This is a self-archived – parallel published version of an original article. This version may differ from the original in pagination and typographic details. When using please cite the original.

This is a post-peer-review, pre-copyedit version of an article published in

Advances in Intelligent Systems and Computing

Rauti S. (2021) Interface Diversification as a Software Security Mechanism – Benefits and Challenges. In: Rocha Á., Ferrás C., López-López P.C., Guarda T. (eds) Information Technology and Systems. ICITS 2021. Advances in Intelligent Systems and Computing, vol 1330. Springer, Cham. https://doi.org/10.1007/978-3-030-68285-9_45

The final authenticated version is available online at

https://doi.org/10.1007/978-3-030-68285-9_45

Interface diversification as a software security mechanism – benefits and challenges

Sampsa Rauti[1]

University of Turku, Finland
sjprau@utu.fi

Abstract. Interface diversification is a proactive approach to combat malware. By uniquely diversifying critical interfaces on each computer, the malicious executable code can be rendered useless. This paper discusses the advantages and challenges of interface diversification as a software security mechanism in order the gauge its feasibility and also gives some ideas for practical implementations. An analysis of strengths and drawbacks related to this security scheme will hopefully facilitate its adoption in practical systems.

Keywords: Diversification, obfuscation, interfaces, software security

1 Introduction

The malware authors rely on the known interfaces in operating systems, libraries and languages. By exploiting the fact that these interfaces are similar in millions of computers, they can launch massive large-scale attacks. The main idea of interface diversification is to make interfaces used on each machine unique. After diversification has been applied, the adversary does not know the "language" used in the system anymore. Any malicious program is considered foreign code and it will malfunction in a diversified system, because it does not know how to use the secret, diversified interfaces.

The approach is particularly useful against automated large scale attacks and self-propagating malware. Malware cannot make use of the computer's services and is rendered useless. Interface diversification is a viable countermeasure when malicious instructions are injected into an existing process (injection attacks) but it also works when a piece of malware tries operate from an independent process that well-known interfaces of the system.

The contributions of this paper are as follows. Based on our practical experiences and literature on the topic, we discuss the benefits and challenges pertaining to interface diversification. In this sense, this paper can be seen as a review in which the key properties, advantages and drawbacks of the new proactive software security approach are highlighted in order to gauge its feasibility. Moreover, this work gives guidance and ideas for practical implementations of interface diversification schemes in different modern application areas. Therefore, the paper aims to discuss the most important characteristics of software diversification on a general level without any fixed execution environment.

The rest of the paper is organized as follows. In Section 2, we first present the general theoretical scheme used in interface diversification. The fundamental basic properties of the approach are then discussed in Section 3. Sections 4, 5 and 6 then move on to describe performance factors, implementation issues and the most important considerations related to deployment and usability of interface diversification. Section 7 concludes the paper.

2 Interface diversification

Internal interface diversification aims to reduce the amount of knowledge an adversary possesses about a particular execution environment [12]. In other words, diversification modifies applications' and operating systems' internal interfaces in order to make them difficult to predict for malware authors. Interface diversification often takes advantage of simple code obfuscation techniques like renaming, but more complex obfuscation techniques can also be used.

The term *interface* should be interpreted broadly here. In this context, the term does not only refer to traditional interfaces provided by software modules and libraries. For example, command sets of different languages [5] and memory addresses [14] should also been seen as diversifiable interfaces in order to prevent malicious programs from exploiting them. For our purposes, an interface is any collection of entry points that can be utilized by malware to abuse the critical resources of a computer.

A practical example of interface diversification is diversifying a shell language such as Bash in the Linux operating system. This diversification process follows the general idea illustrated in Figure 1. We use a secret diversification key to rename all tokens in the shell command language. This change naturally has to be performed both for the interpreter (execution environment) and for the scripts in the system (executable code). By using the secret unique key, the modified interpreter can execute diversified scripts. Now the trusted scripts are compatible with the system but malicious scripts or script fragments are not. Script execution can be halted when erroneous commands are issued by malicious code. Note that a unique key can (and usually should) be used for each separate script or program.

In a similar fashion, we can diversify the system call numbers of an operating system [6]. System calls are a mechanism that provides programs with an access to computer's essential resources through operating system's kernel [22]. For instance, in Linux we can replace a few hundred original system call numbers with new ones [17]. Secret diversification is then propagated to the trusted binaries in the system: the system call numbers have to be accordingly modified in all trusted libraries and programs that make use of system calls. Other examples of interface diversification include diversifying instructions in a machine language command set, altering keywords in the JavaScript language or renaming the SQL commands.

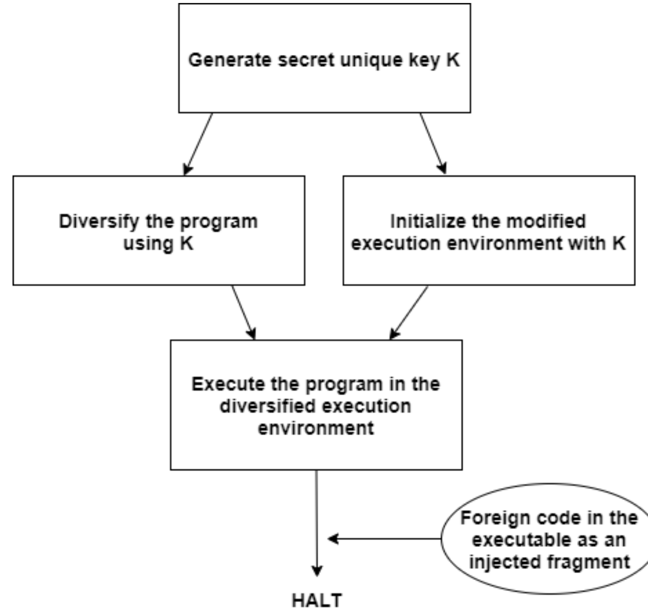

Fig. 1. The general idea of internal interface diversification.

3 Basic properties

This section presents some typical characteristics of interface diversification. Most of these properties can also be seen as advantages. When applying diversification, the malicious code does not have to be known beforehand. Also, diversification does not exclude other security measures but works in combination with them. Diversification can also be utilized in various application areas and fares well against a multitude of attacks.

Proactiveness. Without a doubt, one of the greatest properties of interface diversification is that it is a proactive countermeasure. The exact threat that is being defended against does not have to be known beforehand like e.g. in fingerprint-based malware detection approaches. This is because diversification does not aim to detect the malicious program. Instead, malware is allowed into the system but is not able to consume any resources or use services.

More and more malware is continuously being churned out. According to AVTest [4], hundreds of thousands of new pieces of malware are released into the wild every day. It is therefore hard for traditional malware prevention approaches to keep up with the current development. Novel proactive approaches are needed. This still does not mean the traditional defence mechanisms are useless. They can be used together with interface diversification and other proactive countermeasures. This brings us to the next desirable property of diversification:

orthogonality.

Orthogonality. In many cases, interface diversification can be used in combination with other security measures. For example, if we want to first diversify the executable code and then encrypt it (for example, in order to safely store it when it is not being executed), this is perfectly possible. There is also nothing preventing antivirus programs from functioning in a diversified system, as long as the antivirus program itself is compatible with the diversified system. An antivirus program could still check downloaded programs or updates when they arrive into the system – and in some scenarios, it could even cooperate with the diversification engine.

Another example could be integrating an anomaly detection system with a diversification scheme. This is very possible by employing "fake original" interfaces [16]; for instance, when we diversify the mapping of system call numbers, we can still leave the old original system call interface there as a bait. With this dual interface approach, we can detect and log all the processes that invoke the original system calls – this behaviour is always suspicious in the schemes in which the whole system should be completely diversified and the trusted programs only use the diversified interfaces!

Wide applicability. As we already saw in Section 2, interface diversification can be applied at several software layers and in multiple different execution environments. It can therefore be seen as a global scheme that covers all the important software layers and key interfaces of a computer system (see also [15]). What makes interface diversification unique as a security mechanism is the fact that it is easily applicable in so many different application areas. Interface diversification may not always be as perfect and effective as some environment specific security measures, but the idea can readily be applied almost anywhere.

Another issue that has to do with wide applicability of interface diversification is its capability to prevent or mitigate a broad range of different attacks. In many contexts, interface diversification (or instruction set randomization) has been advocated as a security measure against injection attacks specifically [5]. However, we believe its application area is even wider in this sense as well: it is a good method not only against injection attacks but also against the cases in which a malicious binary executable has somehow been slipped into the system. Of course, this is not to say that interface diversification is a silver bullet that works against any exploit. For example, many software design flaws and logic bugs are beyond the help of diversification.

Additional security. Interface diversification makes many insecure systems more secure. For example, in the pieces of software for Internet of Things devices, security and privacy have not yet received the attention they truly deserve [3]. These devices also often do not receive regular security updates. To relieve this problem, interface diversification can patch up the security of insecure em-

bedded systems at least to some extent.

Local and distributed diversification. Depending on the execution environment which the diversification is being applied to, diversification can be either local or distributed. For instance, when we modify the system call numbers, this only changes things on one computer. On the other hand, when we diversify the language interface of JavaScript on some web page, the client machines have to be able to interpret this new language. One way is to share the diversification key to the client in the HTTP header [2]. However, the adversary might be able to learn the key in a simple scheme like this. Such distributed and public diversification schemes are therefore quite challenging. Note that diversifying the language interface of SQL, although often distributed, is not public in the same sense, because the client programs that require the knowledge about the diversified language are usually known beforehand. Therefore, not all distributed schemes are equally challenging.

4 Performance

Performance is an important factor in any system. Together with security and resilience, it is also one of the most important quality attributes pursued in diversification schemes. The performance losses incurred as a result of interface diversification are quite modest in most cases [11, 18], but performance depends on many design decisions made when outlining a diversification scheme. We will delve into these issues in this section.

Runtime performance. Usually, interface diversification does not have significant effects on runtime performance. For example, simply renaming library functions or changing the mapping of system call numbers has no effect on execution time whatsoever [17]. Then again, if we want a more flexible scheme where the system call numbers can be different for separate processes, the diversified system calls have to be decoded at runtime, which leads to somewhat degraded performance [13].

An important implication of modest performance loss is good energy efficiency. In the era of Internet of Things devices, tablet computers and smartphones, and given the importance of cloud computing today, this is an important consideration in any diversification scheme [9]. Running antivirus programs on energy efficient devices and taking up a huge proportion of available processor cycles and energy for this is often not an option. On energy efficient devices, interface diversification is therefore a viable choice for a security scheme.

Time taken to diversify. The time loss caused by diversification can also occur before runtime, if the diversification takes place before the program's loading and execution phases. Static diversification that does not change at runtime falls into this category. The only waiting time is when the binaries or the program code are being diversified, so there is no additional delay at runtime. Still, if the

whole system is often re-diversified, time spent to diversify it becomes important, even if the diversification does not happen at runtime. For example, diversification performed on each system start-up would cause an additional delay.

Dynamic diversification. One important idea related to diversification is that it does not need to be static but the diversification of a program can continuously change [7, 19]. This improves security, but degrades performance and possibly even makes the whole system unavailable for some time.

While a complete system probably cannot be re-diversified continuously, it might be good to run the diversification process regularly. To beef up security and retain good performance, some parts of the system can have quicker diversification cycles than others; for example certain critical files and libraries can be re-diversified more often. The frequency of re-diversification also depends on the inconvenience caused to the users of the system. However, as the cloud-based environments become more popular, changing to a differently diversified copy of the same system should be easy by utilizing several virtual machines. Uniquely diversified copies of the same system could be readily available.

5 Other implementation issues

Along with performance, there are other important key considerations when designing a diversification framework. These include the question of key storage model and placement of the diversification engine.

Key management and the diversification engine. A diversification tool and a secret key are essential parts of any diversification scheme, but where should they be stored? The simplest option is to have the diversification engine running as a local process (probably in the kernel space so that the user space processes cannot disrupt it) and also store the key locally (e.g. using the in-kernel key management utilities available in Linux). If the whole system is diversified, this should be relatively safe, since malware cannot get access to the key storage easily.

An app store on a dedicated server is likely to provide even more security, as the keys and the diversification engine would not be needed locally at all and could not be compromised that easily. Note that diversification keys cannot be deduced from diversified interfaces; the key is not a part of the diversified program code or scripts. That being said, regular re-diversification is still a good idea. In general, the app store based approach seems like an enticing idea, especially for closed ecosystems (such as Apple's ecosystem and app store).

Algorithms and token space. What kind of function or algorithm should be used for diversification? There are several choices in the literature. The XOR function has been utilized in many schemes because of its simplicity and effectiveness [5]. Again, this is one trade-off point between performance and security.

Other simple and popular diversification functions are appending something to the end of original tokens [15] and using simple hash functions to create diversified tokens. Of course, sometimes the keyword space is fixed (like in the case of 32-bit system call numbers in Linux). In many cases, the token length can be changed, but excessively long diversified tokens result to the loss of disk space which can be a real issue in embedded devices. Somewhat more complex diversification functions that depend on the location and order of the (in addition to the original token itself) are recommended, as we shall see next.

Granularity of diversification. An issue strongly related with the diversification function is the granularity of diversification. In other words, do we use the same diversification key for the whole system (e.g. for all the applications and script files) or do we provide a key specific to each application? The latter increases security of the scheme, but leads to a larger performance loss and causes issues with shared libraries.

Inside one file (or application), we can still continue diversifying parts of the code with different keys. For example, the line number or previous tokens could affect the diversification of a specific token in the code. For example, the token SELECT that is a first token in an SQL query might be diversified differently as the exact same token occurring later in the same query [20]. Obviously, finer granularity leads to a more complex diversification process. However, it also makes it more challenging for the adversary to inject his or her own malicious code fragments into the diversified code.

The extent of diversification. The target of diversification does not always need to be a complete system. Instead, we can only diversify some of the programs. Proxy-based diversification schemes (such as [13]) allow some applications to continue functioning without being diversified. We have usually advocated diversification of the whole system because of better security provided by this scheme [17]. After all, the programs outside of diversification can probably somehow be compromised and an adversary may be able to circumvent a diversification proxy. However, an advantage of proxy-based scheme is that the programs that are allowed to function normally without diversification suffer no performance penalties.

A completely different question related to the extent of diversification is what parts of an interface or an instruction set should be diversified. For example, leaving some less important keywords of an interpreted script language (such as Bash) undiversified might provide better readability of scripts while still securing the language by partial diversification. Usually, though, diversifying the complete set of commands is the best bet when we require decent security.

6 Usability and deployment issues

While diversification is an effective and promising technique to counter malware, it still comes with several challenges related to usability and deployment. These

issues include enforcing transparency to users and developers, handling shared libraries and managing updates received by the system.

Transparency to users. A fundamental goal in any diversification scheme is that uniquely diversified copies of the same program should still behave in an identical way after diversification [8, 10]. Only some modest performance loss is acceptable. Other than that, the user should not notice anything out of ordinary when using a diversified program.

However, this is not the reality in all practical schemes. From time to time, the diversified interfaces pose some challenges for users. If a user wants to write a shell script in a diversified system, we would probably have to provide the user with an interface that allows writing scripts in their undiversified original form so that he or she could easily accomplish this task. The problem is that this user interface could also become a vulnerable spot targeted by malware. Most users, however, do not need to fiddle with internal interfaces that are targets of diversification. Also, the configurations of many systems (such as some embedded systems with limited set of applications and functionality) are quite stable and rarely need this kind of changes. Therefore, the applicability of diversification also strongly depends on the needs of the user and how the system is used.

Transparency to developers. Ideally, diversification does not affect the software development process. This is often the case; we have shown previously that the vast majority of binary executables [17] and interpretable scripts [15] are quite easy to diversify automatically without human intervention after the code has been written. The developer usually does not need to be concerned with diversification.

However, there are some cases that may require the developer to step in. For example, when a PHP script dynamically generates an SQL query at runtime, it is quite hard to statically diversify the query in the source code with an automatic tool. In these cases, the programmer can perform the diversification with the help of a tool that diversifies the required tokens for him or her. This makes the adoption of diversification easier and more pleasant. Dynamically created Bash scripts in Linux constitute a similar problem. In this case, though, we can argue that this kind of programming practice (executables creating shell scripts dynamically) should be avoided anyway and the programs could be rewritten not to use dynamically generated scripts.

Shared libraries. When each (dynamically linked) executable or shell script has its own unique diversification key, shared libraries become a problem. One solution is to create new versions of these libraries for each executable (time and space is lost but better security is gained). This solution is probably better than e.g. using undiversified libraries. Libraries can be diversified for each program or script before or during execution. In any case, diversification of shared libraries only has to be performed once (for each re-diversification cycle).

Updates. Software updates are a challenge for diversified systems. Each arriving patch has to be accordingly diversified in order to be compatible with the respective software and if the whole program is updated, it has to be compatible with the interfaces it depends on. Therefore, the same update issued by a software vendor does not readily fit the diversified systems, but has to be diversified either on an app store server or on the user's machine, depending on where the diversification engine (and the secret key used for diversification) is located. Of course, an arrival of a large update could be a good opportunity to re-diversify the whole system.

7 Conclusions

We have covered several considerations that developers and users of any interface diversification scheme have to take into account. As a malware prevention approach, interface diversification poses some challenges, but still shows a lot of promise because of several desirable properties such as proactiveness and low performance penalties. Diversifying internal interfaces of the system is an especially fitting protection mechanism for securing growing number of Internet of Things devices, which often have poor security and require security mechanisms that are not resource intensive (e.g. in terms of computational power).

The future work in the area involves implementing and experimenting with practical diversification schemes. More work is also needed to solve challenges of interface diversification. For instance, improving transparency to developers and solutions handle the updates are important topics. Finally, diversification has lots of potential when used in combination with other security measures such as anomaly detection systems [1] and machine learning [21]. These possibilities should be further explored and tested with proof-of-concept implementations.

We believe diversification should be seen as a global scheme covering all important layers of the system. This way, its effectiveness as a countermeasure against malware can be maximized. At the same time, orthogonal use with other security measures is still needed. After all, the adversary's imagination can always surpass our expectations: they may exploit implementation details that nobody has thought to diversify.

References

1. Ahde, H., Rauti, S., Leppanen, V.: A survey on the use of data points in ids research. In: International Conference on Soft Computing and Pattern Recognition, Springer (2018) 329–337
2. Athanasopoulos, E., Krithinakis, A., Markatos, E.P.: An architecture for enforcing javascript randomization in web2.0 applications. In: International Conference on Information Security, Springer (2010) 203–209
3. Atlam, H.F., Alenezi, A., Alassafi, M.O., Alshdadi, A.A., Wills, G.B.: Security, cybercrime and digital forensics for iot. In: Principles of Internet of Things (IoT) Ecosystem: Insight Paradigm. Springer (2020) 551–577

4. AVTest: Malware statistics. https://www.av-test.org/en/statistics/malware/ Accessed: 2020-13-09.
5. Boyd, S.W., Kc, G.S., Locasto, M.E., Keromytis, A.D., Prevelakis, V.: On the general applicability of instruction-set randomization. IEEE Transactions on Dependable and Secure Computing **7**(3) (2008) 255–270
6. Chew, M., Song, D.: Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, Pittsburgh, USA (2002)
7. Collberg, C., Martin, S., Myers, J., Nagra, J.: Distributed application tamper detection via continuous software updates. In: Proceedings of the 28th Annual Computer Security Applications Conference. (2012) 319–328
8. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations (1997)
9. Hosseinzadeh, S., Rauti, S., Hyrynsalmi, S., Leppänen, V.: Security in the internet of things through obfuscation and diversification. In: 2015 International Conference on Computing, Communication and Security (ICCCS), IEEE (2015) 1–5
10. Hosseinzadeh, S., Rauti, S., Laurén, S., Mäkelä, J.M., Holvitie, J., Hyrynsalmi, S., Leppänen, V.: Diversification and obfuscation techniques for software security: A systematic literature review. Information and Software Technology **104** (2018) 72–93
11. Larsen, P., Brunthaler, S., Franz, M.: Security through diversity: Are we there yet? IEEE Security & Privacy **12**(2) (2013) 28–35
12. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: Sok: Automated software diversity. In: 2014 IEEE Symposium on Security and Privacy, IEEE (2014) 276–291
13. 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, NSWCTC. (2009) 584–587
14. Marco-Gisbert, H., Ripoll Ripoll, I.: Address space layout randomization next generation. Applied Sciences **9**(14) (2019) 2928
15. Portokalidis, G., Keromytis, A.D.: Global isr: Toward a comprehensive defense against unauthorized code execution. In: Moving target defense. Springer (2011) 49–76
16. Rauti, S.: Towards cyber attribution by deception. In: International Conference on Hybrid Intelligent Systems, Springer (2019) 419–428
17. Rauti, S., Laurén, S., Hosseinzadeh, S., Mäkelä, J.M., Hyrynsalmi, S., Leppänen, V.: Diversification of system calls in linux binaries. In: International Conference on Trusted Systems, Springer (2014) 15–35
18. Rauti, S., Laurén, S., Mäki, P., Uitto, J., Laato, S., Leppänen, V.: Internal interface diversification as a method against malware. Journal of Cyber Security Technology (2020) 1–26
19. Rauti, S., Leppänen, V.: Internal interface diversification with multiple fake interfaces. In: Proceedings of the 10th International Conference on Security of Information and Networks. (2017) 245–250
20. Rauti, S., Teuhola, J., Leppänen, V.: Diversifying sql to prevent injection attacks. In: 2015 IEEE Trustcom/BigDataSE/ISPA. Volume 1., IEEE (2015) 344–351
21. Shaukat, K., Luo, S., Varadharajan, V., Hameed, I.A., Chen, S., Liu, D., Li, J.: Performance comparison and current challenges of using machine learning techniques in cybersecurity. Energies **13**(10) (2020) 2509
22. Tanenbaum, A.S., Bos, H.: Modern operating systems. Pearson (2015)