A Dynamic Detective Method against ROP Attack on ARM Platform

ZhiJun Huang
Software Institute
Nanjing University
Nanjing, 210093, PR China
mg1032004@software.nju.edu.cn

Tao Zheng*
Software Institute
Nanjing University
Nanjing, 210093, PR China
zt@software.nju.edu.cn

Jia Liu
School of Management and Engineering, Software Institute
Nanjing University
Nanjing, 210093, PR China
liujia@software.nju.edu.cn

***Abstract*—with the popularity of embedded devices, especially smart phones, a growing attention has been paid to their programs' security. Many viruses on PC platforms migrated to embedded device have brought new threats to the security of the embedded platform. ROP (Return-Oriented Programming) attack is one of them. At the same time, traditional protective measures on PC platform tend to lose effect in embedded devices due to differences among platforms and architectures which bring significant challenges to virus protection on embedded devices. Defending ROP attack confronts the same problem. Existing protective methods against ROP attack on PC rarely work well on an embedded platform. This paper presents a protective algorithm against ROP virus on the embedded ARM platform. Furthermore, we develop a Valgrind tool to implement this algorithm with dynamic binary instrumentation technology which can effectively prevent the ROP attack and its variants on the ARM platform.**

***Keywords*-ROP, Dynamic Binary Instrument, Embedded Platform, Program Security, Turning-Complete, Control Flow**

I. INTRODUCTION

As a generalization and extension of the Return-Into-Libc attack [1], the idea of ROP [2] poses a new threat to program security. With the mean of malicious use of short sequence fragments of assembly codes of libraries in the computer, the attacker can build a Turing-complete exploit program to achieve any compute and attack. ROP exploit has been mounted on multiple platforms. Hovav Shacham et al [2] conduct a first Turning-Complete ROP attack on the Intel x86-Linux platform. Ralf Hund et al [3] achieve a semi-automatic build method for ROP with DLLs on Windows to conduct rootkit attack. Eric Buchanan et al [4] conduct ROP attack on RISC-based SPARC platform. Stephen Checkoway et al [5,11] mount a non-ret ROP attack which is called JOP(Jump-Oriented Programming),a ROP variant， ending with non-ret instruction, which expands the ROP family. Tyler Bletsch et al [6] and Ping Chen et al [7] work out the automatic build of ROP and JOP on the x86 platform respectively. In addition, Lidner et al [8] mount ROP attack on PowerPC.

With the research deepening, ROP attack on embedded platform has been implemented. Tim Kornau et al [9] first conduct ROP attack on the ARM platform. In that paper, authors elaborate platform specific features which are suitable for ROP attack to mount, and achieve an automatic search algorithm of short instruction sequences for ROP attack gadgets, and provide a build process which makes it possible for ROP exploit on ARM. Lucas Davi et al [10] conduct JOP (Jump-Oriented Programming) on ARM which makes use of indirect jump instructions to control running flow of ROP attack. At the same time, Lucas Davi et al [12] port JOP to the Android system which can circumvent current security model [36,37] and protective measures of Android.

At present, many defense technologies against ROP have emerged; however, almost all of them are on PC platforms but without progress on embedded platforms. Under this circumstance, we propose a defensive algorithm against ROP on ARM and develop a defensive tool with dynamic binary instrumentation framework, Valgrind. In following parts, we will introduce our algorithm and provide details of our implementation. This paper makes the following contributions:(1)We identify and analyze essential features of short sequences of ROP exploits, including traditional ROP exploits and non-traditional ROP exploits on ARM,(2)We propose an effective algorithm to detect ROP exploits,(3)We build a Valgrind tool to implement our solution on ARM Linux platform .

The rest of this paper is organized as follows: Section 2 provides backgrounds of ROP exploits and DBI. Section 3 explains our defensive algorithm and solution against ROP. Section 4 introduces our implementation on ARM Linux. Section 5 evaluates performance and accuracy of our solution, and also presents limitations and explores improvements of our solution. Section 6 covers related works of current ROP exploit defensive solutions and finally Section 7 concludes this paper and introduces future work.

II. BACKGROUNDS

A. ROP Attack and Its Variants

The ROP attack is initially proposed by Hovav Shacham[2]. The essence of ROP is, by extracting short instruction sequences ending with ret instructions with a length of 2-3 from assembly codes of the library in the computer and conducting ingenious combination of these short sequences, to construct a basic operation set (Add, Load, Store, etc.) to achieve the capability of Turning-complete compute. By hijacking normal program control flow, ROP executes ret-ended malicious instruction sequences. Hence it can conduct any purpose of attack. Later, ROP variant appears---JOP [5, 6]. JOP controls the program

flow with indirect jump instructions, rather than ret instructions. However, common features are shared by ROP and JOP which both make use of short sequences of fragments from libraries to execute malicious program flow.

B. ROP Attacks on ARM

At present, ROP and its variant have been mounted on the ARM platform and their generic structure shows in Figure 1.Firstly, the legal program control flow is hijacked by multiple measures, such as stack overflow, string format attack [13]. Secondly, ROP attack flow runs into instruction sequences (ending with BX LR) which are pointed by the address information in the Control Structure (CS), one sequence by another, to execute malicious tasks. Usually, there are stack manipulation instructions in front of the BX LR instructions to control the execution flow according to the CS, such as LDMFD SP!,{R4,LR} which loads the data from the hijacked stack pointed by SP and SP-4 into register R4 and register LR. In Figure 1, serial numbers 1-5 illustrate the ROP control flow on ARM.

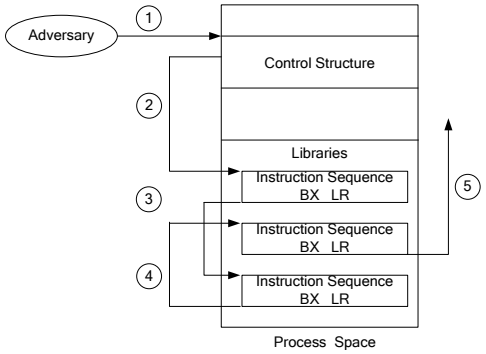

Figure 1. ROP on ARM

In theory, any instruction which can transfer program execution flow can be used to achieve ROP attack as long as there are sufficient library codes for build of ROP gadgets according to Hovav Shacham et al [2]. So, indirect jump instruction is a good choice. On ARM platform, typical JOP attack shows in figure 2. In this figure, the program flow is controlled by indirect jump instruction BLX Rx [10] (Rx means one of the suitable registers from R0-R15 according to the procedure of gadget search and creation). As ROP attack, the JOP attack first hijacks legal program control flow, and then executes short instruction sequences one by one. The only difference is, when each short sequence finishes, its control flow runs into the trampoline area which combines all sequences according to the address information in the Control Structure. The trampoline updates R15 (PC), loads address of the next gadget and transfers the program flow to the next gadget according to the data in CS. Generally, the trampoline consists of two or three instructions from library code fragments. The serial numbers 1-3 in figure 2 illustrate the control flow of JOP on ARM.

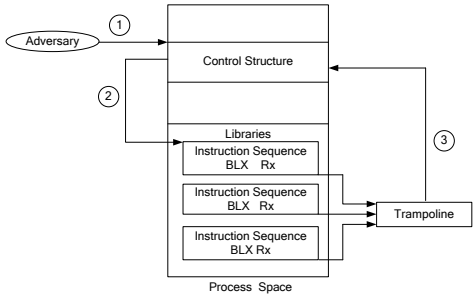

Figure 2. JOP on ARM

C. Dynamic Binary Analysis and Instrumentation

Dynamic binary analysis [14] can extract program characteristics which only show on run-time. As we know, many various static characteristics of viruses are the same as legal programs. It is difficult to distinguish these viruses from legitimate programs just according to static characteristics. Under this circumstance, if we plug in profile code into the program to obtain dynamic real-time and statistical data, we can identify these viruses from legal programs. Dynamic Binary Instrumentation (DBI) is such a technology. It can monitor dynamic program behavior during run-time without changing the source code or even without the source code. It doesn't change program behaviors. By inserting analytical codes into binary codes on run-time, DBI runs these codes on its virtual machine to obtain dynamic behaviors of programs, such as registers, memory, and status of instruction sequences.

At present, there are several good DBI frameworks and tools, such as Pin [15, 16] and Valgrind [17, 18]. In our experiment, we use Valgrind which now supports embedded ARM-Linux platform and ARM-based Android to implement our algorithm. By means of disassemble-and-resynthesize[18](D&R) technology, Valgrind disassembles the binary code into an format of intermediate code and combines the instrument code. Then it converts the intermediate code back to the machine binary code again. Therefore, the machine can run the code to provide dynamic running information of the program and judge whether the control flow of the program complies with rules of legal programs.

III. ALGORITHM AND SOLUTION ON ARM

A. Characteristics of ROP Attack on ARM

Research on ROP shows the following characteristics of instruction sequences of ROP attack: 1. The data stored in the Control Structure are addresses of instructions and program intermediate variables, not instructions themselves; 2. Short instruction sequences maliciously used by ROP are short, usually with a length of about 2-5. The automatic search algorithm for ROP gadget in Tim Kornau et al [9] proves this point. For the following reasons: a short length of instruction sequences can facilitate automatic gadget search,

expand the scope of candidate libraries and reduce the possible side effects of the ROP exploit codes and register dependencies; 3. ROP attack makes use of function epilogues as much as possible, while avoiding function prologues. For the function prologue conducts allocation of stack frames, parameter initialization and other operations that change the stack structure, which may bring more side effects and control flow damage to ROP itself that cannot be recovered easily. Therefore, from the perspective of ROP, there is no concept of function but binary code fragments. Its control flow will inevitably enter into the function from the middle position of function to avoid function prologue and go directly to the epilogue, then transfer to subsequent short instruction sequences. Figure 3 shows control flows of legal programs and ROP attack code.

Thus, the reason why ROP attack becomes feasible is that the malicious use of the function code in the library and the damage to the function call specification. The majority of libraries are provided with the form of function which the caller must comply with the function call specification. However, there are no constraints of this specification from the computing devices when the user makes use of these libraries on run-time, which makes it possible to be exploited by ROP attack. So if we enforce the function call specification to avoid malicious use of libraries, the Turning-complete feature of ROP will lose and its feasibility will disappear.

The Application Binary Interface (ABI) specification of ARM, AAPCS [19] standardizes the use of binary code and specifies principles of the function call on ARM. It provides rules of stack operation and parameter transfer. If users abide by this specification and the computing devices enforce this specification, ROP attack control flow will be substantially constrained. If this specification can be enforced on the ARM platform, it will reduce the feasibility of ROP attack on ARM to a great extent.

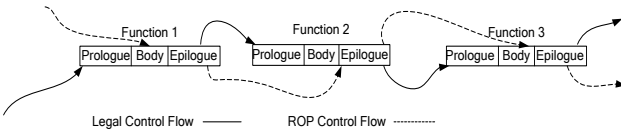

Figure 3. Legal Control Flow *VS* ROP Control Flow

B. Detection Algorithm

It is difficult to detect above dynamic characteristics that only show on run-time. As mentioned in Section 2, DBI tools can provide run-time status of program, such as instruction address, register values, memory access and locations of shared library images without changing the original behaviors and results of the program. Therefore, we enforce this specification with the help of DBI framework. Our detective algorithm against ROP shows in Figure 4.

In ARM specification, function call is implemented by BL or BLX instruction while indirect jump uses B or BX instruction. But these instructions are probably misused, for B or BX instruction can also be used to conduct a function call as long as the parameters are set to R0-R3 registers. There is no fixed return instruction. Any instruction that changes R15 (PC) can be used as a return instruction. Therefore we cannot identify the return instruction like Intel x86 to indicate whether the function returns. In our algorithm, we consider the transfer of control flow within a function is legal and the transfer of control flow across multiple functions must comply with the function call specification defined by the AAPCS. If the program uses a function, its control flow must enter into the function body from the starting position of its prologue. When the program calls a function, we record the next instruction address (PC+4 for ARM, PC+2 for Thumb) after the call instruction into a shadow stack and conduct the jump operation. Then, enforce the control flow to enter into callee function from the starting position of its prologue and we compare the target address of the call with the start address of the prologue of the function to be called to enforce this rule. Only the operation returning back to the position after the call instruction is permissible. When the control flow goes back to the caller's call position, we compare the position's address with the top value of the shadow stack to check this rule. Any other control flow directly entering into the middle position of a function is illegal. Here we relax the restrictions that we do not distinguish B, BL, BX or BLX instruction to perform a function call. Finally, if the control flow doesn't abide by our rule defined above, we judge it as a ROP malicious code. Beyond all question, ROP and JOP will not follow this rule, so we can detect them easily.

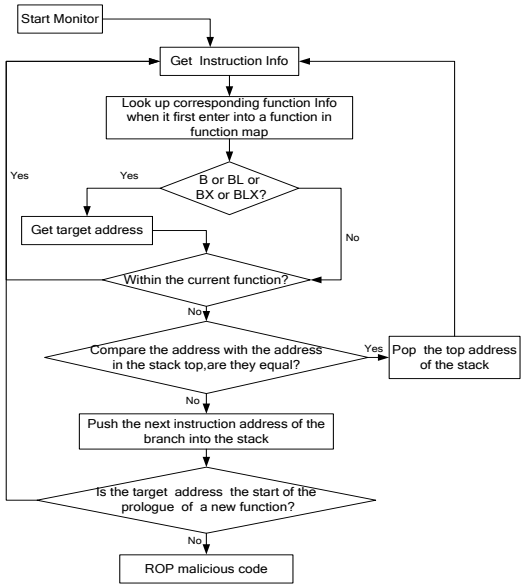

Figure 4. ROP Detection Algorithm

C. Detection Architecture based on DBI

Our detection algorithm is based on DBI, its architecture showing in Figure 5. Firstly we disassemble the shared libraries on ARM to extract function list to obtain each function's offset to the start of the library, its body size. Then we make use of the API provided by DBI framework to obtain the locations of the shared libraries' images in the memory when the program starts under the monitor of DBI tool. After these steps, we store these functions' information in a map with each function's start address as the key which provides comparison data for Ropgrind. Then we use Valgrind to monitor the ARM code to be examined under our Ropgrind tool which implements the detection algorithm. With this detective monitor, we can detect ROP and JOP on run-time.

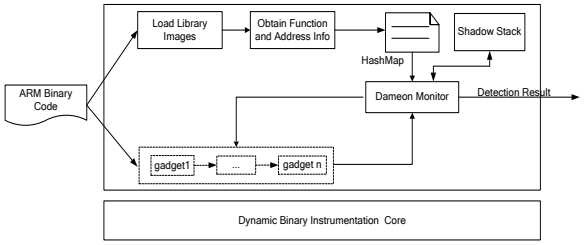

Figure 5. Ropgrind Architecture

IV. IMPLEMENTATION

We implement our detection algorithm against ROP on ARM Linux platform. In this experiment, we use qemu[20] simulator to simulate the ARM platform. The configurations are ARM V7, the Cortex-A9 architecture, with Linux 2.6.38. The cross compiler is arm-linux-gcc[21]. We make use of Valgrind 3.7.0 which now supports ARM-Linux and ARM-Android 2.3.x to provide dynamic status of the program to be examined. We base our Valgrind tool Ropgrind (rg) on Valgrind core.

Firstly, we disassemble the frequently used dynamic link library on ARM Linux platform. In our preliminary version of Ropgrind, we disassemble the libc library because of many programs utilize this library and it is often exploited by ROP attacks as a code base. To be emphasized, our solution can protect any shared library on ARM-Linux platform. Then we obtain all the functions' offsets to the libc start position, their sizes of functions' body and store them in a map for the monitor to search. The disassemble process can utilize readelf [22], objdump[23] and other disassembly tools. And then we use Valgrind to start our Ropgrind to monitor the execution of the program to be examined. Valgrind core provides information of instruction type, instruction address and locations of the shared library image in the process space for daemon monitor which implement our detection algorithm. During the run-time of the program to be examined, our Ropgrind will monitor whether its control flow abides by the rule defined in Section 3. If it violates the rule, Ropgrind will recognize it as a ROP attack and terminate it.

V. EVALUATION

In this section, we will give some experimental data of Ropgrind. We make use of methods raised by Tim Kornau et al [9], Lucas Davi et al [10] and Blackhat et al[24] to create ROP and JOP virus samples to test Ropgrind. These virus samples include arithmetic operations, system calls, logic control operations etc. Ropgrind detects these virus samples accurately. At the same time, we use C to conduct legal function calls of functions in the libc and achieve the same operations as above virus programs do. Ropgrind also judges these C programs as legal programs and makes correct judgments. Furthermore, we test some commonly used tools in Linux under Ropgrind and they all pass our detections. We will show our experimental data in the following parts.

A. Correctness of the Detection

Our experimental data shows in Table 1. It shows that Ropgrind can work effectively on detecting ROP and JOP exploits on ARM-Linux platform. It can correctly distinguish ROP and JOP exploits from legal programs. Its false positive and false negative values are in a very low level. Our experiments on large legal program samples show that it can accurately distinguish between legitimate programs and ROP exploits.

TABLE I. ROPGRIND DETECTION DATA

Program	*gadget*	*description*	*exploit type*	*Detected*
bin-shell-rop	15	execute /bin/sh	rop	√
bin-shell-jop	23	execute /bin/sh	jop	√
bsh-normal	-	execute /bin/sh	no	pass
/bin/sh	-	execute /bin/sh	no	pass
add	5	add two integers	rop	√
add	7	add two integers	jop	√
dup	18	copy and paste a file	jop	√
gcc	-	compile a C file	no	pass
cp	-	copy a file	no	pass
gzip	-	compress a file	no	pass
ls	-	list all files of the dir	no	pass
cat	-	cat multiple text files	no	pass
kill	-	kill a process	no	pass
grep	-	search a word in files	no	pass

B. Performance

We test several common Linux tools to evaluate the performance of Ropgrind. Its performance data shows in Figure 6. We run these tools under three kinds of circumstances: run these tools themselves, under Nulgrind and under Ropgrind. Nulgrind is taken by Valgrind which contains empty analysis code, and thus its performance overhead is almost Valgrind core's overhead.

In Figure 6, we show running time of several Linux tools under Nulgrind and Ropgrind. We illustrate 11 groups of time data. The blue columns of each group present the running time of the tool; the maroon columns of each group present the running time of the tool under monitoring of Nulgrind and the cyan columns presents the running time

under Ropgrind. The units are seconds. To be emphasized, only relative comparison of execution time is meaningful.

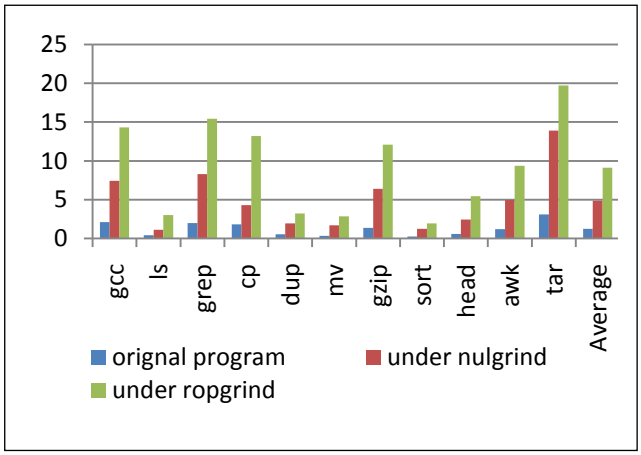

Figure 6. Ropgrind Performance

The rightmost group of column data is the mean values of the former 11 groups of column data. The data in Figure 6 show that the average overhead of Nulgrind is 5X compared to the original program. The overhead of Ropgrind is about 9X. The main reasons are as follows: 1. Valgrind is a heavyweight dynamic binary instrumentation framework and tool, and its original overhead is about 4X (Valgrind [18]). 2. Our detection algorithm is relatively complex and conservative, so it will consume a certain amount of time. 3. The embedded system itself owns a low speed processor and less memory space, so it also will increase the overhead of our detection tool.

C. Improvements

The experimental data show that the correctness of our algorithm is guaranteed. Several samples, including both legal programs and ROP exploits, are checked by the Ropgrind and it conducts the work accurately. By far, the values of its false negative and false positive computed by us are low. It can protect the embedded system ARM from ROP and JOP attack which can circumvent most of current protective mechanisms on ARM. However, its accurate values of false negative and false positive need a large number of tests and ROP samples; and we are conducting this work.

Ropgrind is a fundamental version of our detection tool against ROP and JOP on ARM. However, the overhead of Ropgrind is relatively high. We will conduct some optimizations and improvements for the detection algorithm and Ropgrind. Our improvements go toward the following directions: 1. Make use of static analysis tool to disassemble the shared libraries to remove some invalid entities in the map to improve the performance of search; 2. Add cache to record some intermediate results to eliminate some overhead; 3. Explore and discover lightweight DBI frameworks and tools [25] on ARM to address the high overhead of Valgrind; 4. Design other brief detection algorithms to combine with the current detection algorithm to incrementally detect ROP and JOP on run-time in advance, such as dynamic probability statistically method, to dynamically select the detection method against ROP and JOP.

Our solution is a sandbox-like monitor to detect the ROP viruses. So it has defects of high overhead that most of the sandboxes and virtual machines confront with. However, we use Ropgrind to detect a program when it is firstly used. If it is proved to be legal, we don't need to run it under Ropgrind all the time.

VI. Related Work

There are several detection methods against ROP on PC platforms: 1. Control-flow-based detection method: by the control flow integrity checks to detect ROP attacks [26, 27, and 28], however, as the consequence of complexity and high overhead, it has never been fully implemented. 2. Address randomization [29, 30]: by means of randomizing the load locations of shared libraries to increase the difficulty of address search for creation of gadgets of ROP to damage its control flow. Nowadays, Linux PAX [35] project has supported this randomization. But ARM doesn't have support for this feature. 3. Compile-based defense method [31, 32], that is, by rewriting the compiler to maximize the elimination of susceptible instructions that may be maliciously used by ROP, such as ret instructions. However, this method is difficult to deal with legacy systems and for ARM, it is not suitable to implement because the Instruction Set of ARM is more flexible and difficult to eliminate. 4. Detection Method based on DBI [33, 34]: A. Through Call-Ret instruction pair integrity check, this method can prevent traditional ROP in Intel x86 platforms, but do nothing with JOP and other variants. On ARM, it doesn't work, because ARM has no fixed ret instruction, and it still can't tackle JOP on ARM. B. Through analysis of statistical data to detect ROP, however, we don't think it is useful for small number of samples for ROP and we don't think current simple statistical regularity can make any sense on ARM. Better statistical regularity needs to explore and discover for ARM.

VII. Conclusion and Future Work

In this paper, we design a dynamic detection algorithm against ROP and JOP on ARM platform. Based on dynamic binary instrumentation technology, our algorithm is to check the ROP malicious control flow according to the features of the ARM platform. As best as we know, our method is the first one to detect ROP on ARM platform. We develop a Valgrind tool Ropgrind to implement our algorithm. Our experiment shows that Ropgrind can accurately detect ROP virus and its variants. By increasing the difficulty of ROP creation and reducing the misuse of shared libraries, ROP viruses will be difficult to achieve the feature of Turning-complete.

Next step, we will continue to improve the performance of Ropgrind on ARM. Introduction of probabilistic statistical analysis technology will help to achieve early identification of ROP control flow as soon as possible. At the same time, we will reduce the scope of shared libraries to be

monitored by Ropgrind to protect the core assembly code and improve the performance of detection on ARM, for example, the libc system call interface is important for many applications, if Ropgrind only limits these critical codes, many ROP attacks will disappear.

Introduction of other defensive technologies on PC to ARM will also contribute to system security. Address randomization can increase the difficulty of building ROP gadgets. Compiled-based solutions can make new codes in the future less maliciously used by ROP exploits. The ideas of control flow and hardware support can be simplified to achieve ROP attack prevention to improve the safety of our programs.

Meanwhile, we think that the performance overhead of our solution is an essential issue which is confronted by all the solutions on the basis of software level. A hardware level support of program profiling will remove the overhead and may work better. At the same time, on kernel level, some validation measures of addresses and memory accesses may be achieved to detect and defend ROP viruses. Our future research will also go toward these two directions.

Furthermore, we will improve our detection algorithm to adapt to Android. Currently, performance improvement is a critical issue for Android and we will conduct continuous improvements.

ACKNOWLEDGES

We thank JianJian Song,Wen Chen and other members of our lab for providing significant advices for our work. We also thank Li'Ang for providing support for the experiment of our work. This work was supported by the Chinese National Natural Science Foundation (NSFC 61073027, NSFC 61105069) and State Key Laboratory for Novel Software Technology of Nanjing University.

REFERENCES

[1] Weiliang Du. Return-to-libc Attack Lab[J].2007. http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Return_to_libc/Return_to_libc.pdf.

[2] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86): CCS'07 Proceedings of the 14th ACM conference on Computer and communications security, 2007[C]. New York, NY, USA: ACM, 2007:552-561.

[3] Ralf Hund, Thorsten Holz, Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms: SSYM'09 Proceedings of the 18th conference on USENIX security symposium, 2009[C]. CA, USA: USENIX Association Berkeley, 2009: 383-398.

[4] Erik Buchanan, Ryan Roemer, Hovav Shacham. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC: CCS'08 Proceedings of the 15th ACM conference on Computer and communications security, 2008[C]. New York, NY, USA: ACM , 2008: 27-38.

[5] Stephen Checkowayy, Lucas Daviz, Alexandra Dmitrienkoz, etc. Return-Oriented Programming without Returns. CCS '10 Proceedings of the 17th ACM conference on Computer and communications security, 2010[C]. New York, NY, USA: ACM, 2010: 559-572.

[6] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack.ASIACCS '11, March 22–24, 2011, Hong Kong, China. ACM.

[7] Ping Chen, Xiao Xing, Bing Mao, Li Xie, Xiaobin Shen, Xinchun Yin. Automatic Construction of Jump-Oriented Programming Shellcode (on the x86).ASIACCS '11, March 22–24, 2011, Hong Kong, China. ACM.

[8] Felix Lidner. Developments in Cisco IOS forensics. Router Exploitation. http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf.

[9] Tim Kornau. Return Oriented Programming for the ARM Architecture[C], http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf, Master thesis, Ruhr-University Bochum, Germany. 2009.

[10] Lucas Davi, Alexandra mitrienkoy, Ahmad-Reza Sadeghi etc. Return-Oriented Programming without Returns on ARM: In Technical Report HGI-TR-2010-002,2010[C]. Ruhr University Bochum, Germany ,2010.

[11] Stephen Checkoway, Hovav Shacham, Escape From Return-Oriented Programming: Return-oriented Programming without Returns (on the x86). Techinal Report CS2010-0954,US San Diego, February 2010.

[12] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Marcel Winandy. Privilege Escalation Attacks on Android. System Security Lab. Ruhr-University Bochum,Germany.

[13] Scut/Team Tesco, Exploiting Format String Vulnerabilities. Sep 1,2001.

[14] Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. Trinity College. 2004. http://valgrind.org/docs/phd2004.pdf.

[15] University of Virginia ,Pin, http://www.pintool.org/.

[16] Chi-Keung Luk, Robert Cohn, Robert Muth etc. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation: PLDI '05 Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation,2005[C]. New York, NY, USA ACM, 2005: 190 - 200.

[17] Valgrind Developers: Valgrind (2000–2005), Web page at http://www.valgrind.org/.

[18] Nicholas Nethercote, Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation: PLDI '07 Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation,2007[C]. New York, NY, USA:ACM, 2007: 89-100.

[19] AAPCS. ARM Procedure Call Standard for the ARM Architecture. ARM IHI 0042D,release 2.08, October,2009. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf.

[20] QEMU. http://wiki.qemu.org/Download.

[21] http://code.google.com/p/mini6410-debian/downloads/detail?name=arm-linux-gcc-4.5.1-v6-vfp-20101103.tar.gz&can=2&q=.

[22] readelf. http://www.sourceware.org/binutils/docs/binutils/readelf.html.

[23] objdump.http://sourceware.org/binutils/docs/binutils/obj--dump.html.

[24] Blackhat.ARM EXPLOITATION ROPMAP. https://media.blackhat.com/bh-us-11/Le/BH_US_11_Le_ARM_Exploitation_ROPmap_Slides.pdf.

[25] Ryan W.Moore, Jose A.Baiocchi. Addressing the challenges of DBT for the ARM Architecture. LCTES'09, June 19-20,2009,Dublin ,Ireland, ACM.

[26] Martin Abadi,Mihai Budiu,Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications[J]: ACM Transactions on Information and System Security (TISSEC). 2009,Volume 13 Issue 1: No. 4 article.

[27] Vladimir Kiriansky, Derek Bruening, Saman Amarasinghe. Secure Execution Via Program Shepherding, the Proceedings of the 11th USENIX Security Symposium (Security '02), 2002[C], San Francisco, California, 2002: 191-206.

[28] Crispin Cowan,Perry Wagle,Calton Pu,Steve Beattie and Jonathan. Buffer Overflows: Attacks and Defenses for the Vulnerability of the

Decade[J]: DARPA Information Survivability Conference & Exposition, 2000, Volume 2: 1119.

[29] Hovav Shacham, Matthew Page,Ben Pfaff. On the Effectiveness of Address Space Randomization:CCS '04 Proceedings of the 11th ACM conference on Computer and communications security, 2004[C]. New York, NY, USA: ACM, 2004:298-307.

[30] Hirsto Bojinov,Dan Boneh,Rich Cannings. Address Space Randomization for Mobile Devices. WiSec'11, June 14-17,2011,Hamburg,Germany.ACM.

[31] Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace etc. Defeating Return-Oriented Rootkits With "Return-less" Kernels: EuroSys '10 Proceedings of the 5th European conference on Computer systems,2010[C]. New York, NY, USA ACM,2010: 195-208.

[32] Kaan Onarlioglu,Leyla Bilge,Andrea Lanzi,Davide Balzarotti etc. G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries: ACSAC '10 Proceedings of the 26th Annual Computer Security Applications Conference,2010[C]. New York, NY, USA ACM,2010:49-58.

[33] Lucas Daviy, Ahmad-Reza Sadeghiy, Marcel Winandyz. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks: ASIACCS '11 Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security,2011[C]. New York, NY, USA ACM,2011:40-51.

[34] Ping Chen, Hai Xiao, Xiaobin Shen etc.DROP: Detecting Return-Oriented Programming Malicious Code. In A. Prakash and I. Gupta, editors, Fifth International Conference on Information Systems Security (ICISS 2010), volume 5905 of Lecture Notes in Computer Science, Springer , 2009:163—177.

[35] PaX Team. Documentation for the PaX project. See Homepage of The PaX Team. 2003. http://pax.grsecurity.net/docs/index.html.

[36] William Enck,Damien Octeau,Patrick McDaniel and Swarat Chaudhuri. A Study of Android Application Security. http://www.enck.org/pubs/enck-sec11.pdf

[37] Francesco Di Cerbo,Andrea Giradello. Detection of Malicious Applications on Android OS. IWCF 2010. Springer 2011.