Protection Against Code Exploitation Using ROP and Check-summing in IoT Environment

Rajesh Shrivastava, Chittaranjan Hota
Department of Computer Science
BITS Pilani, Hyderabad campus, INDIA
{p2015005, hota@hyderabad.bits-pilani.ac.in}

Prashast Shrivastava
Purdue University, USA
West Lafayette, IN 47907, USA campus
srivas41@purdue.edu

***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 exploitation. 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-summing 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 an Internet of Things (IoT) environment. But this prominent feature comes with threat where an adversary may tamper the code or clone IoT devices for his or her benefit. 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 [5] and Shellshock [10].

Initially, Return Oriented Programming (ROP) found it's usage as an exploitation procedure that permits the rendering of the arbitrary code in the presence of $W \oplus X$ [15]. 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 [14].

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 used cross verifying code check-sums [4] in conjunction with ROP so as to provide multiple lines of defence against code tampering by a malicious entity.

Our work 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, guards etc. in integration with the program. The attackers are prevented from predicting the form of the chain by a multitude of methodologies to construct the ROP chain which protects the network. In due course, the number of gadgets can be escalated to a required number substantiating the desired level of protection. Existing code tamper proofing methods, such as, Control Flow Integrity(CFI) [17], code protection by inclusion of guards [4] etc. have contributed significantly to run-time overhead which leads to a deterioration in performance. The work done in this research does not degrade the run time performance of the ROP augmented code as compared to the normal code significantly.

The organization of the rest of paper is detailed as follows. Section II discusses related work, Section III discusses background, while Section IV provides an overview. In Section V, we describe the experimental results, and we summarize our work in Section VI.

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 Roundy et al. [12] and Saidi et al. [13], code protection primitives like integrity verification are widely used in practice to delay reverse engineering attacks, and to deter non-persistent adversaries. However, methods like

ISBN: 978-1-5090-4911-0 (c) 2017 IEEE

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

ROP can easily break the code protection integrity without modifying the program code.

Control Flow Integrity (CFI) is another technique that prevents 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 [8] (code that generates other code at run-time) to usage of encryption and decryption mechanisms.

After ten years of the discovery of the return-to-libc technique, the wide adoption of non-executable memory page protections in popular Operating Systems raised curiosity in the endeavours to avert advanced form of code reuse 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 have also explored strengthening of security strength of the IoT code by devising cross-referencing guards using check-sums to protect one another.

III. Background

A. Return-Oriented Programming

Return Oriented Programming (ROP) is a generalized technique of return-into-libc [15] 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 [3].

Table I shows that the ROP functionality can be clearly depicted in various platforms including x86 [15], SPARC [3] and ARM [9]. 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. 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

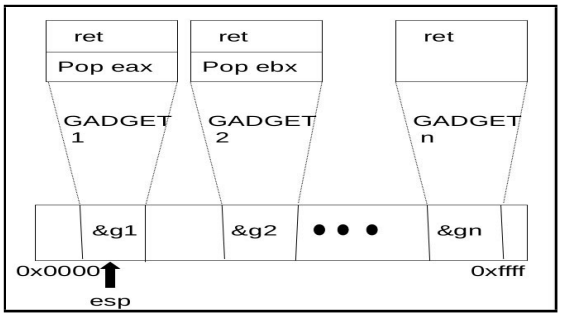


Fig. 1: ROP Chain with Gadgets

the first gadget g1 in the chain. Once the return instruction 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 other gadgets g4, . . . , gn are executed [15].

B. Check-sum code

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. The programs shielded by check-summing techniques are in some way aware of their own individual integrity [4].

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].

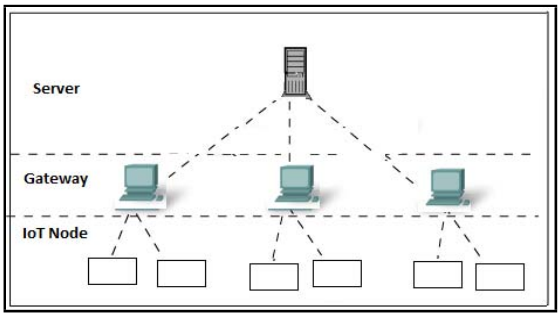
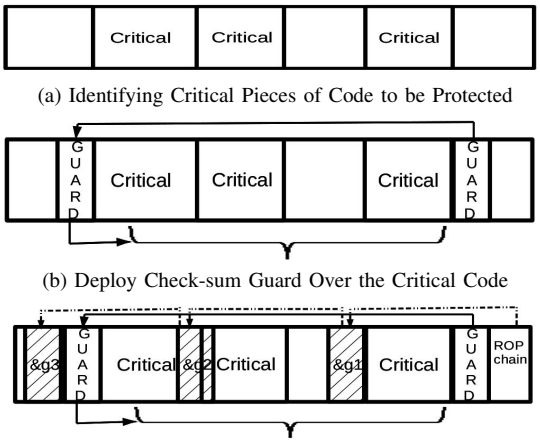


Fig. 2: IoT Architecture



(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 Gadgets from Binary File

Fig. 3: An Overview of Code Protection Model

IV. Overview of the System

A. IoT

Fig. 2 shows three layer architecture [18] [16] 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 gateway 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.

B. Proposed system security

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.

We protect the IoT code by overlapping ROP gadgets within it. 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].

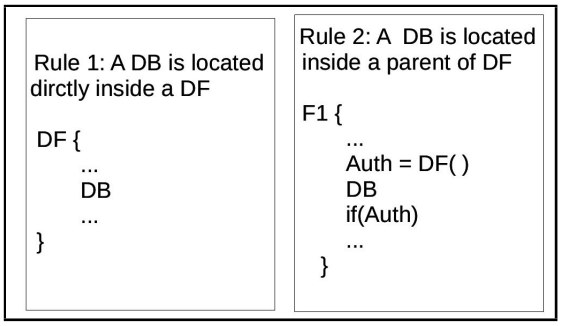


Fig. 4: Heuristics for Finding Security Sensitive Code [7]

To protect code integrity we generate ROP gadget payload, insert this payload into verification function and then with the help of gcc we find assembly level code and then apply check-summing method. In the check-summing method we insert a cross verifying network of guards on the code, these guards protect ROP chain and security sensitive area in the IoT code.

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 vulnerabilities
- Anti-Debugging Checks

To collect these security sensitive points we use PIN tool [11], flaw finder [6] etc. tools and use following heuristics [7]:

- 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.

D. Algorithm

In algorithm 1, we exhibit how to frame a payload with ROP chain. 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 controlled normal execution of the program. After uploading the 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

Algorithm 1 Algorithm to generate a ROP chain

Input: G : set of gadgets[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.

engineer the ROP chain and decode the execution sequence. 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 generate tamper-proof code protection using checksum guard and ROP chain. 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.

Algorithm 2 Algorithm to construct check-sum

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

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

```
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 Function

E. System description

We have built a security mechanism that includes verification function as shown in Fig. 5.

This function contains simple ROP chain for our hot code. In this verification function we mention address of our hot code, this critical code (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. Algorithm 2 inserts 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.

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 in Fig. 5. This will protect our code in data cache. Then finally we create executable file of that code which is tamper proof [4].

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.

In our naively developed IoT test-bed, the IoT nodes capture various sensor values like temperature, light, motion,

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

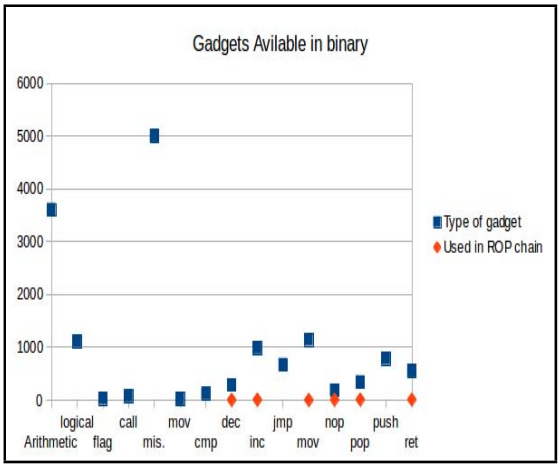


Fig. 6: Random Selection of Gadgets from Binary

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

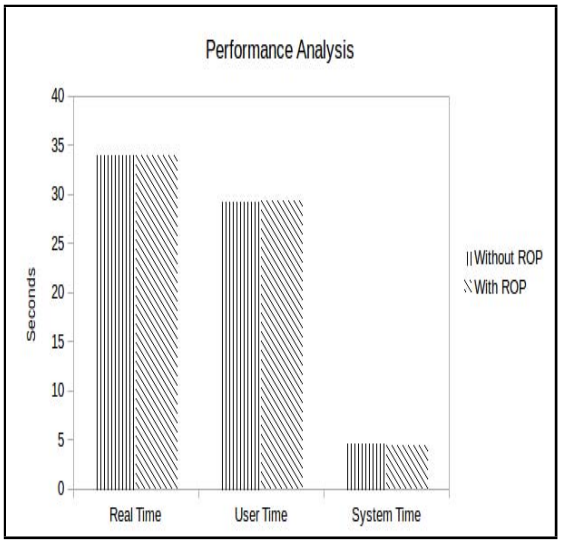


Fig. 7: Run Time Performance Evaluation

energy etc. and send them to a server via gateway. Fig. 7 shows run time performance evaluation that shows negligible performance overhead.

For generalization we also test our code protection method on Ubuntu 14.0, which work well on both the 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 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 random selection of gadgets and with random length of chain makes the task of an attacker difficult. Fig. 6 shows random selection of gadgets from available gadgets in binary. Random ROP chain consists of 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 individually collected addresses from a binary. The strength of ROP chain depends on the actual entropy of the underlying number generator. This technique serves 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 the code.

2) Runtime analysis: Table 3 shows statistics of security level. If a binary is run without any security it only uses system provided security like ASLR, DEP etc. then adversary easily can tamper it [15]. 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 ensures 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 anti-debug checker 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 anti-debug checker adversary will not be able to execute gdb also, and hence the run-time exploitation is not possible.

C. Impacts on Program Size

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

We use Intel Vtune to analyse the CPU performance. Fig.7 plots time taken in seconds by a program without ROP and

a program with ROP against various times, i.e., user, system and real-time. Spin and overhead time adds to the idle CPU usage value. This result also shows that our method will not affect CPU performance. It will increase security, and hence, this new method will be effective for non-cryptic workload also.

VI. Future scope and conclusions

We introduced novel code protection from exploitation using Return Oriented Programming and check-summing techniques. Our approach can protect non-deterministic code. The performance overhead of our approach can be confined to the 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 create guards and ROP chains automatically. Also, we try to apply this technique on different architectures like MIPs, ARM etc.

VII. Acknowledgements

This work was supported by Department of Electronics and Information Technology (DeitY), Govt. of India and Netherlands Organization for Scientific research (NWO).

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] 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.

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

[5] 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.

[6] Oliver Ferschke, Iryna Gurevych, and Marc Rittberger. Flawfinder: A modular system for predicting quality flaws in wikipedia. In *CLEF (Online Working Notes/Labs/Workshop)*, 2012.

[7] 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.

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

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

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

[11] Vijay Janapa Reddi, Alex Settle, Daniel A Connors, and Robert S Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, page 22. ACM, 2004.

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

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

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

[15] 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.

[16] Wayes Tushar, Chau Yuen, Kai Li, Kristin L Wood, Wei Zhang, and Xiang Liu. Design of cloud-connected iot system for smart buildings on energy management. *EAI Endorsed Trans. Indust. Netw. & Intellig. Syst.*, 3(6):e3, 2016.

[17] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940. ACM, 2015.

[18] Sanjana Kadaba Viswanath, Chau Yuen, Wayes Tushar, Wen-Tai Li, Chao-Kai Wen, Kun Hu, Cheng Chen, and Xiang Liu. System design of the internet of things for residential smart grid. *IEEE Wireless Communications*, 23(5):90–98, 2016.