Protecting Computer Systems from Cyber Attacks with Internal Interface Diversification

Sampsa Rauti

University of Turku, 20014 Turku, Finland
sjprau@utu.fi

Abstract. Internal interface diversification is a proactive technique that protects software and devices from malicious cyber attacks by making interfaces unique in every separate system. As malware cannot use the knowledge about internal interfaces in the system to its advantage anymore, it is rendered useless. The current study gauges the effectiveness of internal interface diversification by analyzing three cyber attacks as case studies and showing why internal interface diversification is an effective countermeasure against them. The study will further serve to demonstrate that internal interface diversification is a feasible and widely applicable security measure against numerous common cyber attacks.

1 Introduction

Malware is a huge threat in today's interconnected computer systems. New malicious programs keep appearing at a staggering pace. Hundreds of thousands of new malicious programs are being discovered daily [3]. After a vulnerability is found in a piece of software, adversaries can usually take advantage of it much faster than patches are created and delivered by software vendors. At the same time, anti-virus programs are struggling to keep up with this development, and what is more, they cannot even be run on many resource constrained environments, such as IoT. Therefore, novel proactive methods to mitigate malware attacks are needed [2, 10, 29].

To successfully carry out its objectives, a piece of malware has to know the target system's internal interfaces. For example, a piece of malware knows the system call number it should use or the library function to invoke in order to take use the services provided by the operating system. Therefore, instead of trying to detect each specific type of exploit and render it useless, an apparent solution to the malware problem is to make the well-known internal interfaces secret by obfuscating them. In other words, it is possible to uniquely diversify the internal interfaces in each system to render harmful programs useless and prevent adversaries from reaching their goals.

Trusted binaries and scripts in the system are then altered accordingly so that they conform to the new diversified "language" and can still operate normally. Malicious programs, however, do not know the diversification secret (e.g. a secret unique diversification function used to diversify the interfaces). They are prevented from operating correctly and cannot do

harm in the system. Moreover, even if the diversification secret of one system should somehow be compromised, other systems cannot be attacked with this knowledge because they have different unique diversification. The idea of diversifying the critical parts of a system is not new, and schemes following this general idea have been published in the academic literature [9, 22]. However, steps still need to be taken in the direction of applying diversification in practice and analyzing its effectiveness against dfferent threat scenarios.

While there are papers presenting obfuscation frameworks and gauging the practical applicability of this technique in real-world systems [13, 4, 23], less attention has been paid on analyzing the practical cyber attacks and demonstrating that diversification renders the exploits useless. Previously, Rauti et al. [25] showed that diversification could potentially mitigate or prevent over 80 % of the analyzed exploits. The current study continues this earlier work, but takes a different approach by analyzing three cyber attacks more profoundly as case studies and showing why internal interface diversification is an effective countermeasure against them. This will hopefully further serve to demonstrate that interface diversification is a feasible and widely applicable security measure against many common cyber attacks.

The rest of the paper is structured as follows. Section 2 presents the general idea of interface diversification and covers the interfaces suitable for diversification. Section 3 demonstrates how diversification can be used to defend against different practical cyber attacks. We present examples of real malware and analyze the reasons diversification is an effective countermeasure against these threats. Section 4 discusses the implications of our findings and Section 5 concludes the paper.

2 Interface diversification

2.1 The general idea

Interface diversification makes interfaces on each specific system unique. In other words, diversification is about making the operating environment unpredictable for the malware [17, 18]. Diversifying internal interfaces decreases the number of assumptions an adversary or a piece of malware is able to make about the surrounding system where it is run. In practice, diversification can be implemented using obfuscation transformations [9]. In interface diversification, even quite simple obfuscation transformations such as altering the names of functions and changing the order in which parameters appear in function signatures can be used.

An example of diversification is diversifying the system call interface. For example, in the Linux operating system, the numbers that identify the system calls can be altered according to a secret diversification function. When the interface is altered, we also have to modify the trusted libraries and programs that make calls to this interface accordingly.

In this context, the term *interface* is interpreted quite broadly. By interface, we do not only mean ordinary interfaces provided by software components but also for instance commands of a script language or memory

addresses of important services or resources are considered as interfaces to which diversification can be applied. Therefore, in a broad sense, an interface is anything used to gain access to essential services or resources of a device.

2.2 Interfaces suitable for diversification

The important internal interfaces include but are not limited to the following ones:

- *The system call interface.* The critical functionality of the device is accessed through the system call interface of the operating system. The system calls usually have numbers that are used to invoke them, and these numbers can be diversified (altered according to a secret diversification function) in order to prevent a harmful program that uses the system calls from working [6, 19, 23]. The system calls numbers in the trusted library and application binaries are then changed accordingly. This way, the trusted applications remain compatible with the operating system.
  The malware, however, would need to execute system calls in order to interact with its environment (that is, cause any real harm in the targeted system), but because it does not know the diversified system calls it can not do anything with them. For example, the code injection attacks in which the adversary gets the system to run malicious code containing system calls do not have any effect because the system call numbers are wrong in that specific system. Any other attacks where the attacker depends on system call numbers will also fail.
- *Library functions.* Diversifying the system call interface is a good start, but there are other ways to make system calls indirectly that the adversary could make use of. That is why we also have to prevent the malicious attacker from reaching the critical resources using operating system libraries [6, 23]. These libraries contain wrapper functions that directly or indirectly lead to invocation of system calls (in other words, this is the transitive closure of system calls in the binary files throughout the system). All these functions are diversified – their names are altered and e.g. order of parameters can be changed.
  Moreover, just like with diversification of system calls, we have to propagate the diversification of library functions. Therefore, all calls to these diversified library functions in binary files are also diversified. This means the symbol strings in trusted binaries are altered so that they still work with the diversified system.
- *The command line interpreter.* The protection of library functions and system calls can be circumvented by using the command line interpreters. A piece of malware could employ interpreted languages like command shell script languages. Similarly to the library functions, the command shell also gives an indirect access to the services provided by the operating system. To prevent this threat, the language interface used by the command line interpreter can be diversified [15, 28]. In practice, this means changing the tokens that

are recognized by the command line interpreter. In other words, the command interpreter is modified so that it supports running diversified scripts. The trusted scripts in the system are then diversified accordingly.

- *Address space.* The memory space can also be diversified. Address Space Layout Randomization (ASLR) randomly rearranges the address space locations of important parts of processes [1, 7, 30]. With randomized addresses, it becomes difficult for an attacker to find the memory addresses of a specific piece of code he or she might want to abuse [16]. In other words, the malicious adversary cannot reliably move to a particular position in the memory anymore. ASLR is the only diversification method that is already widely being used; many operating systems such as Linux and Windows implement it. However, not all IoT operating systems are armed with this protection method, for example.
- *SQL query language.* Domain-specific languages such as SQL (Structured Query Language) can also be seen as targets of diversification [5, 20, 24]. This provides protection against SQL injection attacks in which the adversary injects nefarious SQL commands for instance through a web form and the provided data gets interpreted as executable statements.

It is worth noting our study only targets internal interfaces. That is, the diversification focuses on the internal interfaces used by programs but not usually by users. Therefore, the user experience does not suffer from internal interface diversification. Also, unique versions of diversified software for different systems can be created automatically in many cases. Internal interface diversification works against attacks that depend on the knowledge about internal interfaces. Different injection attacks are a common example. Naturally, many other attacks such as circumventing a broken authentication mechanism cannot be prevented by diversification.

3 A Case Study of Defending against Three Cyber Attacks with Diversification

In what follows, we will take a close look at three different exploits and demonstrate how diversifying the internal interfaces prevents them from wreaking havoc in the target system.

3.1 The Mirai malware and OS library diversification

Mirai is a piece of malicious software that is used to create a large botnet by connecting number of infected devices called bots [14]. Mirai uses SSH and Telnet protocols for malicious login attempts, trying commonly used default user names and passwords. Then, without the consent of the owners, the infected devices are then used to launch attacks against computers on the Internet. These attacks are usually Distributed Denial of Service (DDoS) attacks, with a huge number (even tens of thousands) of bots sending traffic to a server, consuming resources and preventing

the server from responding to requests coming from normal clients. It is interesting to note that the devices Mirai infected are different from the infected devices usually gathered to form botnes. While most botnet attacks have in the past have made use of infected home computers, Mirai infected IoT devices such as security cameras [8].
The source code of Mirai [21] was released in October 2016. Mirai widely scans IP addresses and compromises IoT devices in order to increase the number of devices in the botnet. The malware can participate in different kinds of DDoS attacks, for example GRE IP flood and SYN and ACK floods, according to the instructions received from a remote command and control server. Mirai is also a territorial malware, as it also closes processes that use SSH, Telnet and HTTP ports, thus preventing remote connections to the IoT device [12]. Moreover, it searches and kills some competing malicious programs potentially residing on the infected computer.

```
396                     if (util_stristr(exe, util_strlen(exe), inode) != -1)
397                     {
398 #ifdef DEBUG
399                         printf("[killer] Found pid %d for port %d\n", util_atoi(pid, 10), ntohs(port));
400 #else
401                         kill(util_atoi(pid, 10), 9);
402 #endif
```

Fig. 1. An excerpt from Mirai's source code. The code kills a process that uses a specific port number.

```
157     int pid1, pid2;
158
159     pid1 = fork();
160     if (pid1 == -1 || pid1 > 0)
161         return;
162
163     pid2 = fork();
164     if (pid2 == -1)
165         exit(0);
166     else if (pid2 == 0)
167     {
168         sleep(duration);
169         kill(getppid(), 9);
170         exit(0);
```

Fig. 2. An excerpt from Mirai's `attack_start` function that invokes several system call wrapper functions.

The Mirai source code makes use of the well known wrapper functions in order to indirectly use system calls. For example, we can see in Figure 1

that a short excerpt of the malicious code that kills a process that uses a specific port number in the system. If the `kill` wrapper function in C standard library was diversified by modifying the symbols in the binaries (the library and the trusted programs using it), Mirai would not be able to invoke it and would be rendered useless. Similarly, Figure 2 shows an excerpt from Mirai's `attack_start` function that uses several wrapper functions that act as wrappers for system calls, such as `fork` and `exit`. Aside from these examples, interface diversification would naturally also prevent many other actions taken by the malware, such as contacting the command and control server for further instructions.

In Unix-based systems, symbol modification can be done by rewriting executable ELF (Executable and Linkable Format) binaries. The symbols in an ELF file can be found by inspecting the sections with the `DYNSYM`-type. The symbols are then gathered, diversified and rewritten in the file. Changing symbols does not have a large effect on performance. However, if the new diversified symbols are longer than the old ones, the size of the ELF file grows.

It is worth noting that interface diversification is a good fit for resource constrained IoT devices. Unlike security solutions such as anti-virus software or intrusion detection systems, internal interface diversification is a very performance efficient security measure. Although Mirai used simple default passwords to compromise IoT devices and could be defeated by paying closer attention to this simple security issue, interface diversification would have provided another layer of security.

3.2 ShellShock and command language diversification

ShellShock is a vulnerability in Bash (the Unix command shell), first discovered in September 2014. With ShellShock, a malicious adversary can make Bash execute arbitrary commands and gain unauthorized access to public-facing services, for instance web servers that use the Bash shell to handle requests [26]. The vulnerability comes up when a newly created shell instance encounters an environment variable containing code that looks like a function and then evaluates it. The Bash code had a bug where the evaluation did not stop when the function definition ended. The vulnerability can be exploited as follows:

```
env x='() { :;}; echo TEST' bash -c :
```

In the code, the value set to the variable `x` bears a resemblance to a function definition. Here the function is just a single colon, a simple command that does nothing. After the semi-colon ending the function definition, there is an `echo` command. Although this command is not supposed to be here, nothing prevents the adversary from putting it there. Finally, a new shell instance is started, again with a colon command that does nothing.

However, when the new shell instance starts up and reads the environment we have provided, it also receives the reads the `x` variable. Because

the contents of this variable look like a function, they are evaluated. After the function definition gets loaded, the malicious payload crafted by the adversary is executed as well. Therefore, on a vulnerable system, the code above would print the string `TEST`. However, it is clear that the attacker could do something much worse than simply printing some innocent messages.

Apart from fixing the bug, a proactive approach to prevent a malicious attack exploiting ShellShock can be put in place. In this solution, a diversified version of the Bash command language is used in the system. When the keywords in the command shell language are diversified and the malware does not know them, it cannot get proper malicious code executed in the system. In the example above, the `echo` keyword will not work and the malware will not know how to print messages. For example, the diversified version of the command language can use a secret keyword `echo7419` instead of `echo`. What is more, the way function definitions are made and the way statements are ended can be completely different in the diversified version of the command language.

Each script in the system can be diversified differently for additional security but in this case the performance also suffers as each script has to be decoded by the modified Bash interpreter. For example in tests by Uitto et al. [28] the diversified scripts took 2.7 times longer to execute. However, even this is quite performance increase in systems where script execution does not play a large role.

The discovery of the ShellShock vulnerability clearly showed that widely used and trusted older software can still have unknown critical bugs and zero day exploits taking advantage of the vulnerability will appear. As a proactive security measure, interface diversification can protect against this kind of unpredictable and completely novel threats.

3.3 The "Advanced Power" botnet and diversifying the SQL language

One of the significant threats web applications face today is an SQL injection attack. Consider the following example that shows pseudocode run on a web server. It is used when a user authenticates with a user name and a password. The database has table users with columns username and password.

```
// Get the POST variables
user = request.POST['username'];
pass = request.POST['password'];

// Form the (vulnerable!) SQL query
sql = "SELECT id FROM users WHERE username='"
      + user + "' AND password='" + pass + "'";

// Execute the query
database.execute(sql);
```

In the example, an adversary can use malicious SQL commands in the input so that the SQL statement the database server executes is changed. For instance, if the password field is set to

```
password' OR 1=1
```

the server executes the following SQL query:

```
SELECT id FROM users WHERE username='username'
                    AND password='password' OR 1=1'
```

The `OR 1=1` statement makes the `WHERE` clause return the first id of the `users` table. It does not matter what the values of the username and password are. If the user listed first is an administrator (which is often the case), the attacker also gains administrator privileges.
Of course, inventing and testing malicious SQL inputs can be a tedious and time consuming process. Therefore, SQL injection attacks has been automated in many attacks, such as the "Advanced Power" botnet. This malware disguised itself as a legitimate Firefox add-on and used the infected machines to test several different SQL injection attacks on websites browsed by the user.
Much like in the case of Bash command line interpreter, the solution here is to diversify SQL so that the attacker does not know keywords. For example, in MySQL, this can be done by altering the symbols in the C header file `sql/lex.h`. This file contains a `symbols` array defining the keywords and the operators of the language. In the same manner, an array named `sql_functions` defines the SQL functions.
As a results of the changes, the diversified version of SQL engine excepts something like this as an input (here we have just added numbers after the keywords for clarity, but naturally the keywords could be better obscured):

```
SELECT3892 id FROM6370 users WHERE2081 username='username'
                    AND1187 password='password' OR0385 1=1'
```

The attacker, in the attack described above, has to know how the keyword OR is presented in the diversified language in order to be successful. For additional security, operators can be diversified as well. Also, the expected diversification of a keyword can even vary depending on time or on the location of that specific keyword in the statement. This way, even if the process of searching for SQL injection vulnerabilities is automated with a program like "Advanced Power", chances for success are very slim. Moreover, continuous attempts to guess keywords of the diversified language can be detected and stopped.

The attacks and countermeasures are summarized in Table 1.

Table 1. Summary of attacks and countermeasures.

Attack	**Countermasure**	**Performance overhead**
Malicious use of library functions	Diversify symbols in binaries	None/negligible
SQL injection	Use diversified SQL	Modest
Shell script injection	Use diversified command shell	Modest/moderate

4 Discussion

The previously discussed examples have shown that novel proactive solutions are needed in today's complex threat landscape. In this study, we have further demonstrated the effectiveness of internal interface diversification by presenting several real-life examples of vulnerabilities and exploits that can be defeated with diversification. Internal interface diversification prevents – or at the very least mitigates – several different exploits and vulnerabilities.

Internal interface diversification as a security measure has several advantages. First, it is a solution that effectively helps to mitigate the threat of large-scale cyber attacks targeting interfaces (e.g. injection attacks). Billions of devices are being distributed with identical publicly known interfaces. By breaking this software monoculture, diversification prevents the attackers from building a single exploit that could easily attack all instances of a specific piece of software.

Second, as we have mentioned, internal interface diversification causes only a negligible overhead in many cases. Many IoT devices, for instance, only have extremely limited resources in terms of computational power and memory. Diversification is a good fit for this environment because it only incurs either very small overheads or no overhead at all. Simply replacing the system call numbers with new ones or altering the keywords of a domain-specific language often has no performance penalties and no unfavorable effect on memory usage. Compared to resource intensive security solutions that scan the system for malware or monitor processes to detect abnormal behavior, interface diversification is much less performance-intensive. This also has positive implications on energy efficiency: in many cases, interface diversification provides a nice trade-off between sufficient security and energy consumption.

Third, many devices are not updated regularly or updates may be absent altogether. There are some embedded devices, for example, that cannot be updated at all. Still, when a vulnerability is found, these devices are left exposed to malicious attacks. Interface diversification does not (always) prevent the malware from entering the system and it most definitely does not fix vulnerabilities, but it makes it much more difficult for a harmful program to operate in the surrounding system. This also makes interface diversification *proactive*: it protects the system against many zero-day exploits that are not publicly known at the time they happen.

Fourth, although interface diversification is not a silver bullet that protects against all attacks, it is *orthogonal* in the sense it can be used in combination with other security measures such as encryption and intrusion detection systems. Moreover, internal interface diversification can act as a safeguard when some other security measure fails (such as weak passwords in Mirai's case). At the same time, diversification also has a wide applicability, as it can be applied practically to any critical internal interface in a computer system.

Finally, interface diversification usually prevents the propagation of malware. When a piece of malware cannot operate on a machine, it cannot spread to the surrounding network. This makes building botnets such as Mirai significantly more difficult.

Of course, there are also some limitations in the presented approach. First, internal interface diversification cannot prevent all attacks. For example, in Return-oriented Programming (ROP) attacks, the attacker executes carefully selected instruction sequences that are already present in the memory. By chaining these sequences together, an adversary can execute arbitrary code on the machine. Because these attacks do not depend on the well-known interfaces the same way many other attacks do, diversification is not in general a very effective measure against these attacks. There are, however, also some positive results on using fine-grained diversification to mitigate ROP attacks [11, 27].

Another limitation is that setting up the diversified system is not completely straightforward, as all trusted programs have to be diversified so that they correspond to the diversified interfaces. There are, however, promising results regarding automatic diversification of binaries [18, 23]. It would seem diversification is a very feasible security approach at especially in smaller environments with restricted set of programs such as IoT operating systems and devices.

5 Conclusion

In the current study, we have discussed internal interface diversification as a method that prevents malware from using the important resources of the target system. We presented analysis of three different real world attacks and demonstrated how diversifying different interfaces can prevent these attacks from working.

While diversification of internal interfaces has been discussed in the academic literature, operating systems and other pieces of software still do not widely employ this solution in practice – address space layout randomization being the only exception. However, although internal interface diversification may not provide the most usable security solution for all systems – such as a system of an average home user – at the moment, we believe it has lots of potential in systems with relatively small number of programs, especially in lightweight IoT operating systems. Therefore, we hope to see practical diversification solutions implemented and used to protect operating systems in the near future.

References

1. Abadi, M., Plotkin, G.: On protection by layout randomization. In: 2010 23rd IEEE Computer Security Foundations Symposium. pp. 337–351 (2010)
2. Albanese, M., Jajodia, S.: Proactive defense through deception. In: Industrial Control Systems Security and Resiliency, pp. 169–202. Springer (2019)
3. AVTest: Malware statictics. https://www.av-test.org/en/ statistics/malware/, accessed: 2019-08-20
4. Boyd, S., Kc, G., Locasto, M., Keromytis, A.: On the General Applicability of Instruction-Set Randomization. IEEE Transactions on Dependable and Secure Computing 7(3) (2008)
5. Boyd, S., Keromytis, A.: SQLrand: Preventing SQL Injection Attacks. In: Jakobsson, M., Yung, M., Zhou, J. (eds.) Applied Cryptography and Network Security, Lecture Notes in Computer Science, vol. 3089, pp. 292–302. Springer Berlin Heidelberg (2004)
6. Chew, M., Song, D.: Mitigating buffer overflows by operating system randomization. Tech. rep., CMU (2002)
7. Chongkyung, K., Jinsuk, J., Bookholt, C., Xu, J., Peng, N.: Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In: Computer Security Applications Conference, 2006. ACSAC '06. pp. 339–348 (2006)
8. Cloudflare: Inside the infamous Mirai IoT Botnet: A Retrospective Analysis. https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/, accessed: 2019-08-20
9. Cohen, F.: Operating system protection through program evolution. Computers & Security 12(6), 565 – 584 (1993)
10. Evans, N., Horsthemke, W.: Active defense techniques. In: Cyber Resilience of Systems and Networks, pp. 221–246. Springer (2019)
11. Gupta, A., Kerr, S., Kirkpatrick, M., Bertino, E.: Marlin: A Fine Grained Randomization Approach to Defend against ROP Attacks. In: Lopez, J., Huang, X., Sandhu, R. (eds.) Network and System Security, Lecture Notes in Computer Science, vol. 7873, pp. 293–306. Springer Berlin Heidelberg (2013)
12. Imperva: Breaking Down Mirai: An IoT DDoS Botnet Analysis. https://www.imperva.com/blog/malware-analysis-mirai-ddos-botnet/, accessed: 2019-08-20
13. Jiang, X., Wang, H.J., Xu, D., Wang, Y.: Randsys: Thwarting code injection attacks with system service interface randomization. In: Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on. pp. 209–218 (2007)
14. Kambourakis, G., Kolias, C., Stavrou, A.: The mirai botnet and the iot zombie armies. In: MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM). pp. 267–272 (2017)
15. Kc, G., Keromytis, A., Prevelakis, V.: Countering Code-injection Attacks with Instruction-set Randomization. In: Proceedings of the 10th ACM Conference on Computer and Communications Security. pp. 272–280. CCS '03, New York, NY, USA (2003)

16. Kim, J., Jang, D., Jeong, Y., Kang, B.B.: Polar: Per-allocation object layout randomization. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 505–516. IEEE (2019)
17. Larsen, P., Brunthaler, S., Franz, M.: Security through diversity: Are we there yet? Security Privacy, IEEE 12(2), 28–35 (2014)
18. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: Automated software diversity. In: Security and Privacy (SP), 2014 IEEE Symposium on. pp. 276–291 (2014)
19. Liang, Z., Liang, B., Li, L.: A System Call Randomization Based Method for Countering Code Injection Attacks. In: International Conference on Networks Security, Wireless Communications and Trusted Computing. pp. 584–587. NSWCTC 2009 (2009)
20. Locasto, M., Keromytis, A.: PachyRand: SQL Randomization for the PostgreSQL JDBC Driver. Technical report CUCS-033-05. Columbia University, Computer Science. 2005.
21. Mirai source code: https://github.com/jgamblin/Mirai-Source-Code, accessed: 2019-08-20
22. Portokalidis, G., Keromytis, A.: Global ISR: Toward a Comprehensive Defense Against Unauthorized Code Execution. In: Moving Target Defense, Creating Asymmetric Uncertainty for Cyber Threats, Advances in Information Security 54 (2014)
23. Rauti, S., Lauren, S., Hosseinzadeh, S., Mäkelä, J.M., Hyrynsalmi, S., Leppänen, V.: Diversification of System Calls in Linux Binaries. In: Proceedings of the 6th International Conference on Trustworthy Systems (InTrust 2014) (2014)
24. Rauti, S., Teuhola, J., Leppänen, V.: Diversifying SQL to Prevent Injection Attacks. In: Proceedings of Trustcom/BigDataSE/ISPA. pp. 344–351 (2015)
25. Rauti, S., Lauren, S., Uitto, J., Hosseinzadeh, S., Ruohonen, J., Hyrynsalmi, S., Leppänen, V.: A survey on internal interfaces used by exploits and implications on interface diversification. In: Brumley, B.B., Röning, J. (eds.) Secure IT Systems. pp. 152–168. Springer International Publishing, Cham (2016)
26. Shetty, R., Choo, K.K.R., Kaufman, R.: Shellshock vulnerability exploitation and mitigation: A demonstration. In: Abawajy, J., Choo, K.K.R., Islam, R. (eds.) International Conference on Applications and Techniques in Cyber Security and Intelligence. pp. 338–350. Springer International Publishing, Cham (2018)
27. Sinha, K., Kemerlis, V.P., Sethumadhavan, S.: Reviving instruction set randomization. In: 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). pp. 21–28 (2017)
28. Uitto, J., Rauti, S., Mäkelä, J.M., Leppänen, V.: Preventing Malicious Attacks by Diversifying Linux Shell Commands. In: Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15). CEUR Workshop Proceedings 1525 (2015)
29. Wang, C., Lu, Z.: Cyber deception: Overview and the road ahead. IEEE Security & Privacy 16(2), 80–85 (2018)
30. Xu, H., Chapin, S.: Address-space layout randomization using code islands. J. Comput. Secur. 17(3), 331–362 (2009)