Protection against code exploitation using ROP and check-summing in IoT environment

***Abstract*—The operations of devices in automated, possibly in hostile environments, puts the dependability and reliability of the IoT systems at stake. More specifically, adversaries may tamper with the devices, tamper with sensor values triggering false alarms, instrument the data gathering and overall operation to their own interest. Protecting integrity and confidentiality of IoT devices from tampering attempts is a big challenge. Protection against code tampering is the focal point of this research. This paper entails a contemporary methodology to guard the code against exploitations. The approach focuses on a novel distributed solution by which the tamper resistance of the program code is magnified by the inclusion of two modules that work in tandem with each other. These security modules employ Return Oriented Programming (ROP) techniques and code check-sum techniques to protect critical pieces of code. When working together they provide dual lines of defence to the critical piece of code where the malicious entity has to bypass both the modules in order to tamper the critical piece of code thereby hardening the overall security and increasing the cost of exploitation drastically making it infeasible to mount an attack on IoT devices.**

Keywords:- IoT, ROP, Guards, Check-summing.

I. Introduction

Smart connectivity with the existing networks and context-aware computation using network resources is an indispensable part of IoT. But this prominent feature comes with threat where an adversary may tamper the code or clone IoT devices for his or her benefit. There are a myriad of malicious entities who rely on vulnerabilities present in the code deployed in the real world in order to mount an attack [18]. The attack can range from creating modified versions of the software and selling it in the market to making the code perform tasks that it isn't supposed to. Malicious code injection has been a major bottleneck for either providing a service from a stand-alone machine or receiving a service from a remote server. The importance of this attack is presented substantially in the recent targeted threats like Heartbleed [7] and Shallshock [11].

Initially, ROP found it's usage as an exploitation procedure that permits the rendering of the arbitrary code in the presence of $W \oplus X$ [16]. Gadgets which are, in fact, short return-terminated instruction sequences are in a chain fashion constructed by the arrangement of their respective addresses on the stack. Each return that gets concluded shifts the locus of control to the next available gadget. ROP behaves as a Turing complete programming technique, in the presence of abundant number of gadgets, which can execute random computations on the top of a host program. In most of the programs, a presence of a Turing complete gadget is witnessed [15].

The code verification methodology adds to the robustness of the code by the inclusion of ROP gadgets in it. Instructions are tabbed from a program under protection and are translated into ROP chains which implement the overlapping gadgets. Translated instructions malfunction if the gadget undergo some sort of tampering. This helps in implicit verification of the integrity of the protected code. Thus, the rendered instructions are referred as verification code [1]. In order to address the issues mentioned above we first use ptrace checker. Ptrace is a system call used for debugging purposes. Our aim was to use ptrace as an anti-debugging call. If an adversary tries to debug the code during run time using gdb compiler then ptrace checker returns an error and generates a segmentation fault. We used cross verifying code check-sums [5] in conjunction with ROP so as to provide multiple lines of defence against code tampering by a malicious entity.

Our work in this presents a security blueprint for protection of program code against tampering. The primitive ideas to a modified, generalized setting is proposed in which a program is preserved by a legion of functional units, such as, gadgets, guard etc. in integration with the program. The attackers are prevented from predicting the form of the chain by a multitude of methodologies to form the ROP chain which guards the network. For a higher order of protection, the number of gadgets can be escalated to a required number substantiating the desired level of protection.

The organization of the rest of paper is detailed as follows. Section 2 discusses related work, Section 3 discusses background, while Section 4 provides an overview. In Section 5, we describe the system. Section 6 discusses the experimental result, and we summarize our work in Section 7.

II. Related Work

Andriesse et al. [1] created ROP chains for protecting the code . But, ROP alone is not sufficient to mitigate run time attacks. In this work we add another layer of protection.

According to Borello et al. [3], Roundy et al. [13] and Saidi et al. [14], code protection primitives like integrity verification are widely used in practice to delay reverse engineering attacks, and to deter non-persistent adversaries. For example, code protection is commonly used by a malware to prolong its lifespan and monetization period. However, methods like ROP can easily break the code protection integrity without modifying the program code.

Control Flow Integrity (CFI) is another technique that 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]. There are plethora of other software based approaches for addressing the above said problem which might range from usage of self modifying code [9] (code that generates other code at run-time)

After ten years of the discovery of the return-to-libc technique, the wide adoption of non-executable memory page protections in popular OSes raised curiosity in the endeavours to avert advanced form of code reuse attacks. ROP exploits are facilitated by the lack of complete address space layout randomization in both Linux [12], and Windows [6], which otherwise would prevent or at least hinder [17] these attacks. As far as authors know, other than Andriesse et al. [1] no other researcher has used ROP as a defensive mechanism. Authors in [1] have suggested usage of check-summing as a possible measure of second line of defence which we intend to propose in this work. We would also explore possibility of strengthening security strength of the IoT code by devising cross referencing guards using checksums to protect one another.

III. Background

A. Return-Oriented Programming

Return Oriented Programming (ROP) is a generalized technique of return-into-libc [16] attacks by virtue of which an attacker can motivate the program to boomerang back to arbitrary points embedded in the code. This grants one to operate malicious computations without the necessity of injection of any new malicious code by obtaining control of the execution flow. This section, in particular, details a concise sketch of ROP. The reader, in order to grab more knowledge can find a detailed study in [4].

TABLE I: Available CPU architecture defence

Architecture	Type	ASLR	NX	ROP-Attack
X86	CISC	YES	YES	YES
X64	CISC	YES	YES	YES
ARM	RISC	YES	YES	YES
ARM64	RISC	YES	YES	YES
MIPS	RISC	YES	YES	YES

Table 1 shows the ROP functionality can be clearly depicted in various platforms including x86 [16], SPARC [4] and ARM [10].ROP uses short instruction sequences profoundly that can be found in a host's program memory space mostly identified as gadgets, each of which terminates with a return instruction. Each gadget is endowed with an operation of basic nature i.e., either addition or logical comparison. A stack encapsulates a chain of gadget addresses that constitute the ROP program such that with the termination of each gadget via each return instruction the control gets transferred to the immediate neighbouring gadget in the chain thus, forming the modus operandi of this approach.

Fig.1 describes the manner in which the ROP chain works. The stack pointer (esp) points to the address referring to the first gadget g1 in the chain. Once the return instruction

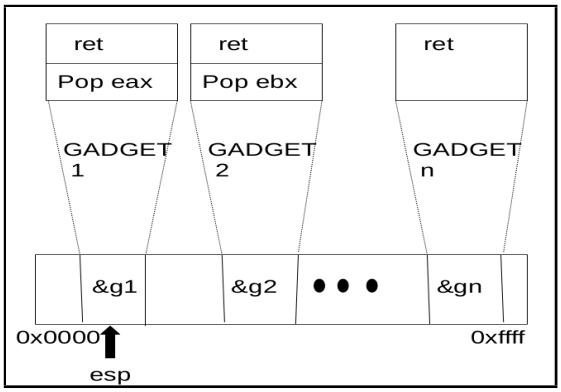


Fig. 1: ROP chain, in which gadgets take care of instructions

is executed the control shifts to the current gadget. On performing a pop operation, the stack gets loaded with a binary code value (that is subsequent to eax register) into the eax register that, in turn, increments esp to point to the gadget g2. 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 [16].

B. Check-sum code

A network of execution units called guards are incorporated within a program providing the necessary protection. The guard, in particular, is nothing but a code fragment that helps in the execution of certain security sensitive actions that take place during the execution of the program. Guards offer the flexibility to do any computation,e.g., whether its check-summing another code fragment at the point of runtime and check it's integrity (i.e., to review if the code has not been tampered) etc. In case the code that has been guarded is found altered, the guard possesses the authority to fire the suitable actions necessary at that moment. These actions might range from silent login of the detection event to making the software inoperative,e.g., by means of halting the execution that may lead to a crash that will be untraceable to the guard in question. In case no modifications are made to the code, the program executes normally. The programs shielded by check-summing techniques are in some way aware of their own individual integrity [5].

C. Threat Model

The hostile host threat model forms the base model for verifying tamper-proofing techniques. It is presumed that the application made tamper proof is performed on a system that is controlled by a hostile user, which holds the entire control on the runtime environment and might be modifying the tamper-proofed executable. The sole intention of the hostile user is presumed to bypass access controls in the protected application, in particular, anti debugging checks. The challenge faced by our tamper-proofing methodology is

to maximize the attempts required by the user to be successful in tampering with the protected code without considering the trusted components in the runtime environment [1].

IV. Overview of the System

A. IoT

Fig. 2 shows three layer architecture of our test-bed developed at our campus. In this test-bed there are various types of sensors like light, temperature, smoke, energy etc. used. IoT node collects all sensor values and passes it to gatway layer. Gateway layer pre-processes collected data and passes it to the server. Server uses this data for triggering various autonomous and intelligent activities like finding the presence of a person in a room etc. Server also connects to the Internet and hence, users can also view sensor values.

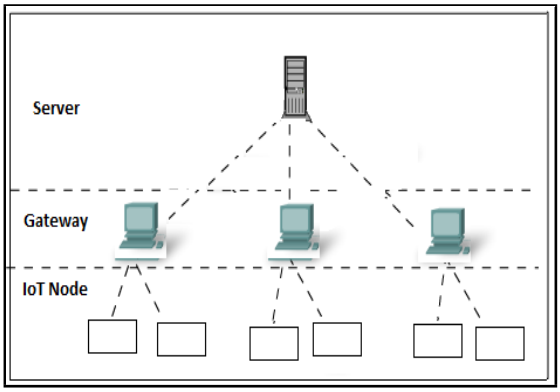


Fig. 2: IoT architecture

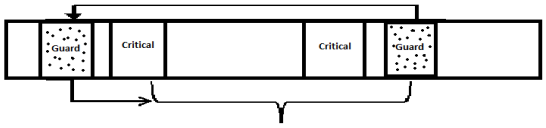
B. Proposed system security

In this section, we give an overview of how our system implements its protection mechanism. Our technique protects against memory corruption in both static and dynamic way. Thus, we protect against attacks ranging from the circumvention of anti debugging checks to large-scale software cracking. Fig. 3 illustrates how our system protects a binary.

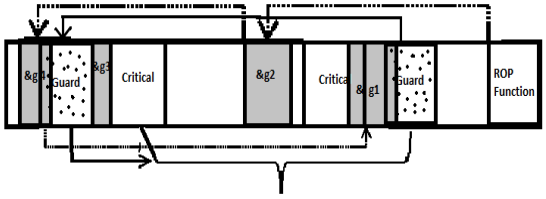
Now initially being used as an exploitation technique, researchers have found out that it can indeed be used as a code verification technique as well by passively verifying the integrity of the protected code. We protect the code by overlapping ROP gadgets with it. Then, selected instructions from the protected program are translated into ROP chains which use the overlapping gadgets. Since tampering with the gadgets cause the translated instructions to malfunction, this implicitly verifies the integrity of the protected code. Thus, we refer to these translated instructions as verification code[1].

To protect code integrity we generate ROP gadget payload, put that payload to verification function then with the help of gcc we find assembly level code then apply check-sum method. In the check-sum method we put a cross verifying network of guards on code, these guard protect ROP chain and security sensitive area.

(a) Identifying critical pieces of code to be protected



(b) Deploy check-sum guard over the critical code



(c) Create a ROP function which uses random gadget from binary file

Fig. 3: An overview of code protection model

C. Security sensitive code

The attacker mainly tries to identify access control points in the code and tries to circumvent them. We consider following types of code as security sensitive:

- User Authentication Points
- Memory vulnerability
- Anti-Debugging Checks

To collect these security sensitive points we use PIN tool, clang, flaw finder tools and we use following heuristics [8].

- DB - Branches that were taken in one case and not taken in another.
- DF - Functions that returned 1 in one case and 0 in another.

1) A DB is located inside a DF. This could mean that the function returns 0 or 1 depending on the branch being taken or not.
2) DB is located inside the parent function of a DF. This means that the parent function actually performs the authentication and returns the respective value. In our case both the DB as well as the branch inside the parent function can be considered as critical points in the program.

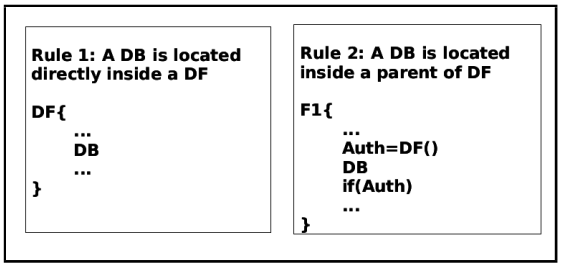


Fig. 4: Heuristics for finding security sensitive code

D. Algorithm

Algorithm 1 Algorithm to generate a ROP chain

Input: G : set of gadget[g1,g2..., gn], A : Addresses of functions and security sensitive code
Output: payload (PL) with ROP chain.

1: **if** (A.length > 0) **then**
2: **for** $i = 0$ to length of A[] **do**
3: $PL \leftarrow rand(G[n])$;
4: $PL \leftarrow A[i]$;
5: $PL \leftarrow rand(G[n])$;
6: **end for**
7: **end if**
8: **return** payload with ROP chain.

In algorithm 1, we exhibit how to frame a payload with ROP chain. For this, we use Ropgadget compiler to find Turing complete gadgets. Every gadget ends with a return statement so it can be used for creating a ROP chain.Then we prepare a list of security sensitive functions for protection from the existing program. Now we put gadget payload to verification function which decides the execution sequence. Thus, we hacked normal execution of program. After uploading payload, execution is carried out according to the ROP chain. Here, construction of ROP chain is random. It is extremely time consuming for an adversary to reverse engineer the ROP chain and decode the execution sequence. To decode the execution sequence, the adversary may try to debug during runtime but in our method we use ptrace detector that will not allow the adversary to debug during runtime. If adversary tries to run gdb then ptrace debugger simply stops any such efforts and quits from gdb.

Even though the self verification defense mechanism used above is a robust technique but still has some shortcomings discussed in the previous sections which we rectify using a network of cross verifying checksum guards.Algorithm 2 explain overall method to genrate tamperproof code protection using checksum guard and ROP chain.

Fig. 3b shows a simple cross verifying guard network [5]. In the Fig. 6 the guards are working in protecting the code segments and in addition the guards protect each other as well. This is done so as to ensure that the code segments as well as the guards themselves are protected from tampering. e In this algorithm we first set ptrace detector on vulnerable code of programs, so if adversary tries to debug program, it returns an error code. Then add variable like key and test for time to time perform xoring and check the integrity of code. Now we add check-sum guard as per the guard template. During instantiating of the guard, the system initializes $client_start$ and $client_end$ with the addresses of the target code range that the guard needs to protect. The check-sum value is obtained by the system. In this experiment we deploy two guards. First guard protects whole code and second guard. Second guard protects first guard. If any of the guards get currpt then it will generate

Algorithm 2 Algorithm to construct check-summing

Input: Assembly level code,Vulnerable function set V.
Output: Check-sum and ROP protected code

1: Set ptrace detector on vulnerable code.
2: Assign key and test variable with pre-calculated register value.
3: $ebx \leftarrow valueofcheck_sum_guard$.
4: Set $client_start$ in the beginning of code to be protected.
5: $ecx \leftarrow client_start$.
6: Every instruction, compare ecx with $client_end$.
7: **while** $ecx < client_end$ **do**
8: ebx= dword[ecx]+ebx
9: ecx= ecx+4
10: **end while**
11: If it is greater then corrupt ebp register value and generate segmentation fault.
12: Set $client_end$ at the end of code.
13: Perform XOR test on variable key and test.
14: **if** (result == 0) **then**
15: exit normaly
16: **else**
17: Currupt ebp register.
18: **end if**
19: **if** (Address of ROP gadget changed) **then**
20: Call algorithm 1
21: **end if**
22: **return** Check-sum and ROP protected assembly level code

segmentation fault. In addition to this, XOR test is also applied in this code. At the end we apply the ROP chain.

E. System description

We have built a security mechanism that include verification function below in the program. This function contains

```
void verify()
{
        char *rop="AAAAAAAAAAAAAAAABBBB\
\x5a\xe8\x06\x08\x60\xa0\x0e\x08\x36\
\xae\x0b\x08Yes!\x8d\xa1\x09\x08\x70\
\xf6\x04\x08\x5a\xe8\x06\x08\x60\xa0\
\x0e\x08\x24\x8e\x04\x08\x36\xae\x0b\
\x08\xef\xbe\xad\\xde\x60\xe7\x04\x08";
        char buffer[4];
        strcpy(buffer,rop);
}
```

Fig. 5: Verification functiom to put ROP chain.

simple ROP chain for our hot code. In this verification function we mention address of our hot code, this hot code will not run without the help of this verification function.This contains a execution chain, if adversary tampers it, the execution stops. To protect this chain we add a cross verifying network of check-sum guards as well. These

guards are responsible for performing code check-summing. The following example reflects a guard template, which is given the functionality to corrupt stack frame pointer ebp in case the computed check-sum is found to be different from the pre-calculated check-sum.

```
# Guard  Template
#guard is verifying guard1+
#protected code and vice versa
        guard:
                movl .checksum,%ebx
                movl
$client_start,%ecx
        for:
                cmpl $client_end,%ecx
                        jg end
                addl (%ecx),%ebx
                addl $4,%ecx
                jmp for
        end:
                cmpl .key,%ebx
                je succ
                addl %ebx,%ebp
        succ:
        movl $4, 8(%esp)
        client_start:
        ......
        ......
        (Source code for protection)
        ......
        guard1:
                movl .checksum,%ebx
                movl  $guard,%ecx
        for1:
                cmpl $succ,%ecx
                jg end1
                addl (%ecx),%ebx
                addl $4,%ecx
                jmp for1
        end1:
                cmpl .key1,%ebx
                je succ1
        client_end:
                addl %ebx,%ebp
        succ1:
        jmp     .L4
```

Fig. 6: Guard template used for check-summing

These guards will protect our code in assembly level. After completing this we recalculate ROP chain for verification function and hot code and put in a verification template that is shown above. This will protect our code in data cache. Then finally we create executable file of that code is tamper proof [5].

V. Experimental Results

In this section, the robustness of this technique is tested against hostile entities and the resources required and overheads while mounting this security measure have been evaluated.

A. Experimental setup

The security measures were mounted and tested on C programs which were being run both on laptop running Ubuntu as well as on IoT test-bed which has been developed in our campus.

TABLE II: Experiment Environment

Resource	System1	System2
Category	Laptop	IoT Node
OS	Ubuntu	Linux
CPU	Core i5	Intel Quark
Cache	3072 KB L2	16 KB
RAM	4 GB	256 MB

In our naively developed IoT test-bed, the IoT nodes capture various sensor values like temperature, light, motion, energy etc. and send them to a server via gateway.Fig. 8 shows run time performance evaluation that shows performance overhead is negligible.

For generalization we also test our code protection method on Ubuntu 14.0, this method work well on both environments.

B. Security

In this section we discuss attack resistance of our code.

1) Impact on attack resistance: Return Oriented Programming has certain vulnerabilities as discussed before when used as a standalone technique. Deploying code check-sum in conjunction with ROP addresses the vulnerabilities discussed by providing a second line of defense to the critical pieces of code. We have experimentally found out

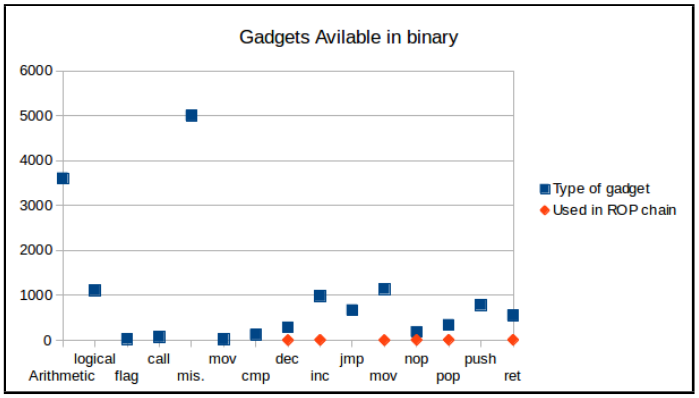


Fig. 7: Random selection of gadgets from binary

that the only way an attacker can come in and tamper with the critical piece of code is only if he circumvents both the ROP functionality and the code check-sum guard network that we have put into place. But ROP chain created with ranom selection of gadget and with random length of chain makes the task of an attacker difficult. Random ROP chain consists of a hexadecimal addresses of specified length taken from some set of gadgets using a random selection process in which each gadget is equally likely to be selected. The gadget can be individualy collected addresses from a binary. The strength of ROP chain depends on the actual entropy of the underlying number generator. This technique is serving the purpose of hardening the security. 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.

TABLE III: Statistics of Protection Mechanism

S.No.	Type of Binary	File Size	Security	Level
1	Plain	734.5 KB	ASLR,DEP	Low
2	ROP Based	733.4 KB	1 + ROP gadget	Medium
3	ROP + Guard	734.6 KB	2 + Guard	High

2) Runtime analysis: Table 3 shows statistics of security level. If a binary run without any security it only uses system provided security like ASLR, DEP then adversary easily can tamper it [16]. Now if we apply only ROP gadget based security it provides self code verification and we provide another layer of defence by using code check summing guard. This level of security ensure code self verification and integrity.
If an adversary tries to analyse the code using compiler utilities like gcc, he would not be able to use them owing to our utilization of the ptracer module. At run time, if adversary tries to add some code then our self verification technique shows segmentation fault. Adversary may also try to modify code in runtime. Due to ptrace detector adversary is not able to execute gdb also, the run time exploitation not possible.

C. Impacts on Program Size

In our experiment we add minimum number of guards and a small verification function in original C program that will not increase the size of file. The modified file is almost same in size when compared to the original file. We believe the issue of storage space does not pose a problem.

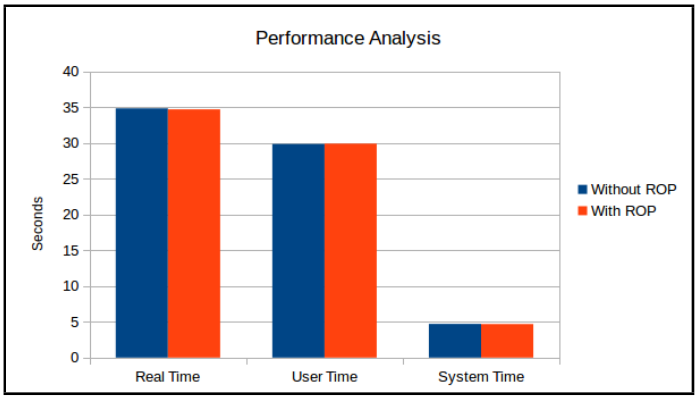


Fig. 8: Run time performance evaluation

We use Intel Vtune to analyse the CPU performance. Fig. 8 shows a histogram that shows the percentage of time to the specific number of CPU's were running simultaneously. Spin and overhead time adds to the idle CPU usage value. This result also shows that our method will not effect CPU performance. It will increase security, and hence, this new method will be effective for non-cryptic workload.

VI. Future scope and conclusions

We introduced novel code protection from exploitation using return oriented programming and check summing. Our approach can protect non-deterministic code. The performance overhead of our approach can be confined to verification code which is separate from the protected code. Thus, performance sensitive code is protectable without any slowdown, confining the performance penalty to other code.

In future we plan to calculate guards and ROP chain automatically. And try to apply this technique on different architecture like MIPs, ARM etc.

References

[1] 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.

[2] 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.

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

[4] 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.

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

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

[7] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, 2014. ACM.

[8] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P Kemerlis, and Angelos D Keromytis. Adaptive defenses for commodity software through virtual application partitioning. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 133–144. ACM, 2012.

[9] HG Joepgen and S Krauss. Software by means of the'protprog'method. ii. *Elektronik*, 42(17):52–56, 1993.

[10] Tim Kornau. Return oriented programming for the arm architecture. *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[11] Ruth Leys. Traumatic cures: Shell shock, janet, and the question of memory. *Critical Inquiry*, 20(4):623–662, 1994.

[12] 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.

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

[14] Hassen Saıdi, V Yegneswaran, and P Porras. Experiences in malware binary deobfuscation. *Virus Bulletin*, 2010.

[15] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[16] 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.

[17] 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.

[18] 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.