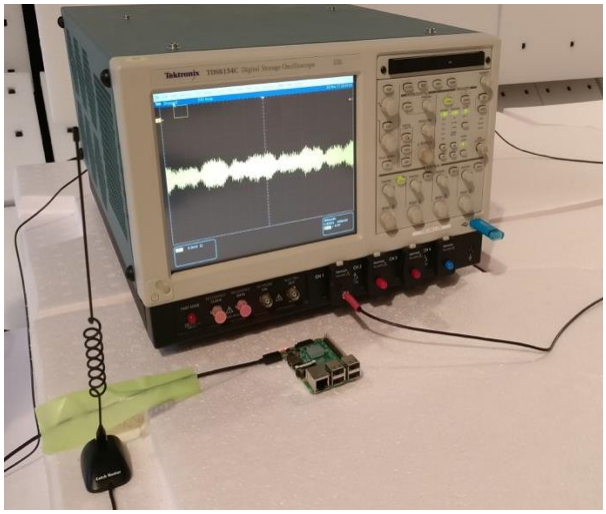

Fig. 9. The picture of measurement setup. (During measurement, the oscilloscope was placed outside the radio anechoic chamber.)

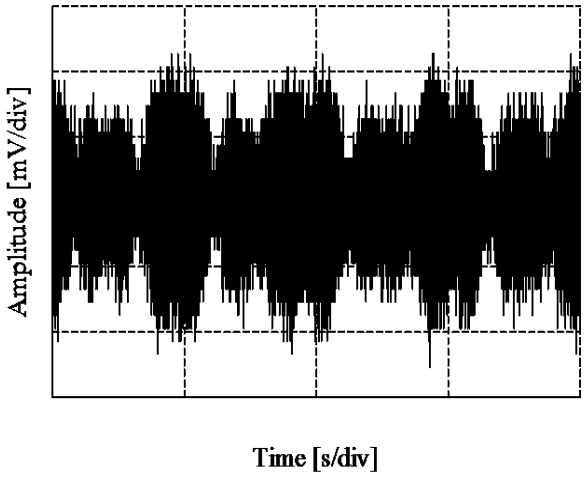

Fig. 10. An example of the waveform of the received signal (Raspberry pi 3).

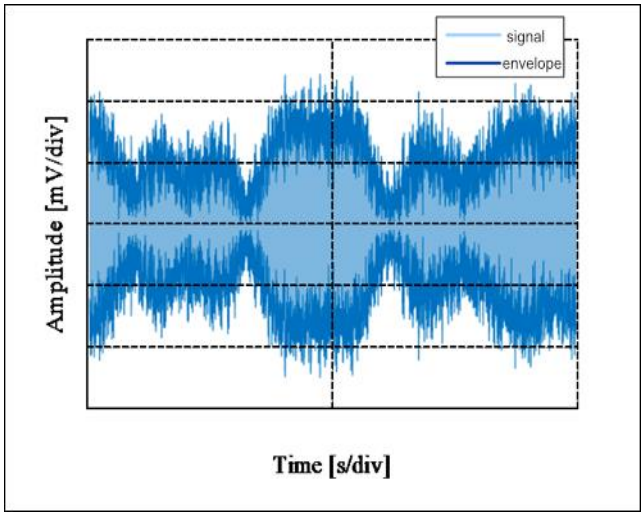

Fig. 11. The waveform and the envelope of the received signal (Raspberry Pi3).

TABLE I. A COMPARISON OF GSMEM AND PROPOSED ALGORITHM IN THIS PAPER

	Frequency	Platform	Memory usage	Languages
GSMem	GSM band (800MHz -)	Intel based PC	Low (~KB)	C++
Proposed algorithm	AM band (1.0 - 1.6MHz)	ARM and other CPUs	Medium (~MB)	C++, javaScript(asm.js)

VI. DISCUSSION

The results shown in the previous section shows that the proposed algorithm is able to emit electromagnetic waves from the device (Raspberry Pi 3). However, since the ARM architecture is adopted for power saving devices, there is a possibility that the power of the emitted electromagnetic wave becomes smaller than GSMem.

Table I also shows a comparison between the proposed algorithm and GSMem. In GSMem, the frequency of electromagnetic waves emitted is the GSM frequency band, and the receiver is a modified mobile phone. In the proposed algorithm, the frequency of the emitted electromagnetic waves is AM frequency band, and ordinary AM radio can be used as the receiver.

The environment in which GSMem operates is limited to computers equipped with Intel CPU. CPU instructions used by GSMem are unique instructions not found in other architectures, and it is difficult to replace them. On the other hand, the proposed algorithm operates on a computer equipped with an ARM CPU. Furthermore, it can be used also on other platforms simply by replacing ARM specific instructions. This algorithm does not depend on the architecture, and it can be replaced with CPU instructions with memory access. The authors are doing work to operate this algorithm with a web browser.

The proposed algorithm requires assigning a memory greatly exceeding the capacity of the CPU cache. It involves more memory consumption than GSMem's algorithm.

VII. CONCLUSION

This paper presented a new algorithm to exfiltrate data from ARM-based computers causing electromagnetic wave radiation, while Guri et al. presented the algorithm on Intel-based computers. Since the proposed algorithm accesses memory aggressively, the electromagnetic waves are radiated.

Frequency response of the electromagnetic wave is measured when the proposed algorithm is running with Raspberry Pi 3 and ASUS Zenpad 3S 10 Z500.

Guri et al. adopted an x86 CPU instruction that stores data in RAM explicitly bypassing the CPU cache. On the other hand, since SIMD of the ARM architecture has no instructions to bypass the CPU cache, the SIMD instruction (VLD 1.32) was adopted to load the data and proposed the algorithm that avoids CPU data cache hit by making data to be loaded different each time VLD 1.32 instruction is executed.

The measurement is concluded and it is confirmed that the program implementing the proposed algorithm is able to successfully radiate electromagnetic waves.

System administrator who manages ARM-based air-gap computers also needs to consider covert channels using electromagnetic radiation. Furthermore, the ARM computer is compatible with the mobile computer. Attackers will also be able to extract data without hacking the network.

Although GSMem used a specific SIMD instruction to bypass the CPU cache, this algorithm does not have to use such an instruction. The algorithm can also use generic instruction set instead of SIMD instruction. It has a disadvantage of requiring more memory than the capacity of the CPU cache, but it is effective in environments where the cache cannot be ignored. And I think that this algorithm can also be used in web browser and mobile OS, for example. It does not matter what kind of CPU these platforms are using. In future research, we will work on making programs applying the algorithm proposed in this paper work on web browsers and others.

REFERENCES

[1] R. J. Anderson, "Emission security," in Security Engineering, 2nd Edition, Wiley Publishing, Inc., 2008, pp. 523-546.

[2] Rick Lehtinen, Howard Hecht, Deborah Russell, G. T. Gangemi, "Computer Security Basics: Computer Security", Oreilly & Associates Inc, pp.256, 2006.

[3] W. van Eck, "Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk? , " Computers and Security 4, pp. 269-286, 1985.

[4] M. G. Kuhn and R. J. Anderson, "Soft tempest: Hidden data transmission using electromagnetic emanations," in Information Hiding, 1998, pp. 124--142.

[5] M. G. Kuhn, "Compromising emanations: Eavesdropping risks of computer displays," University of Cambridge, Computer Laboratory, 2003.

[6] G. Mordechai, G. Kedma, A. Kachlon and Y. Elovici, "AirHopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies," in Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on, IEEE, 2014, pp. 58-67.

[7] M. Hanspach and M. Goetz, "On Covert Acoustical Mesh Networks in Air.," Journal of Communications, vol. 8, 2013.

[8] Guri, M., Kachlon, A., Hasson, O., Kedma, G., Mirsky, Y. and Elovici, Y., (2015). GSMem: data exfiltration from air-gapped computers over GSM frequencies. In 24th USENIX Security Symposium, USENIX Security 15, pp. 849-864.

[9] Sarah Harris, David Harris, "Digital Design and Computer Architecture: ARM Edition", Morgan Kaufmann, pp487-529, 2015.

[10] Rajat Moona, "Assembly Language Programming in GNU/Linux for IA32 Architectures", Prentice-Hall of India Pvt.Ltd, pp.404, 2007.

[11] Richard Gerber, "The Software Optimization Cookbook: High-Performance Recipes for the Intel Architecture (Engineer-To-Engineer)", Intel Press, pp.97, 2002.

[12] JEDEC Solid State Technology Association, "JEDEC Standard: Low Power Double Data Rate 4 (LPDDR4)", JEDEC Standard JESD209-4, Aug 2014.

[13] JEDEC Solid State Technology Association, "JEDEC Standard: DDR4 SDRAM", JEDEC Standard JESD79-4B, Nov 2013.

[14] Bruce Smith, "ARM A32 Assembly Language: 32-Bit ARM, Neon, VFP, Thumb", Bruce Smith Books, 2017.

APPENDIX

The proposed algorithm repeats trying to load four-word vector 8 words far from the previously loaded vector. It successfully generates electromagnetic radiation from memory bus between ARM CPU and RAM in the measurements. However, in order to estimate the efficiency that the algorithm avoids cache hit, this section presents an abstract data cache model and slightly modify the proposed algorithm into an inefficient version that randomly and independently chooses a position in the array and load four

words at the position each time. For simplicity, it is assumed that in mapping from virtual memory to physical memory, paging does not cause fragmentation of the array and page fault. Thus we identify an array in virtual memory with mapped contiguous data in physical memory.

This section considers an abstract data cache model and a modified algorithm as follows. The total amount of data cache is calculated in the form of (the cache line length) * (the number of cache line) * (the number of ways). Here let the cache line size be 32 words, the number of cache lines be 1024 and the number of ways be 2. Then the total amount of data cache is 16384 words. The modified algorithm allocates an array of the size three times as much as the total amount of data cache. For simplicity, it is also assumed that the data cache is used only for storing data in the array. Thus, each cache line stores only elements in the array. The algorithm randomly chooses an address of elements in the array and tries to load four-word data beginning at the address from the array.

If the four-word data does not include any array element stored in the data cache as illustrated in Fig. 12, the algorithm successfully loads the four-word data directly from memory through memory bus, without hitting cached data.

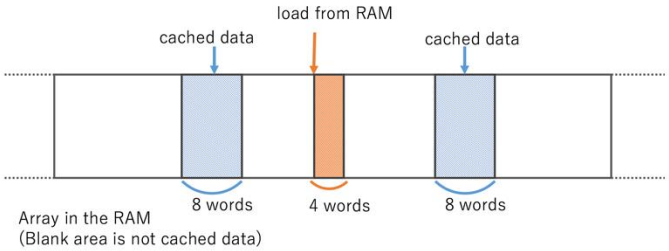

Fig. 12. An example of loading non-cached data from array in the RAM.

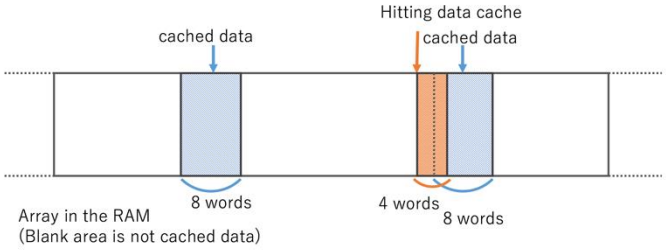

Fig. 13. An example of loading cached data from array in the RAM. Part of the data to be loaded is cached data.

If the four-word data at the randomly chosen address includes at least one array element stored in the data cache as illustrated in Fig. 13, this case is called "hitting data cache" since the algorithm may hit cached data and fail in loading four-word data directly from memory.

So, if the four-word area overlaps a part of the cached area, especially, if the chosen address is point to the position three words right shifted from the cached area, it is also the case of "hitting data cache". In the case that an eight word cached area and a four-word area that the algorithm tries to load overlap each other, the size of the combined area is at most 11 words. Since there exist at most 2048 cached areas of eight words exist in the array, there also may exist 2048 such combined areas of at most 11 words. If there are such combined areas at 2048 locations in the array, and the algorithm tries to access any element in these areas (2048*11 words) in the array (49152 words), then the case of "hitting data cache" occurs.

Since the algorithm randomly chooses four-word area from the array, the probability that it can avoid the case of "hitting data cache" is (49152-22528)/49152. Consequently, the algorithm successfully causes electromagnetic radiation with the probability over 1/2.

Here, in order to estimate the efficiency of algorithm a random algorithm is used and a worst case such that whole data cache is occupied with data in the array is assumed.

However, in reality, since the operating system and device drivers may cause interrupts and many threads concurrently run, it is unlikely that the data cache holds only data from an array. Since electromagnetic waves are constantly radiated in the experiment, it is concluded that the algorithm rarely occurs cache hits in practice.

Algorithm 3. A random algorithm for ARM computers

```
1:  p = (int32_t *)malloc(size)
2:  for (int i=0; i<=n; i++):
3:    p[i] = i;
4:
5:  data_bits[] = {1, 0, 1, 0,}
6:  period = 500000
7:  for data in data_bits :
8:    if (data == 1):
9:      i = 0
10:       start = now()
11:       while (period > now() - start):
12:         va = vld1q_s32( random(0 to n) )   //Random select
13:   if (data == 0):
14:     sleep(period)
```