Tamper-Resistant Code Using Optimal ROP Gadgets for IoT Devices

Chittaranjan Hota, Rajesh Kumar Shrivastava, Shivangi Shipra
Department of Computer Science & Engineering
BITS Pilani, Hyderabad campus, Hyderabad, India
{hota,p2015005,f2012154}@hyderabad.bits-pilani.ac.in

***Abstract*—IoT is an emerging area with revolutionary applications in areas like transportation, health care, energy, and agriculture etc. It can be realized through the massive deployment of IoT devices in the physical world. But massive deployment comes with severe security concerns. IoT devices are capable of running operating systems within themselves. An adversary may tamper or clone these IoT devices for his or her benefit. Most of the modern operating systems today use attack prevention techniques based on No-eXecute (NX) memory page protection scheme, which prevents execution of malicious code injected into the device. But, Return Oriented Programming (ROP) has the capability to break this security cover and take control over the IoT device. In this paper we present an effective implementation of the ROP, used as a defensive weapon against code tampering by adversaries. We also propose techniques to choose the best gadget set for building the ROP chain.**

Keywords:- Internet of Things, ROP, Gadget, Genetic Algorithm.

I. Introduction

Internet of Things (IoT) is an inter-connected network of devices like a tablet, a smart phone, a fitness device, a light bulb, a football helmet, a temperature sensor, an energy meter, a door lock, and an actuator etc. that enable these objects to collect and exchange data. IoT offers both telescopic and microscopic insights into the once invisible world between people, machines, and physical objects. By tagging physical objects or putting sensors around these objects, with Internet connectivity currently it is being possible to not only track these objects and collect new types of data but also combine these data to generate a greater level of information. As far more sophisticated sensors, micro chips and data analytics capabilities emerge, there is a growing ability to observe environments and understand complex relationships. Fig.1 shows an overview of IoT architecture which mainly consists of IoT node layer, Gateway layer, and Server layer.

IoT architecture shows in Fig.1 comprises of clients as IoT nodes, gateways connecting several IoT nodes and an application server. The IoT node layer collects data from various sources like temperature sensors, light sensors etc. using various wireless communication protocols like Z-wave, ZigBee, WiFi etc. The gateway layer processes the incoming readings and filters them into meaningful(relevant) data. The server layer collects data from the gateway, analyses it and runs smart applications with insights from this analysis. Shielding these devices (node, gateway, server)

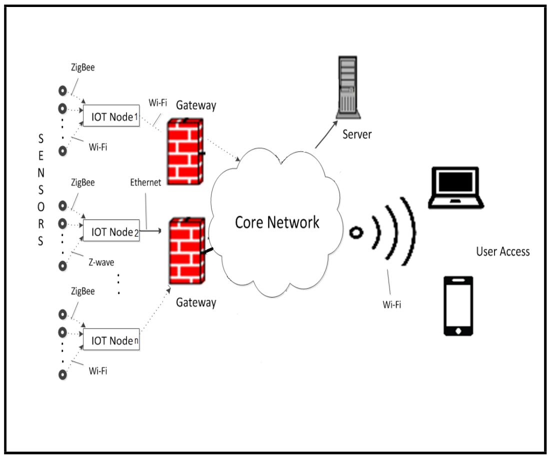

Fig. 1: An Overview of IoT Test-bed

from unauthorized modifications or code tampering is of paramount significance to their unobstructed operations. A novel technique known as Return Oriented Programming (ROP) can be used as a defensive technique to protect the code from being tampered. ROP uses memory vulnerability exploits to transfer payloads into the stack and gain control over the execution sequences.

To prevent tampering of execution sequences, various researchers have proposed numerous tamper-proofing [7] [21] and obfuscation schemes [12]. However, Wurster et al. [22] proved that these technique for tamper proofing are breakable using reverse engineering techniques. Further researchers have developed Control Flow Integrity (CFI) techniques [1] [21] to overcome the problems of previous techniques in tamper proofing the code. CFI works well if processor takes code and data from a single cache. But in modern processors like Intel x86, AMD64 etc, L1 cache is divided into two part L1(I) cache for code segment and L1(D) cache for data segment. Since processors use an operation like W$\oplus$X [22] for protecting code tampering in the instruction cache, methods like code obfuscation [20], CFI [1], and check-summing [7] etc. work only on instruction cache. This leaves the data cache vulnerable. Research by Shacham et al. [18] shows that adversaries may take advantage of data cache vulnerability to tamper the code using techniques like ROP.

In this work of ours, we used ROP chains [18] consisting of short instruction sequences called as gadgets, for example, pop eax, ret; pop ebx, ret; etc, to protect the code from being tampered. A gadget is a short instruction sequence ending with return (ret instruction) like pop eax, ret. A ROP chain consists of several such gadgets executed in some order. Mainly an adversary can push a ROP chain into the code using memory vulnerability like buffer overflow and gain access to the victim's machine or device. In this work we propose usage of this technique (ROP chain) to make the code tampering difficult. This is further enhanced by creating an optimal ROP chain that improves the execution time of the code running on the device. Specifically, we make the following contributions:

- We protect data cache by using ROP chain exploiting the memory vulnerability exploit, hence providing resistance to attacks outlined in Wurster et al.[22].
- We use Genetic Algorithm (GA) to find an optimal number of gadgets to control normal execution sequence of the code. This makes the life of reverse engineer difficult to understand the control flow and tamper the code.
- Gadgets are chosen from an executable binary that makes the system dependent, disallowing cloning of the code.
- To validate aforesaid claims, a prototype model of our technique is deployed on the IoT test bed at our campus.

II. Related Work

Most of the code tamper-proofing algorithms make use of check-summing [7], code obfuscations [5], [16] or inspecting control flow [1]. Only Andriesse et al. [2] created ROP chains for protecting the code . But Andriesse et al. [2] never experimented with IoT devices. Since an IoT device has a limited computation power and less number of gadgets are available within the code running on IoT devices, selecting an appropriate number of gadgets in such scenarios is a main challenge. To make the brute force attack or reverse engineering attack difficult, we randomly selected gadgets to build an optimal chain.

Control Flow Integrity (CFI) is another alternative to prevent subvert of machine code from exploitation. But it is a static method and is not able to protect code from ROP based attacks [2]. Modern operating systems use other techniques like Address Space Layout Randomization (ASLR), usage of NX bit, usage of stack canaries etc. to protect code integrity. Andrea et al. [4] showed that in the presence of these defence mechanisms, ROP is able to alter the normal control flow of the program.

Introduction of ROP [8] and it's subsequent advancements [13], [11], [10], [9] led to its adoption in carrying out real-world attacks [14], [3]. To overcome the limitations of ASLR, ROP exploits are used in all modern operating systems [15], [10], which otherwise would prevent or at least hinder [19] these attacks. As far as authors know, other than Andriesse et al. [2] no other researcher has used ROP as a defensive mechanism. We also explore possibility of strengthening security strength of the IoT code by devising optimal ROP chain drawn from random set of gadgets to tamper proof the code.

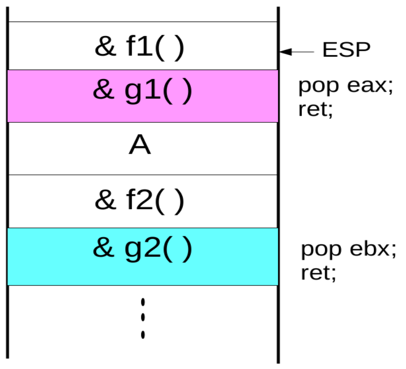

Fig. 2: Example ROP chain with Gadgets

III. Background

A. Return-Oriented Programming

ROP was originally developed as an exploitation technique to bypass the computer security measures such as Data Execution Prevention (DEP), ASLR and allow malicious code execution. This technique uses the memory corruption vulnerability to gain control of the call stack and then executes carefully chosen short return terminated machine instruction sequences called gadgets. All gadgets use small instruction sequences ending with ret or $c3$ [18]. ROP is a technique in which control of execution can be hijacked. To implement ROP no prior knowledge of the structure of the source code or of the binary is required [4]. ROP chain is a collection of these gadgets such that the return instructions terminating each gadget transfers control to the next gadget [18]. This allows a hijacker to take control of the execution without necessarily injecting any malicious code [6].

Fig.2 illustrates an example of ROP chain, initially the stack pointer (esp) points to the address of function f1(). Upon execution of return instruction in f1(), control is transferred to gadget g1 in the chain. This pops stack top (value A) into eax register and the ret of g1 increments esp to function f2(). Then when f2() returns, g2 is executed, and so on. The ret instruction in gadget g1 helps in the transfer of control to gadget g2 that helps to perform next instruction fetched from memory. Gadget g2 now returns to gadget g3 and the process continues until all gadgets g1, . . . , gn have been executed.

B. Threat Model

We consider a hostile host threat model similar to Andriesse et al. [2] and Wurster et al. [22] which is considered to be a standard threat model for tamper-proofing techniques.

In our threat model we assumed that the tamper proofed application runs on an IoT node controlled by a hostile user, who has entire control over the run time environment and can modify this tamper-proof code. The hostile user can alter the code during runtime debugging and also can add small pieces of code to tamper the functioning of IoT nodes. The hostile user may also be interested to bypass the access control checks like anti-debugging or license verification to gain control over the device. To circumvent the hostile user from carrying out these tasks our goal is to devise an efficient tamper-proofing technique using ROP chains to maximize the time and effort required by the hostile user to make these changes in the worst case possible. As the tamper-proof code runs on resource constrained IoT nodes, we also aim to improve the execution time of the tamper-proof code by not augmenting it with a longer ROP chain, rather with an optimal ROP chain.

IV. Overview of the System

A. Algorithms

Algorithm 1 Algorithm for Minimal Gadget Set Creation

```
Input: G: set of gadgets [g1, g2, ..., gn]
Output: list of gadgets based on least avg. exec. time
 1: if (G.length > 0) then
 2:     $i = 0$;
 3:     $j = 0$;
 4:     $T[i] = 0$; // Execution time of ROP chain
 5:     Generate ROP chain with G [i];
 6:     while j < 100 do
 7:         $T[i] += time(ROPchain)$;
 8:     end while
 9:     $avg = T[i]/100$;
10:     $i = i + 1$;
11: end if
12: Sort (G[i] based on T[i] index);
13: return List of best possible gadgets for building ROP
    chains.
```

We used Ropgadget [17] compiler to find out a set of Turing complete gadgets for building a ROP chain. We evaluated execution time of these chains where each chain is built using one gadget. Algorithm 1 shows our approach where we averaged out execution time of each ROP chain over 100 instances to avoid any biases. Implementation of algorithm 1 using Python script resulted in 8 ROP chains (shown in Table 1) with their average execution times. These 8 ROP chains where used as initial population in the Genetic Algorithm(GA) explained next. Other ROP chains like inc eax, pop esp, ret; etc. generated by Ropgadget compiler [17] were not considered because of their larger execution times.

TABLE I: Gadget list with their avg. execution time

ROP chain	Gadget	Avg. exec. time
1	Mov dword ptr[esi],edi;pop ebx, pop esi; pop edi; ret	0.00013982
2	Pop esi; ret	0.00012074
3	Pop edi; ret	0.00012497
4	Pop eax; ret	0.00013152
5	Pop ebx; ret	0.00011766
6	Pop ecx; ret	0.00013212
7	Pop edx; ret	0.00011583
8	Pop eax; ret; Mov dword ptr[edx], eax; ret	0.00013207

B. Genetic Algorithm to choose an optimal chain

A Genetic algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a set of stochastic operators.

Algorithm 2 Genetic Algorithm to find optimal ROP chain

```
Input: G : set of gadgets [g1,g2..., g8]
Output: Optimal ROP chain
 1: Set time t = 0;
 2: Initialize population set, $g[\,] \leftarrow g1, g2, ...g8$;
 3: while time (ROPchain) $\leq$ time (Original code) do
 4:     Evaluate fitness function(time) of each member of G;
 5:     Select new parent for reproduction based of fitness;
 6:     Produce the offsprings of these parents using genetic
        operators;
 7:     Mutate individuals, based on fitness function;
 8:     Set time t = t+1;
 9: end while
10: return Optimal ROPchain.
```

In Algorithm 2, we use 8 gadgets (output from Algorithm 1) to frame a binary chromosome as below:
$Chromosome = (g1\ g2\ g3\ g4\ g5\ g6\ g7\ g8\ )$

After initializing the population set we evaluate the fitness of these individuals using a fitness function. The fitness function computes execution time of an individual. The selection operator generates a new population of individuals from the current population probabilistically according to individual fitness. The crossover function used in Algorithm 2 forms offsprings by combining the part of genetic information from their parents. We used GA package of language R to implement Algorithm 2. An example of crossover operator is shown in Fig.3. Two parents are represented by two 8-bit strings which are fit to participate in the crossover. Both the offsprings inherit genes like 11 and 00 from parent1 and parent2 respectively. Other genes are inherited randomly from either of them. The mutation operator used in Algorithm 2 randomly alters the values of some genes in the new population as shown in Fig.3. The GA function in Algorithm 2 produces new population from the existing ones iteratively till the stopping condition is met, i.e, execution

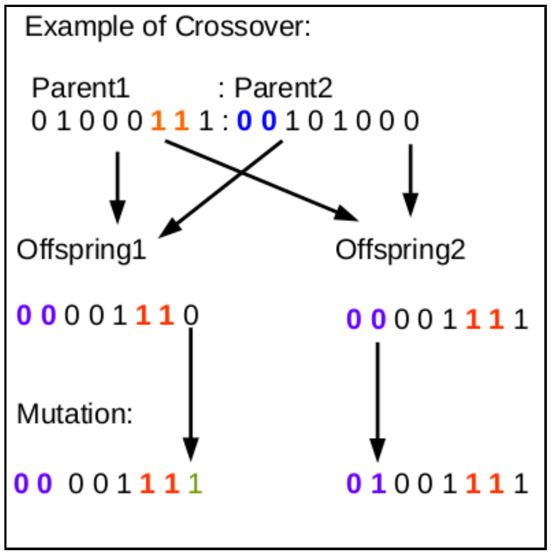

Fig. 3: Crossover example of two bit strings of length 8

time of ROP chain produced by GA is less than equal to execution time of original code without ROP chain.

V. Experimental Results

In this section, we evaluate the performance of our prototype implementation of optimal ROP chain based code integrity verification. We compared our GA based solution with standard ROP augmented code.

A. Experimental setup

We used the prototype IoT test-bed developed in our campus as shown in Fig.1 running smart energy application with Intel Galileo gen2, Arduino, Raspberry pi boards as IoT nodes and gateways. The C programs running on these IoT devices were augmented with ROP gadgets to form chains as described in Table 1. We found out best possible gadgets through different ROP chains by considering their minimal average execution time as shown in Table 1.

B. Optimal ROP chain

The GA works by mimicking the Darwin theory of evolution. In our problem each gadget is a gene. Chromosomes are modeled as sequences of 0 and 1, 0 indicating presence of that gene or gadget and 1 indicating otherwise. A set of initial solution is created based on the fitness value. This set of solution is called population. The chromosomes with best fitness value are combined (crossover) to create offsprings with mutation to create a new population. The process is repeated to get better and better solutions.

The fitness function here checks which bit from the 8 bit string is set. If the i^th bit is set, it implies that the i^th gadget is selected for the ROP chain. After evaluating the string, the ROP chain is created using those gadgets. The Python script calls the 'C' executable with ROP chain and measures

TABLE II: The gadgets, the number of times that gadget was selected by GA in 109 solutions

Gadgets	Numbers of times it occurs in the optimal solution	%of time it occurs in the optimal solution
Mov dword ptr[esi],edi;pop ebx, pop esi; pop edi; ret	51	0.467889908257
Pop esi; ret	48	0.440366972477
Pop edi; ret	47	0.43119266055
Pop eax; ret	55	0.504587155963
Pop ebx; ret	47	0.43119266055
Pop ecx; ret	58	0.532110091743
Pop edx; ret	56	0.51376146789
Pop eax; ret; Mov dword ptr[edx], eax; ret	52	0.477064220183

the execution time taken. Since we try to minimize the time taken, so we maximize 1/time function.

The GA is run for 100 times and the number of times a gadget occurred in the optimal solution is counted. The Table 2 shows the gadgets, number of times it occurred in the optimal solution and the percentage of the same. On 100 runs, we got 109 optimal solutions. Experiments show that gadgets like pop eax; ret, pop ecx; ret, and pop edx; ret are the most preferred gadgets because these gadgets occurred more number of times in the optimal solutions.

Fig.4 shows run time performance evaluation of different ROP chains of length 3. It shows all permutations of the above 8 gadgets of 3 length. Some sequences had exit status of timeout and segmentation fault. The Fig.4 shows the times taken by successful sequences of length 3. The x axis is the serial number of permutations arranged in lexicographic order and y axis is the time in seconds.

C. Impact on attack resistance

Although we have experimentally found out that if an adversary tampers the ROP augmented code he or she will not succeed and execution will result in a segmentation fault. However, in this section we mathematically prove that our optimal ROP chain is resistant to any tampering. The strength of ROP chain depends on the actual entropy of the underlying gadget selection. When a ROP chain is generated by a process that randomly selects a sequence of gadgets of length L from a set of N possible overlapping gadgets, then the number of possible ROP chains can be found by raising the number of symbols to the power L, i.e. N^L. Increasing either L or N will strengthen the generated ROP chain. The strength of ROP chain can be measured by the information entropy S, which is the expected value (average) of the information contained in the chain. For expressing it in term of binary digits we take base-2 log of N. Each gadget in a ROP chain is produced independently. Thus, the entropy

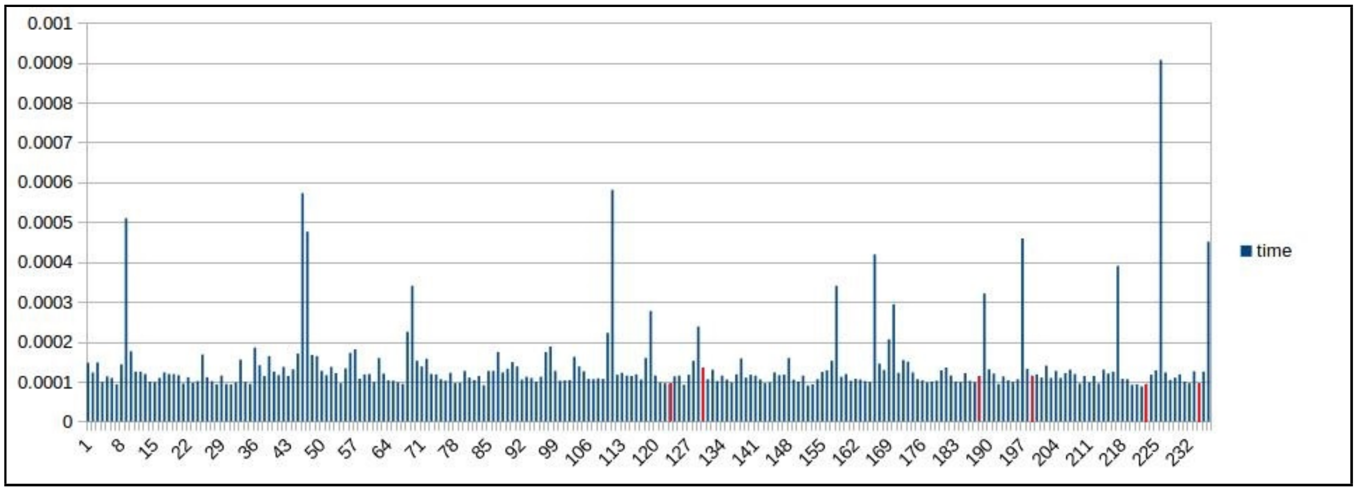

Fig. 4: Run Time Performance Evaluation

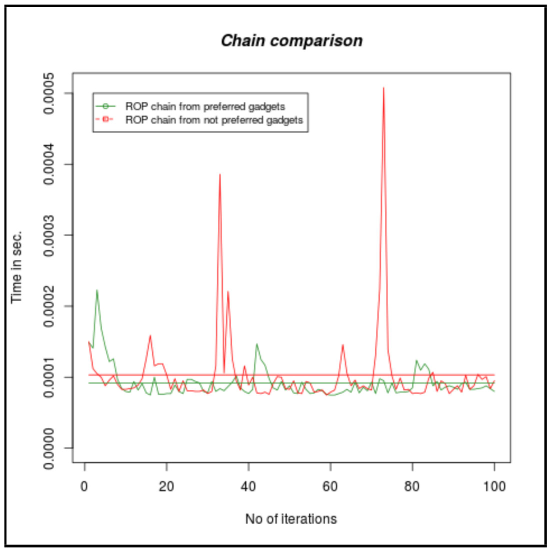

Fig. 5: Comparison between Best Gadgets and Random Selected Gadgets Chain

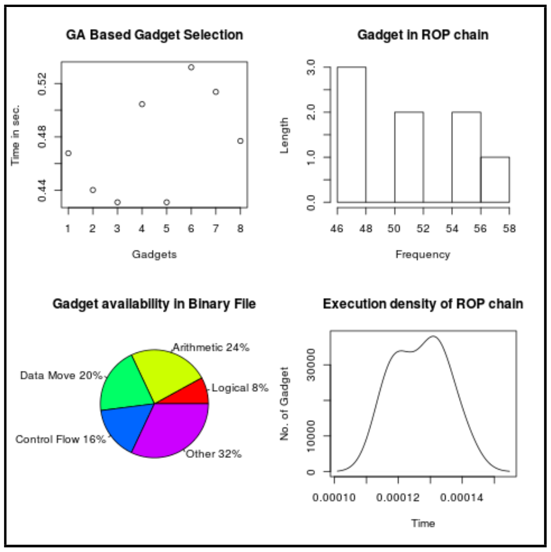

Fig. 6: Analysis of Gadget in Various Parameter

of a ROP chain is given by equation 1.

$$S = log_2 N^L = L\, log_2 N \tag{1}$$

So, from equation 1 we can say that, it is almost impossible to generate the same chain again. Hence, we can assuredly say that building a ROP chain from the best set of gadgets hardens the security strength The amount of effort and computation power that the attacker will have to put in order to bypass these security measures will be of high magnitude sometimes not even commensurate with the returns he will get from tampering with the software. This will serve to deter the attackers from mounting an exploit and tampering with the code.

D. Runtime analysis

Fig. 5 shows a comparison between preferred and non-preferred gadget chains. The average execution time of the optimal ROP chain is lower than the average execution time of the random ROP chain. Fig. 6 shows the number of times different lengths of gadgets appearing in the optimal solutions, the execution time of each gadget, different types of gadgets available in the binary code and execution density of the ROP chain.

VI. Future Scope and Conclusions

In this research we introduced novel code protection techniques using ROP which is an exploitation technique in itself. Our approach can protect code in non-deterministic way. GA helps us choose the subset of gadgets using which we can make an ROP chain with similar execution time as that of the normal code. Thus, performance sensitive code is protectable without any slowdown. GA also chooses the gadgets which take less time for execution making it an optimal chain.

In future we plan to explore techniques for protecting ROP chains which itself is a challenging task. We also plan to experiments our code verification techniques on other platforms like MIPS.

VII. Acknowledgements

This work was supported by Department of Electronics and Information Technology(DeitY) and Netherlands organization for scientific research(NWO).

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] Dennis Andriesse, Herbert Bos, and Asia Slowinska. Parallax: Implicit code integrity verification using return-oriented programming. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 125–135. IEEE, 2015.

[3] Kurt Baumgartner. The rop pack. In *Proceedings of the 20th Virus Bulletin International Conference (VB)*, 2010.

[4] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[5] Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[6] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[7] Hoi Chang and Mikhail J Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2002.

[8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[9] Stephen Checkoway, Ariel J Feldman, Brian Kantor, J Alex Halderman, Edward W Felten, and Hovav Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. *Proceedings of EVT/WOTE*, 2009, 2009.

[10] Dino Dai Zovi. Practical return-oriented programming. *SOURCE Boston*, 2010.

[11] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A framework for automated architecture-independent gadget search. In *WOOT*, 2010.

[12] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE Computer Society, 2013.

[13] Ralf Hund, Thorsten Holz, and Felix C Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium*, pages 383–398, 2009.

[14] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 601–615. IEEE, 2012.

[15] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib (c). In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 60–69. IEEE, 2009.

[16] Kevin A Roundy and Barton P Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR)*, 46(1):4, 2013.

[17] Jonathan Salwan. Ropgadget tool, 2012.

[18] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[19] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[20] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[21] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395. IEEE, 2010.

[22] Glenn Wurster, Paul C Van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *Security and Privacy, 2005 IEEE Symposium on*, pages 127–138. IEEE, 2005.