Taylor & Francis
Taylor & Francis Group

Journal of Cyber Security Technology

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/tsec20

Internal interface diversification as a method against malware

Sampsa Rauti , Samuel Laurén , Petteri Mäki , Joni Uitto , Samuli Laato & Ville Leppänen

To cite this article: Sampsa Rauti , Samuel Laurén , Petteri Mäki , Joni Uitto , Samuli Laato & Ville Leppänen (2020): Internal interface diversification as a method against malware, Journal of Cyber Security Technology, DOI: 10.1080/23742917.2020.1813397

To link to this article: https://doi.org/10.1080/23742917.2020.1813397

Published online: 31 Aug 2020.

JOURNAL OF CYBER SECURITY TECHNOLOGY
https://doi.org/10.1080/23742917.2020.1813397

Taylor & Francis
Taylor & Francis Group

OPEN ACCESS Check for updates

Internal interface diversification as a method against malware

Sampsa Rauti, Samuel Laurén, Petteri Mäki, Joni Uitto, Samuli Laato and Ville Leppänen

Department of Future Technologies, University of Turku, Turku, Finland

ABSTRACT

Internal interface diversification is a proactive software security method that prevents malware from using the fundamental services provided by an operating system by uniquely diversifying internal interfaces and propagating the information only to trusted programs. There are three main internal interfaces in operating systems that have been diversified in previous studies: (1) system calls (2) library functions and (3) shell commands. Based on previous studies and our own work, we implemented diversification for all interfaces in order to test their suitability and feasibility for real-world use. All three solutions enhanced the multi-layer security of the testing environment with little to no cost on system performance. However, maintaining such diversification tools might be troublesome in large and complex systems where new software is frequently added and software versions are updated. Thus, the solutions would be ideal for IoT devices and other smaller systems which rarely require updating, as well as restricted and static systems and critical systems with high-security requirements.

ARTICLE HISTORY
Received 17 April 2020
Accepted 14 August 2020

KEYWORDS
Proactive software security; diversification; obfuscation

1. Introduction

According to AVTest, hundreds of thousands of new malicious programs are being discovered each day [1]. Once a vulnerability is found, attackers usually move in to exploit it faster than software vendors can create and distribute patches. A key observation is that many of the vulnerabilities are related to misuse of internal interfaces, e.g. all injection attacks try to exploit internal interfaces of the target system. To counter this problem, novel proactive measures are needed. Simultaneously industrial systems, homes and cars are all being increasingly digitalized and connected to the Internet. With the increase of IoT devices and critical infrastructure connected to cyberspace, and other critical solutions such as online banking already in place, cybersecurity has become of critical importance. As anti-virus software is struggling to keep up with development of new exploits, software security vendors are providing

CONTACT Sampsa Rauti ✉ sjprau@utu.fi Department of Future Technologies, 20014 University of Turku, Finland

multi-layered security measures to combat these issues [2,3]. One such measure which has recently received attention is internal interface diversification (IID) [4,5].

The reasoning behind IID is that typically malware attacks and malicious programs are familiar with the target system's internal interfaces. For example, a piece of malware knows what library function it should invoke or what kind of shell commands it should issue in order to take advantage of the operating system's resources. Instead of thinking how to separately prevent all types of exploits that enable execution of malicious code in the target system, IID uniquely diversifies the interfaces that malware uses to reach its goals. Knowledge of the new interface is then propagated to trusted programs and scripts in the system so they conform to the new ″language″ of the system. As malware does not know the diversification secret (e.g. a unique key used to diversify the interfaces), it cannot function as intended.

Several diversification schemes employing this general idea have been published in the literature [4]. In the Linux operating system, these schemes often propose diversification of three important interfaces: (1) the system call interface [6]; (2) binary symbols and library functions [7]; and (3) command shell language [8,9]. In this study, our aim is to investigate the feasibility of each solution for system's multi-layered security. To that end, we deploy solutions for each of the three interfaces, and run tests to determine in what kinds of systems and environments the solutions could be used.

The rest of the paper is structured as follows. Section 2 presents the general idea of interface diversification and provides some background of the three interfaces we are covering in this paper. This is followed by a description of the empirical research methodology. Sections 4, 5 and 6 discuss the experiments in the three main categories mentioned above. Section 7 then summarizes the findings and discusses their implications. Finally, before conclusions, we discuss other solutions in related work.

2. Multilayer interface diversification

Various software interfaces have been proposed for diversification in order to enhance system security. Applied methods range from binary level solutions to upper software levels such as source code or even higher levels of abstraction [5]. Cohen advocated diversification as a method for operating system protection already in 1993 [10]. He proposed utilizing several obfuscation techniques to create unique software instances to make it harder for malicious programs to function. However, his ideas did not catch wind until 2003 which was the first year in which over 10 peer-reviewed papers on diversification were published [4]. Another pioneer of diversification as a security solutions, Forrest [11], describes in 1997 diversified computer systems as a feasible countermeasure against malware. Some later studies, i.e [2,12]., and books [13,14], outline an idea of system-wide multilayer diversification. While earlier work focuses primarily on operating system level diversification, this idea has also been proposed to be used in web environments [15].

A few surveys on diversification research have been written. Larsen et al. [5] survey the state-of-the-art in automated software diversity as a mechanism to improve security and privacy. They argue for automating the diversification processes for ensuring practical use of such systems. Baudry et al. [16] conducted a literature review in 2015 that captures the big picture of software diversification: along with security, diversification can also be used for fault tolerance, reusability and testing. The survey referred to what we call diversification as *randomization at different system levels*. The most recent review conducted by Hosseinzadeh et al. [4] and published in 2018 reviews over 200 studies on diversification and obfuscation techniques to identify the goals of such solutions as well as all potential interfaces which can be diversified. This is also the most exhaustive work on the field. The majority of diversification solutions were found to be designed to work on the operating system level with some aimed at web interfaces. Surprisingly, only a few studies focused on IoT devices specifically. Three main interfaces in operating system diversification were: (1) the system call interface; (2) shared libraries and library functions; and (3) shell scripts.

2.1. Internal and external interfaces

In this study, the term *interface* is broadly defined to mean any entry point that enables malware or an adversary to access critical services of the operating system. Let us remember that operating system guards the use of all resources of the system. Therefore, an interface can refer to traditional interfaces such as the system call interface or operating system APIs, but also to higher level interfaces (based on lower level interfaces) such as memory or command shell languages that can be exploited by malware e.g. through buffer overflows.

In this study, we concentrate on *internal interfaces* that are not directly used by users. For example, if the system call interface is altered and changes are propagated to binaries in the system, user experience does not change. External interfaces, such as graphical user interfaces, are not diversified. Malware usually uses internal interfaces to reach its goals. Malware may execute in the context of the operating system either as its own process, or as a piece of malicious code injected to another process. In both scenarios, IID prevents the malware from accessing internal interfaces in the system.

Although developers of services or applications need to know the internal interfaces, the users do not need such knowledge. Malware creators should not have knowledge of internal interface details. It is important to note that the software developers can do software development against known standard internal interfaces – development should not be done against diversified internal interfaces.

2.2. The general idea of internal interface diversification

IID modifies applications' and libraries' internal structure so that an adversary or malware cannot predict their implementation details anymore. By diversifying the system's internal interfaces, we decrease the number of assumptions the adversary can make about the execution environment [17]. Importantly, this does not change the external interfaces exposed to the user, thus ideally maintaining the level of usability. IID can also utilize obfuscation methods such as renaming or altering the order of parameters in function signatures.

When applying IID, there are always two parts in the system that need to be diversified: first, the place where the interface is defined (e.g. system call number list in the kernel or the command set in a command shell interpreter) and second, the executable code that makes use of this definition (e.g. the code using the system call number or script file containing shell commands) [7,17]. After diversifying the interface itself, changes must be propagated to all trusted programs that are using it. Thus, IID retains the original functionality of the programs and does not affect the user experience [18]. Ideally the only noticeable change is a slight increase in execution time, but often there is no performance penalty at all, as demonstrated by, for example, the case of changing the system call numbers.

IID also has minimal impact on the work of a software developer working on user applications, as IID is applied to the programs or scripts after they have been written via an automatic tool. A developer needs to interfere with IID only in rare cases, e.g. if the coverage achievable by automatic tools is incomplete [6]. In an ideal case the IID process is automated to the extent that a developer does not need to even be aware of its existence. In principle, the same applies to updating software versions.

2.3. Covered interfaces and attack scenarios

Our scheme consists of three layers of interface diversification: diversifying the system calls, library functions leading to invocation of system calls and the command-line interpreter. All three are essential interfaces in an operating system, as shown in Figure 1. In order to use services of the operating system, user applications invoke system calls either directly or through wrapper functions provided by operating system libraries. It is also possible to access many critical system resources through opening a command shell. In what follows, we will give a more detailed description on diversification of these three layers.

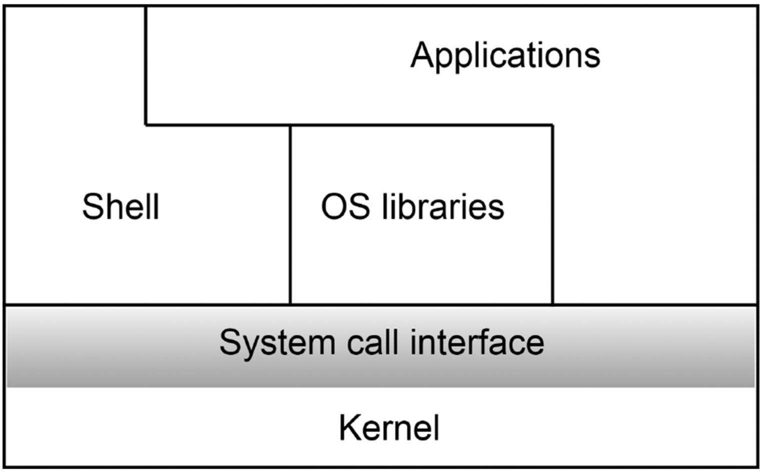

Figure 1. Software layers in an operating system.

2.3.1. System calls

The lowest layer of diversification in our scheme consists of uniquely changing the mapping of system call numbers [6]. The system call numbers in binaries of libraries and applications that invoke system calls are then diversified accordingly so that the trusted applications are compatible with the operating system.

As the attack code needs to execute system calls in order to cause any real harm in the system, diversifying system call numbers prevents it from doing this directly. For example, the code injection attacks tricking the system into executing code that contains direct system calls no longer work because the attacker does not know the system call numbers [17]. Other attacks where the attacker uses system call numbers directly will also fail.

2.3.2. Function names in binaries

It directly follows from the diversification of the system call interface that we also have to prevent the adversary from reaching the critical resources through those library functions that directly or indirectly lead to invocation of system calls (transitive closure) [7]. In order to propagate the diversification of library functions, we diversify the corresponding function names in binary files. In other words, the symbol strings in binaries are diversified. This diversification is performed both for libraries and applications invoking the library functions.

As an example attack scenario, failing to invoke a system call directly, the attacker might try to use the symbol table or PLT/GOT to find out a function’s address based on its name. They would then invoke a function that makes a specific system call. However, since the function names have been diversified, this approach will not work.

2.3.3. The command-line interpreter

To circumvent the protection for library functions and system calls, malware or an adversary could try to make use of interpreted languages like command shell script languages [8]. Therefore, we also include shell scripts in our protection scheme. Much like the library functions, the command shell provides access to the resources provided by the operating system.

This fact has been exploited in previous attacks, for example, injection attacks such as ShellShock [19]. In order to prevent the attackers from using the command line, we diversify the language interface used by the command interpreter, that is, change the set of tokens recognized by the command interpreter's lexical analyzer. To this end, the command interpreter is modified to support execution of diversified scripts and all the shell scripts in the system are diversified [8,9].

2.4. Interfaces not covered

It is important to note again that diversifying these three interfaces does not provide a totally comprehensive protection against malicious attacks. First of all, there are attacks that can not be prevented with diversification. These mostly have to do with faulty implementation details and logic bugs in the public interfaces of the system.

Second, there are some other interfaces, most notable those present in web environment, that can be used by the adversary to subvert the system. SQL and JavaScript language interfaces are such examples. Diversification of these interfaces has been discussed elsewhere [20,21].

Third, an attacker may find some way around the diversified interfaces, e.g. using a new interface that anyone has not thought to diversify yet. For example, blind hacking attacks, where the attacker simply guesses system call numbers and function names and tries to invoke them, are also threats our approach might not prevent. Still, the interfaces we propose for diversification here are used by a large number of malware exploits and diversifying them is therefore worthwhile [22].

Finally, Instruction Set Randomization (ISR) on machine code level is not included in our framework, because we believe a good protection can already be achieved by diversifying system calls and library functions. Additionally, ISR has many challenges like not being supported by current CPU architectures. We also do not discuss Address Space Layout Randomization (ASLR) because many papers have already been published on the topic [23,24] and the protection scheme is already incorporated into most modern operating systems [12,20]. This protection can be used in combination with our scheme.

3. Materials and methods

Linux-based operating systems, Linux From Scratch (LFS), Gentoo Linux Minimal Installation and Fedora Linux, were selected as environments for implementing the OS-level diversification techniques. The advantage of these systems is the availability of source code for the majority of applications and libraries, which is useful for understanding the underlying system and engineering diversification solutions. Linux-based operating systems are also world-wide the most popular in smartphones [25], supercomputers [26], IoT and embedded devices [26,27] and web-servers [26].

With regard to the three OS techniques, we provide a description of each solution with reference to the code of the implementation. The feasibility of each solution is tested with the aim of assessing the costs and benefits of adopting the solution for use. Accordingly, we test the impact of the solutions on (1) System performance, by measuring execution times in the system with and without the diversification in place; (2) System security, by empirically testing popular attack scenarios against the diversified system; and (3) Implementation and maintenance costs, by discussing deployment time, difficulty of implementation and updating of the diversification tools. The tests are carried out individually for each diversification technique and reported coupled with relevant discussion.

4. Experiments on system call diversification

In this section, we discuss and evaluate the first part of our diversification scheme, changing the mapping of system call numbers. We first explain the scheme and then cover some of the challenges that make automatic diversification of all the system calls difficult. We also present some experimental data on problematic system call invocations that are difficult to diversify with an automatic tool in a practical system. Finally, we discuss practical methods to cover the difficult cases.

4.1. Diversifying system calls in ELF binaries

In [28] we provided an implementation for a system call diversifier. Our diversification tool uses a straightforward linear-sweep algorithm [29] to rewrite the system calls in 64-bit ELF (Executable and Linkable Format used in Unix-based systems) binary executables. This disassembly method decodes everything in the ELF sections that are usually used to store machine code. Our diversification method is used on binary files after compiling, before they are deployed for execution.

Our diversifier tool finds system calls by linearly going through the program code sections in an ELF binary. A system call invocation consists of two separate phases: first, putting the system call number into a predefined register and second, transferring the control to operating system's system call handler. Our tool first looks for SYSCALL instructions that are used to invoke system calls in the x86-64 architecture. After finding a SYSCALL instruction, the tool starts looking for a system call number associated with this call. The instruction that sets the system call number can be found by backtracking from the location of SYSCALL instruction. The number of the system call to be invoked is first moved into a register, so our tool searches for instructions changing values of the registers used for this purpose (RAX, EAX, AX, AH, AL). The number of a specific system call is then rewritten according to the chosen diversification function (a transformation that maps the original system call numbers to the diversified ones).

4.2. Challenges

Because we used a simple linear-sweep based disassembly algorithm and static analysis of binaries, our approach has some limitations:

- *Gaps between instructions*. A system call invocation consists of two phases, which means there are two possible cases to be dealt with. Either the two instructions are consecutive or there are other instructions between them. The first case is mostly trivial but the second case may introduce some problems. For example, a jump instruction between the two phases is problematic for our algorithm. This might be caused by a conditional structure in the code.
- *Indirectly moving values to registers*. As we limit our analysis to simple mov instructions, some complications arise. For example, the number of the system call to be invoked can be moved to the final register indirectly using other registers as temporary storage. Tracing this kind of data flow would require a more advanced algorithm. Many compilers also circulate values through memory before they are moved to a specific register, which is a similar problem.
- *Manipulating the system call number before use*. The system call number may also be manipulated after it is moved to a register, say, by incrementing EAX register. A more advanced algorithm should also take these kinds of changes into account.
- *Alignment*. The way data is arranged and accessed affects the success rate of our tool. Because of the straightforward manner our tool disassembles the file, excessive data or zero bytes between instructions lead to a failure. Therefore, in the worst case, a system call can be erroneously found in a binary file. However, in practice, compilers rarely produce this kind of faulty executable code.
- *Compiler optimization and settings*. In our analysis, we noticed that using different compilers or even just different versions of the same compiler results in differences in binaries. These differences have an effect on the results of our tool. Compiler settings, such as optimization, also have an influence on the accuracy of our tool. According to our experiments,

optimization is strongly related to the number of gaps in the binaries; with no optimization at all there will be problematic gaps [6]. However, as we will see next, this only causes problems in a few cases.

4.3. Experiments on the problematic cases

We have seen that there are several challenges in applying our approach in practice. However, these problematic cases are often caused by the gaps between the two instructions. It is therefore interesting to see how many system call invocations actually have these gaps. To this end, we analyzed Linux Gentoo distribution (a fairly minimal installation with only a few packages other than the default/linux/amd64/13.0 profile). We found that of 807 system call invocations, 736 had no gaps. That is, *91.2 % of system calls had no gaps*. The remaining 71 invocations had gaps, but these were very small almost in all cases. Figure 2 shows the lengths of the gaps system call invocations have in Linux Gentoo.

Because of the small proportion of problematic cases, it is to be expected that our tool performs well. The tests carried out on Gentoo support this hypothesis. Table 1 shows the identified (and correctly diversified) system calls and unidentified system calls for binaries that contain direct system calls in Gentoo. In unidentified cases, the tool finds the SYSCALL instruction but cannot correctly identify the system call number associated with it. Note that there can be

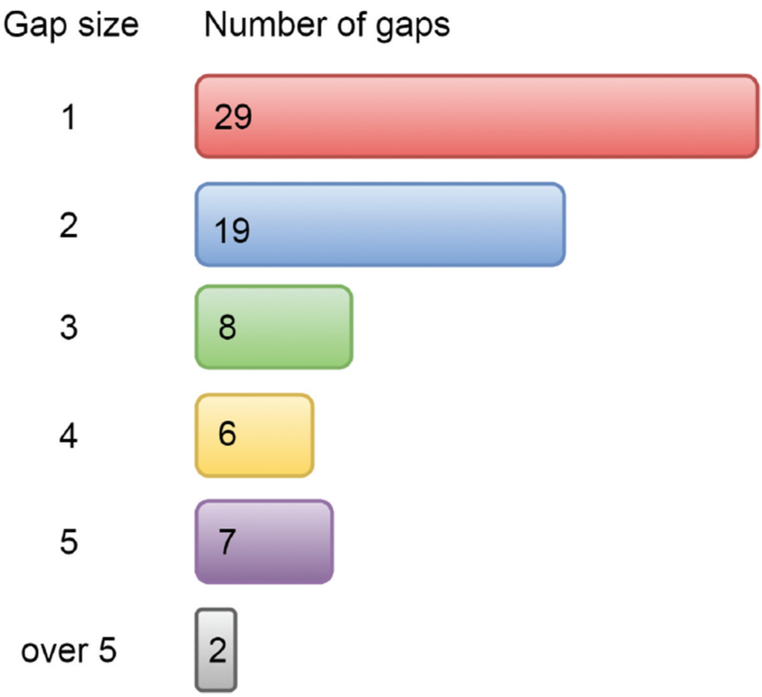

Figure 2. Gaps found in binaries of Gentoo distribution.

Table 1. System calls in binaries of Linux Gentoo.

Path	Not identified	Identified	Total
/lib64/libpthread-2.17.so	23	144	167
/lib64/libc-2.17.so	19	411	430
/lib64/ld-2.17.so	5	32	37
/lib64/libanl-2.17.so	5	1	6
/lib64/librt-2.17.so	5	24	29
/sbin/sln	5	84	89
/sbin/ldconfig	5	102	107
/lib64/libcrypt-2.17.so	0	3	3
/lib64/libnss_db-2.17.so	0	1	1
Total	67	802	869

several branches in a conditional statement that set the system call number, and all of these are counted as separate system call invocations (while each SYSCALL instruction was counted only once when finding gaps previously). We can see from the table that our tool performs well with this distribution – *92% of system calls were correctly identified*. In our view, this is an acceptable level of accuracy, as we will present methods to cover the remaining cases below.

Interestingly, only nine out of 569 binaries in Gentoo distribution had direct system calls. However, these executables are very important to the system, and are, for example, shared libraries that are invoked by almost all binaries. For example, the ones with most unidentified cases are libc, the C standard library and libpthread, the POSIX threading library. Still, as can be seen from Table 1, our tool performs well with almost all binaries in the system.

We consider the result of correctly identifying 92% of system calls very good in general. As the table shows, however, our tool did not perform well with the problematic cases caused by the gaps between system call invocation instructions. A more advanced algorithm with data-flow tracing can be developed. The next subsection presents several other methods to address the majority of remaining hard cases.

4.4. Diversifying the remaining system calls

In order to diversify the remaining hard cases that our algorithm did not identify and diversify correctly, a combination of the following methods can be used:

- *Including the diversification function in the binary*. The diversification function that performs the mapping between original and diversified system call numbers can be embedded in the binary. However, there are some challenges with this approach. Along with some relocation problems, the diversification secret, that would now be a part of binary, could potentially be leaked. If the malware were to find a way to get into the memory space of a running process, it could attempt to perform an analysis on the diversified system calls. This threat can be mitigated by applying some additional obfuscation to the binary.

- *Hard-coding the diversification*. In some of the code sections where system call diversification is troublesome, diversification could be hard-coded in the binary. That is, statically embed the diversified system call numbers into the binary. This is usually not a very flexible option, but could be done for some common parts of Linux operating system such as C standard library.
- *Rewriting source code*. Many problems that occur when diversifying binaries can be fixed on source code level. Writing the source code differently would often solve the problem, and this is also easily possible in open-source operating systems. Of course, this would mean lots of work if it were to be done to several user applications. However, as we have seen, mainly libraries used in many systems and many distributions use direct system calls. Many of these could be rewritten on the source code level to make the binary diversification process easier. This method is not complete, but, for example, clearly making the systems call numbers visible in the source code, so that the system call numbers are not determined as a result of an obscure calculation and do not come as a user input, the diversifier can do much better.
- *Changing the compiler settings*. We have seen that the order of instructions in machine is occasionally changed due to optimizations performed by compilers. For example, this happens when the system call numbers are circulated through memory or extra registers before invoking the system call. The binaries could be compiled using some specific compiler with certain configuration that would make things easy for the diversification tool. We cannot expect the software vendors to do this for us, but the approach fits for open source programs and libraries. Also, there could be a dedicated service for compiling programs with the right configuration.
- *Making the correct choice of the application area*. The methods discussed above all have some challenges. Still, we believe that by using these methods, it is possible to reach 100% diversification accuracy at least in many restricted systems. For instance, many embedded systems and Internet of Things devices are lightweight and easier to adapt to our scheme.

5. Experiments on symbol diversification

The second part of our scheme includes diversifying the function names in all binaries. This is achieved by changing the string symbols in ELF files. We first give a detailed introduction to this process and then present some challenges related to it. We also provide some experiments to gauge the usability of our scheme and finally present some methods to alleviate the identified challenges.

5.1. Diversifying symbols in programs and libraries

Using shared libraries, programs can implement part of their functionality by linking their code to functions, variables and other data provided by the library when the program starts or during execution. Dynamically linked libraries and programs (in ELF format) contain symbols for the functions and other data. The resources these symbolic names refer to are either provided for other executables or expected to be found from external binaries.

Our proof-of-concept implementation diversifies – that is, renames – the symbols in shared libraries. Also, these changes are propagated to all trusted ELF files which depend on the entities referenced by the diversified symbols. Naturally, the mapping between the original symbol names and new diversified symbols is kept secret. The adversary will have a hard time creating ELF binaries that make use of known function names and are therefore compatible with the system. A program will not function correctly unless the contents of the symbol table of the ELF file correspond to the names in the file providing the needed resources.

The proof-of-concept implementation of our symbol diversifier is made of three separate tools. Each tool takes care of one step of the symbol diversification process. By dividing diversification into these steps we aim to make the tool as flexible as possible in order to allow future changes and extensions. The three steps of our diversifier tool along with their respective inputs and outputs are shown in Figure 3. In what follows, we will describe these steps in more detail.

5.1.1. Symbol collector

The first step is collecting symbols. The *symbol collector* gathers symbols from 64-bit ELF files and gives a plain text list of them as a result. The process is carried out by iterating over the.dynsym symbol table that references the . dynstr string table that contains the symbol names. The tool also allows us to filter symbols based on their properties, such as the type of the symbol or whether the symbol is external.

5.1.2. Symbol diversifier

The *symbol diversifier* takes the symbol list produced by the symbol collector and diversifies each symbol using a specific diversification method. For example, our implementation uses salted SHA-256 hashing with Base32 encoding or

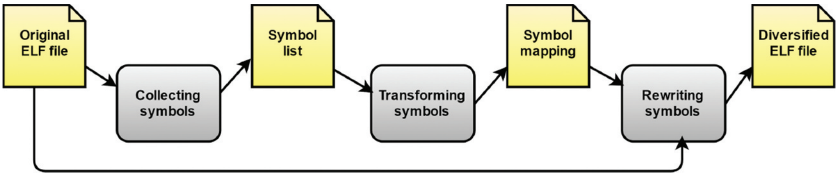

Figure 3. The symbol diversification process.

alternatively simple prefixing. This step does not involve the ELF files; it only needs the symbol lists. This step is therefore easy to replace or extend with any diversification method.

5.1.3. Symbol rewriter

The *symbol rewriter* takes the original ELF file and the diversified symbol definitions and generates a modified version of the file using this information. Editing ELF files involves updating several data structures in the object file, which makes symbol rewriting the most complex step in our scheme.

Symbol rewriter takes the mapping between original and diversified symbols and proceeds to rewrite the symbol definitions in the object file accordingly. To do this, we need to update many things in the ELF file: the hash tables that are created based on the symbol names, the string tables (.dynstr and .strtab) that contain the actual symbol names, data structures that point to offsets within, or other data regarding the string tables; .dynsym and .gnu.version sections, and the ELF section and segment header tables.

An apparent challenge is that rewriting the string table in-place is not possible in all cases; the diversified symbol names may require more space than the original ones! In our implementation, we chose a simple approach of moving the extended sections to the end of the ELF file. The gaps left in the original section locations are padded with NUL bytes.

For this paper, we have improved our original diversification tool [28] to better mitigate the known problems in the diversification process. The previous implementation had issues tracking dynamically what loaded libraries are used by the binary. An application can always try to dynamically load new dependencies at runtime, and these libraries are not easy to find while analyzing binary files statically. The improved version of our tool finds symbols for the dynamically loaded dependencies more accurately and diversifies these strings. The following experiments have been conducted with this new improved version of the symbol diversifier.

5.2. Experiments with the symbol diversification tool

Table 2 shows the applications we diversified using our symbol diversifier tool. The columns indicate whether diversification worked, whether the program started and whether there were any visible errors in its execution.

We can see that a majority of the diversified programs worked correctly in our tests. Command-line tools like cat, echo, less and interpreters such as gawk and Perl interpreter worked as expected. Most of the user applications with a graphical user interfaces such as inkscape and gedit also appear to work correctly.

Unlike in [7], gnumeric also works with our solution, which is evidence that we managed to improve our earlier solution with regards to diversification

Table 2. Results of symbol diversification tests with various Linux applications.

Application	Diversification works?	Starts?	Visible problems?
cat	yes	yes	no
echo	yes	yes	no
less	yes	yes	no
gawk	yes	yes	no
perl	yes	yes	no
python2	yes	yes	no
ghex	yes	yes	no
emacs	yes	yes	no
dia	yes	yes	no
inkscape	yes	yes	no
filezilla	yes	yes	no
pcmanfm	yes	yes	no
gnumeric	yes	yes	no
vim	yes	yes	no
gedit	yes	yes	no
gnome-calculator	yes	yes	no
charmap	yes	yes	no
gnome-terminal	yes	no	–
lxterminal	yes	yes	no
mplayer	yes	yes	no
cheese	yes	no	–
firefox	yes	yes	no

accuracy. The older diversifier still had some problems with diversification due to dynamic libraries, like icons not showing up correctly in diversified gnumeric. With the improved version of our diversifier, even Firefox, consisting of several million lines of code and containing several optimizations that make automatic diversification harder, was diversified correctly by our tool and appeared to function normally. Browsing web-pages such as Wikipedia and YouTube works as expected.

Although the experiments were not exhaustive in terms of input combinations or code coverage, they do provide backing for successful symbol diversification. More complete testing would of course be necessary for production systems. While we think these results are positive, there were two programs (gnome-terminal and cheese) that were not diversified correctly with our current diversifier implementation. This is due to library path dependencies that our tool could not track and diversify correctly.

However, we will see next that these remaining issues can be identified and methods to alleviate them can be developed.

5.3. Methods to diversify the remaining symbols

The problems with diversifying two applications seen above were caused by the use of dynamically loaded libraries. Applications often dynamically load new libraries during execution, and it is not totally straightforward to find these dependencies when analyzing binary files statically. Plug-in mechanisms often employ this kind of loading. In run-time linking, the symbol name sometimes cannot be determined by statically analyzing the binary file.

There are two main approaches to solve this problem. First, we could make modifications to the library in order to make it perform symbol diversification at the time the run-time loading is done. The symbol diversification could be done in the address space of the process (this has negative implications on security if the malware also operates in the same address space), or by utilizing an external diversification server. Second, we could statically search for function names stored in the binary (e.g. the .rodata section of an ELF file, when the function name is stored in the binary as a string). We would then alter these strings according to our diversification scheme.

These methods would modestly increase the execution time (that otherwise does not increase due to diversification), but they would also solve most of the issues that occurred in our experiments. The problem with dynamic libraries is mostly present in the user applications with many library dependencies and graphical user interfaces.

6. Experiments on diversification of the command shell language

In this section, we discuss the third part of our scheme, diversifying command shell languages. We first present our approach for shell language diversification. Some challenges and implications of applying this idea are then discussed. Finally, we show some methods that can be used to mitigate these challenges.

6.1. Diversifying shell languages

Diversification of shell languages can be achieved by changing the commands in the language's command set to ones the adversary does not know. The idea is the same as in the symbol diversification scheme in the previous section: the language interface is diversified so that the malware author can not make use of his or her knowledge about the language. Malware that is not armed with this knowledge will not be compatible with the rest of the system.

In the current approaches in literature, diversification of programming language interfaces is usually performed by appending a diversifying tag after each token in the language [30–32]. Our scheme is similar but utilizes a stronger form of diversification by using several unique tags. Another alternative would be to modify the language so that the tokens are entirely changed into a form where the original token is not visible at all. However, by using tags in diversification, readability is preserved and implementation for many shell languages is simplified.

In our diversification scheme, a secret key is used to create uniquely diversified scripts. Each token in the script is transformed by combining it with the key (any collision-free mapping can be used). Here, a *token* refers to a sequence of characters that the interpreter's lexical analyzer considers as a token identifier. The diversified scripts can only be run with a diversified interpreter that supports executing them. The interpreter needs the diversification key to execute the scripts. The scripts that are not correctly diversified cannot be executed. The malware author does not possess the secret key and cannot diversify any malicious code. Attempts to inject code or directly use the original script commands are therefore prevented.

Our approach also allows diversification to depend on the token's context. This option provides further security. For example, we can make the diversified version of a token depending on the other tokens present in the same script. Diversified form of a token can also depend on the source code before the token. This way, each token instance in the code will have a unique tag (two echo commands at different positions in the code will have different tags appended to them) which makes things even more difficult for the attacker as he or she cannot expect the same token to be diversified in the same way in different locations. In a way, this kind of diversification bears a close resemblance to encryption, but unlike encrypted code, diversified code can still be normally executed.

It is also worth noting that our approach uniquely diversifies only the tokens appearing in a specific script. The adversary has no way of finding out the diversified forms of any other tokens even in the case they have an access to the source code.

6.2. *Challenges and solutions*

Deploying a diversifying shell language interpreter in a practical environment entails more challenges than just replacing the original interpreter with a modified one. In what follows, we identify these challenges and present ways to mitigate them.

6.2.1. *Several interpreted languages*

One problem with diversifying shell scripts is that there are several interpreted script languages in most systems (such as sh, Python and Perl). All of these require a diversified interpreter. Each interpreter could of course be modified separately while still reusing some common functions such as the tag generation. A generalization of this approach would be to build a common library providing general tools to implement diversification for several interpreters. Because of the differences in interpreters' implementations, this is quite challenging. Finally, tools for building diversifying interpreters – interpreters that have the diversification functionality embedded from the beginning – could be implemented. This would be hard but would also make adding new diversified languages easy.

6.2.2. Secret key management

Because we use a unique key for each script, key management also becomes a problem we have to solve. In Portokalidis and Keromytis's solution for Perl [32], the diversification secret is input via command-line. While this idea may work in some restricted environments, we consider it an ad-hoc solution and it puts a lot of responsibility on the interpreter and the user.

Portokalidis and Keromytis also propose storing keys to a local database. In addition to a database, we also need a key management service. This results in a centralized secret key management system which aids with realizing system-wide re-diversification and diversification key granularity policies.

Key management is not a totally straightforward task. Implementation complexity depends on the chosen granularity of diversification keys (for example, one key for the whole system or file-specific keys). Finer granularity leads to increased complexity. Particularly, two scripts with unique keys both using the same library (which has its own key) cause a collision. The interpreter now has to handle two keys and has to know which one to use with each command.

6.2.3. General usability

Several interpreted languages are used from the command line. It is clear that diversification causes a usability issue in this case. It is not possible for the users to easily learn the diversified commands that they would need to make use the new language.

This problem could be solved by allowing the modified interpreter to execute non-diversified scripts from the command line. This solution, however, would defeat the purpose of script diversification. A malicious piece of code could exploit the command-line interface as an attack vector in order to compromise the system. If an attacker manages to run a malicious script fragment via the command-line interface, our diversification scheme is completely circumvented.

If a command-line interface with an option to input non-diversified commands is needed, it should therefore be implemented in such a way that it becomes very laborious for the malware to exploit this interface. For example, the internal structure of the interface (but not the commands of the script language) can be obfuscated to prevent malicious attacks.

Finally, it is noteworthy that many users do not need or are not aware of command-line tools at all. Several restricted environments also exist in which the command-line is used very rarely or is not needed at all.

6.2.4. Script invocations in source code

Finally, a big challenge with diversified scripts in the system are the script invocations made by programs. In other words, programs sometimes generate and execute script fragments at run-time. Source code of each program issuing script invocations needs to be modified to be compatible with the diversified shell language. Another option of course is to not support such usage of script engines. This issue is addressed next in our practical experiments.

6.3. A study on script invocations in source code

To see how much work script invocations in the source code would cause when diversifying the scripts in a system and whether the modification of code could be automated, we statically analyzed the source code of Linux from Scratch distribution (version 7.8) and identified all the script invocations present in the source code.

We divide the script invocations into three groups: inline scripts, command-line scripts and nontrivial cases. *Inline scripts* are scripts held entirely in the program's source code as plain-text. This makes them relatively easy to diversify automatically in most cases. A precompile process that takes care of diversifying these parts can be created. As can be seen in Table 3, our experiments show that majority (70.5%) of the script invocations are inline scripts and thus relatively easy to diversify. Table 3 also shows the proportions of different system calls (system, exec, popen) used to invoke the shell scripts.

The cases we have labeled as *command-line scripts*, the script that gets executed is supplied via the command-line to the software. The command-line may be external (program parameters) or internal (program offers a command-line during execution). Whichever the case, the program usually passes the script to an external interpreter. The interpreter needs to be provided with a secret key it can use to execute the diversified code. For example, either the user or the system can provide the key for the interpreter.

Finally, there are *nontrivial cases* in which the call-graphs and the flow of data are so deep that it is difficult to know exactly what is executed. This may be result of large program size, intricate wrapping of functionality or several other reasons. In these nontrivial cases, an automatic diversifier would usually not be able to diversify the scripts correctly and manual work would be required.

Because of these non-trivial cases, diversification of all scripts cannot be fully automated. For example, undecidability, data flow analysis precision and compiler limitations are ultimately reasons for this. Just like with system calls, setting some limitations for the diversified scripts would increase precision of the automatic diversifier by eliminating many hard cases. When the commands are clearly visible in the code and not result of some runtime calculation or

Table 3. Shell script invocations in LFS 7.8.

TYPE	SYSTEM	EXEC	POPEN	TOTAL	%
Inline scripts	99	11	38	148	70.5%
Command line scripts	17	14	5	36	17.1%
Nontrivial cases	6	9	11	26	12.4%

user input, diversification is much easier. The scripts the diversification scheme is applied to should therefore be restricted to those scripts that do not make use of reflection, that is, modify their own structure and behavior at runtime. Comprehensive testing of diversified scripts also alleviates the problem of non-trivial cases.

6.4. Experiments with the diversified interpreter

The diversified Bash interpreter was tested with 30 diversified scripts, some of which were written by us and some taken from our test environment. In our tests, all the diversified scripts worked correctly and produced expected results. In terms of performance, the diversified scripts were 6.1% slower than the original scripts on average, which we do not consider a significant performance overhead, as the overall impact on the system performance is likely to remain minimal.

Naturally, the overhead also greatly depends on the diversification function used. In this experiment, diversification was done using a hash of 6 characters (and a 3-character separator). The six characters were the six first characters of an SHA-1 digest based upon the semantic value of the diversified token.

7. Discussion

7.1. Towards practical multilayer interface diversification

In the previous sections, we discussed challenges present in the implementations of diversification of three interfaces: (1) system call; (2) library functions; and (3) shell scripts. Based on conducted tests and evaluation, we then proposed methods to alleviate the identified issues.

None of the three proposed diversification schemes imposed significant performance penalties, meaning implementing these solutions did not hurt usability by slowing the system down. For instance, a static rewrite of system call numbers only had a slight impact on performance in the rare case where manual intervention was required. Similarly, the library function symbol diversification did not impact performance, unless done on dynamically loaded libraries which caused a small increase in execution time. As demonstrated, diversifying shell languages may incur a larger increase in execution time compared to diversification of system calls or symbols. In the presented scheme, this is mainly because we allow several different diversification keys (for different scripts) and the modified interpreter has to support them all. It takes time to check the correctness of a specific diversified script. Also, the Bash interpreter is quite complex and is not well suited for diversification. Moreover, the proof-of-concept implementation could be further optimized. Still, we do not consider the overhead significant, as executing scripts usually consume only a minor share of available processor capacity. In summary, interface diversification does not have a significant negative impact on execution time. This finding aligns with results from previous studies [33]. The observed modest performance penalty ensures energy efficiency, which is important to consider especially for embedded and IoT devices.

Transparency is another major factor in applicability and usability of diversification solutions [5]. Diversifying interfaces is transparent in the sense that it does not change the user experience. The diversified version of the program works in the same way as the original program. In some cases, however, diversification causes inconvenience to the users. An example is the case when the command shell language has been diversified and a user wants to use the command shell. This at least requires the user to learn the new diversified commands, but in most cases it completely prevents writing scripts. From this standpoint, the system call interface and library function symbol diversification solutions are more transparent. With regards to software development, none of the three IID solutions should complicate the process. Software is developed in a standard way via referencing APIs and languages, and a uniquely diversified version is automatically generated for every execution environment in the deployment phase.

Finally, IID is a practical approach because it is orthogonal to many other security measures. For example, it can be used together with traditional security measures such as encryption and intrusion detection. This way, IID can present operating systems with additional security. This is especially useful for systems with high-security requirements such as governmental applications, banking servers, military applications and certain critical devices. Furthermore, sensitive devices such as IoT machines inside a private residence could be equipped with such protection. The reasoning is that as IoT devices might be quite rarely or never updated, their security could be enhanced as a countermeasure to the lack of security updates.

7.2. Limitations

Automation of diversification arose as a major challenge when implementing the three solutions. While the IID process can in most cases be performed automatically, there are some challenging cases that require attention from the programmer. For example, Section 4 shows that automatic system call diversification is not possible in, for example, cases where there are gaps between instructions. However, at least in the lightweight operating systems having less program code, the number of problematic cases is not large. Moreover, for the essential parts of the system such as popular libraries, manual fixes only need to be done once.

In a diversified operating system, updates are a challenge. Obviously, any update would have to be compatible with the interfaces it depends on.

Therefore, it has to be diversified accordingly. This can be done for instance by having the diversification engine automatically operating in the diversified system. Another option is to locate the diversification engine on a dedicated server and making it more difficult to discover and compromise for adversaries. Sharing updates through an app store, as suggested by previous studies [20], is also a possibility. The main reason to use external servers for storing the diversification engine or key is that there exists a risk that the key might somehow leak to adversaries if the system was compromised and inspected. Once an attacker obtains the key, diversification becomes useless. It is thus essential to store keys safely and use robust diversification schemes to ensure that they are not discoverable.

Another method for securing the IID is to make it dynamic, that is, by re-diversifying the interfaces in the system regularly. Different diversification cycles can also be applied to different interfaces in the system; the components considered critical to the system could be diversified more often than others.

One limitation that is particularly relevant in implementing IID is that some applications might be already obfuscated for additional security, which makes it more difficult to further diversify system call numbers and symbols in binaries. However, if obfuscation is applied to the binary after diversification, then it obviously does not cause any problems for IID. When considering using IID in a system, software developers could be advised not to use obfuscation on source code level to avoid the above-described issues.

Finally, it is worth considering that IID solutions are not perfect. For example, they do not prevent return-oriented programming (ROP)-based attacks, where the attacker makes use of carefully chosen machine instruction sequences that are already present in machine's memory. One example to protect from such attacks would be the G-free technique developed by Onarlioglu et al. [34] that transforms binaries so that they are protected against any possible form of ROP. IID can very well be implemented in a system together with the G-free technique, and they could both play their part in enhancing the systems multilayered security protection. Another way to counter ROP attacks would be to dynamically change diversification forming several possible execution paths for different diversified versions of an interface in the binary executables. This could mitigate also attacks where an adversary blindly tries to guess system call numbers. To prevent scenarios where the function names or system calls are not used directly (e.g. exploiting knowledge of the order of the PLT/GOT entries), the binary could be for obfuscated after diversification.

8. Related work and other solutions

Many authors have proposed renaming or randomizing symbols as a method of software diversification. Diversification of system call entry points by function renaming in order to defeat buffer overflow attacks was initially proposed by

Chew and Song in 2002 [17]. Along with system call entry points, they diversify the mapping of system calls in the kernel. They also propose diversifying the stack placement, which makes it more difficult for the adversary to find out the return address needed for executing injected code.

Jiang et al. [35] introduced a scheme called RandSys that combines Instruction Set Randomization (ISR) and Address Space Layout Randomization (ASLR) to build a robust protection against malware. RandSys utilizes a dynamic load-time scheme that allows each process to have uniquely diversified system calls at the cost of an increase in execution time. The scheme includes diversifying libraries (e.g. renaming functions uniquely for each executing process) and diversifying the system calls. In a similar fashion, Liang et al. [36] discuss diversifying the system calls in order to defeat malicious attacks. Our multilayer diversification scheme introduced and implemented in this paper can be seen as an extension to this previous work. In addition, we provided a technical description, an implementation as well as empirical results of testing done on the solution to solidify its feasibility as a malware countermeasure.

Diversification solutions have been created for the Windows operating system as well. For example, Abrath et al. [37] obfuscated the interfaces between application binaries and dynamically linked libraries in Windows. They noted that by statically linking the libraries into the program and obfuscating the whole resulting binary makes reverse-engineering considerably harder and shrinks the attack surface.

When it comes to diversifying script languages, Portokalidis et al. proposed an approach closest to our work [32] with the Bash interpreter. They modified the Perl interpreter so that it executes diversified scripts. Execution of malicious Perl code will fail because it is not correctly diversified, which makes it incompatible with the system. Rauti et al. continue this research by diversifying the Bash shell language and strengthen the original scheme by uniquely diversifying individual script.

Boyd and Keromytisproposed a similar approach for the SQL language [38]. They adopted a proxy-based approach in which an intermediate proxy component transforms diversified queries into queries conforming to original SQL language, passing them on to the database. Rauti et al. improved upon this scheme by making the diversification application specific and more secure, and presented a proof-of-concept implementation for SQL diversification in [21].

Generally, diversifying internal interfaces, or instruction sets [39,40], can be considered an effective approach for preventing code injection attacks. When applying diversification on source code or byte-code level, identifier renaming is also regularly used to disrupt the decompiling process. The resulting code will be filled with syntax or semantic errors and reverse-engineering efforts will be thwarted [41].

9. Conclusions

In this paper, we discussed diversification as a method that prevents malware from using the important resources of the operating system. We gauged the applicability IID by presenting experiments with three interface diversification schemes that can be seen as one joint approach to protect the system: (1) diversifying system calls; (2) library functions; and (3) shell commands. We showed that diversification can be done automatically to a great extent. We presented several challenges in interface diversification and proposed methods to alleviate them. These methods can be used to build practical tools that boost the multi-layer cybersecurity of systems.

As outlined in this paper, we see diversification as a comprehensive security mechanism that encompasses all the important interfaces and software layers of operating systems. Internal interface diversification methods show a lot of promise in systems with relatively small amounts of code and infrequent updates such as IoT devices. Furthermore, it can boost the multi-layered security of systems with high-security requirements. We hope to see practical diversification solutions being applied to embedded systems with lightweight operating systems (IoT, industrial Internet) in the near future.

Acknowledgments

The authors gratefully acknowledge the support of The Scientific Advisory Board for Defence (MATINE).

Disclosure statement

No potential conflict of interest was reported by the authors.

Notes on contributors

Sampsa Rauti is a University Teacher currently working at the University of Turku, Finland. His research interests include proactive software security (for example interface diversification, honeypots and fake service generation), software architectures and location-based games. He has over 50 peer-reviewed publications.

Samuel Laurén is currently working as a Senior Developer at F-Secure Corporation. At the University of Turku, he has developed several diversification tools. He has 15 peer-reviewed publications on proactive software security.

Petteri Mäki is a project researcher at the University of Turku. He has been involved in developing diversification tools, and has several publications on software diversification.

Joni Uitto is a Master of Science in Technology and currently works in software industry. He has been involved in diversification tool development as a research assistant, and has several publications on proactive software security, including diversification and honeypots.

Samuli Laato is a project researcher and a PhD student at the University of Turku, Finland. His research interests include educational technologies, human-computer interaction, location-based games, learning, music and math education. His research has been published in, for example, European Journal of Information Systems, Technological Forecasting & Social Change and Telematics & Informatics.

Ville Leppänen received the Ph.D. degree in computer science from the University of Turku, Turku, Finland, in 1996. He has been a Full Professor of software engineering and software security with the University of Turku, since 2012. He currently serves as the Head of Software Engineering and Leader of 5 research and development projects. He has over 230 international conference and journal publications. His current research interests include software engineering and security, ranging from software engineering methodologies, practices, and tools to security and quality issues, and to programming languages, parallelism, and architectural design topics. His security related research has focused on Internet of Things and cloud security, software-based diversification, vulnerability analyses, machine learning-based profiling for host intrusion detection systems, introspection mechanisms, and fake service generation.

References

[1] AVTest. Malware statistics. https://www.av-test.org/en/statistics/malware/.

[2] Alves-Foss J, Taylor C, Oman P. A multi-layered approach to security in high assurance systems. In *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*. Los Alamitos, CA: IEEE; 2004. p. 10.

[3] Hong JB, Kim DS. Towards scalable security analysis using multi-layered security models. J Network Comput Appl. 2016;75:156–168.

[4] Hosseinzadeh S, Rauti S, Laurén S, et al. Diversification and obfuscation techniques for software security: A systematic literature review. Inf Software Technol. 2018;104:72–93.

[5] Larsen P, Homescu A, Brunthaler S, et al. Sok: automated software diversity. *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP ’14, 276–291. Washington, DC, USA: IEEE Computer Society; 2014.

[6] Rauti S, Lauren S, Hosseinzadeh S, et al. Diversification of system calls in Linux Binaries. Proceedings of the 6th International Conference on Trustworthy Systems (InTrust 2014). IEEE; 2014. p. 255–271.

[7] Lauren S, Mäki P, Rauti S, et al. Symbol diversification of Linux Binaries. Proceedings of World Congress on Internet Security (WorldCIS-2014). Infonomics Society; 2014. p. 75–80.

[8] Uitto J, Rauti S, Mäkelä J-M, et al. Preventing malicious attacks by diversifying linux shell commands. *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST’15)*, CEUR Workshop Proceedings, 1525, CEUR; 2015, p. 206–220.

[9] Uitto J, Rauti S, Leppänen V. Practical implications and requirements of diversifying interpreted languages. Proceedings of the 11th Annual Cyber and Information Security Research Conference, Article No. 14. ACM; 2016.

[10] Cohen FB. Operating system protection through program evolution. Comput Secur. 1993 October;12(6):565–584.

[11] Forrest S, Somayaji A, Ackley D. Building diverse computer systems. In: *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS ’97. 1997. p. 67.

[12] Keromytis AD. Randomized instruction sets and runtime environments past research and future directions. IEEE Secur Privacy. 2009;7(1):18–25.

[13] Jajodia S, Ghosh AK, Swarup V, et al. Moving target defense, creating asymmetric uncertainty for cyber threats, advances in information security 54. New York: Springer; 2011.

[14] Jajodia S, Ghosh AK, Subrahmanian VS, et al. Moving target defense II, advances in information security 100. New York: Springer; 2013.

[15] Allier S, Barais O, Baudry B, et al. Multitier diversification in web-based software applications. IEEE Software. 2015;32(1):83–90.

[16] Baudry B, Monperrus M. The multiple facets of software diversity: recent developments in year 2000 and beyond. ACM Comput Surv. 2015 September;48(1):16 1–16:26.

[17] Chew M, Song D. Mitigating buffer overflows by operating system randomization. Technical report. Pittsburgh: Carnegie Mellon University; 2002.

[18] Collberg C, Thomborson C, Low. D. A taxonomy of obfuscating transformations. Technical report 148. department of computer science. The University of Auckland; 1997.

[19] National Vulnerability Database. Vulnerability summary for CVE-2014-6271. https://web.nvd.nist.gov/view/vuln/detail?vulnId= CVE-2014-6271.

[20] Larsen P, Brunthaler S, Franz M. Security through diversity: are we there yet? Secur Priv IEEE. 2014 Mar;12(2):28–35.

[21] Rauti S, Teuhola J, Leppänen V. Diversifying SQL to prevent injection attacks. In: In *Proceedings of Trustcom/BigDataSE/ISPA*. Helsinki: IEEE; 2015. p. 344–351.

[22] Rauti S, Lauren S, Uitto J, et al. A Survey on Internal Interfaces Used by Exploits and Implications on Interface Diversification. Cham: Springer International Publishing; 2016. p. 152–168.

[23] Shacham H, Page M, Pfaff B, et al. On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security. NY, USA: CCS '04; 2004. p. 298–307.

[24] Chongkyung K, Jinsuk J, Bookholt C, et al. Address space layout permutation (aslp): towards fine-grained randomization of commodity software. Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual; Dec 2006, p. 339–348.

[25] Iadarola G, Martinelli F, Mercaldo F, et al. Formal methods for android banking malware analysis and detection. 2019 Sixth International Conference on Internet of Things: systems, Management and Security (IOTSMS). IEEE; 2019. p. 331–336.

[26] Corbet J, Kroah-Hartman G. 2017 linux kernel development report. Publ Linux Found. 2017.

[27] Cozzi E, Graziano M, Fratantonio Y, et al. Understanding linux malware. *2018 IEEE Symposium on Security and Privacy (SP*); IEEE 2018, p. 161–175.

[28] Rauti S, Laurén S, Hosseinzadeh S, et al. Diversification of system calls in Linux Binaries. *Submitted to The 6th International Conference on Trustworthy Systems (InTrust 2014*); 2014.

[29] Schwarz B, Debray S, Andrews G. Disassembly of executable code revisited. In: In *Proceedings of Ninth Working Conference on Reverse Engineering*. 2002. p. 45–54.

[30] Athanasopoulos E, Krithinakis A, Markatos EP. An architecture for enforcing javascript randomization in web2.0 applications. In *Proceedings of the 13th International Conference on Information Security*: ISC'10, 203–209. Berlin, Heidelberg: Springer-Verlag; 2011.

[31] Boyd SW, Kc GS, Locasto ME, et al. On the General Applicability of Instruction-Set Randomization. IEEE Trans Dependable Secure Comput. 2008;7:3.

[32] Portokalidis G, Keromytis AD. 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. New York, NY: Springer; 2014. p. 469–480.

[33] Kc GS, Keromytis AD, Prevelakis V. Countering code-injection attacks with instruction-set randomization. *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03. New York, NY, USA; 2003. p. 272–280.

[34] Onarlioglu K, Bilge L, Lanzi A, et al. G-free: defeating return-oriented programming through gadget-less binaries. In: *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10. New York, NY, USA: ACM; 2010. p. 49–58.

[35] Jiang X, Wang HJ, Xu D, et al. RandSys: thwarting code injection attacks with system service interface randomization. In: *IEEE International Symposium on Reliable Distributed Systems*, SRDS 2007. 2007. p. 209–218.

[36] Liang Z, Liang B, Li. L A system call randomization based method for countering code injection attacks. International Conference on Networks Security, Wireless Communications and Trusted Computing. NSWCTC; 2009. p. 584–587.

[37] Abrath B, Coppens B, Volckaert S, et al. Obfuscating windows dlls. Proceedings of the 1st International Workshop on Software Protection, SPRO '15. Piscataway, NJ, USA: IEEE Press; 2015, p. 24–30.

[38] Boyd SW, Keromytis AD. SQLrand: preventing SQL injection attacks. Appl Crypto Network Secur. 2004;292–302. Lecture Notes in Computer Science 3089.

[39] Boyd SW, Kc GS, Locasto ME, et al. On the general applicability of instruction-set randomization. Depend Sec Comput IEEE Trans. 2010 July;7(3):255–270.

[40] Barrantes EG, Ackley DH, Forrest S, et al. Randomized instruction set emulation. ACM Trans Inf Syst Secur. 2005 February;8(1):3–40.

[41] Hunt J. Byte code protection. In: In *Java for Practitioners*, Practitioner Series. London: Springer; 1999. p. 427–429.