

# Diversification of System Calls in Linux Binaries

Sampsa Rauti<sup>(✉)</sup>, Samuel Laurén, Shohreh Hosseinzadeh,  
Jari-Matti Mäkelä, Sami Hyrynsalmi, and Ville Leppänen

University of Turku, 20014 Turku, Finland  
{sjprau, smrlau, shohos, jmjak, sthyry, villep}@utu.fi

**Abstract.** This paper studies the idea of using large-scale diversification to protect operating systems and make malware ineffective. The idea is to first diversify the system call interface on a specific computer so that it becomes very challenging for a piece of malware to access resources, and to combine this with the recursive diversification of system library routines indirectly invoking system calls. Because of this unique diversification (i.e. a unique mapping of system call numbers), a large group of computers would have the same functionality but differently diversified software layers and user applications. A malicious program now becomes incompatible with its environment. The basic flaw of operating system monoculture – the vulnerability of all software to the same attacks – would be fixed this way.

Specifically, we analyze the presence of system calls in the ELF binaries. We study the locations of system calls in the software layers of Linux and examine how many binaries in the whole system use system calls. Additionally, we discuss the different ways system calls are coded in ELF binaries and the challenges this causes for the diversification process. Also, we present a diversification tool and suggest several solutions to overcome the difficulties faced in system call diversification. The amount of problematic system calls is small, and our diversification tool manages to diversify the clear majority of system calls present in standard-like Linux configurations. For diversifying all the remaining system calls, we consider several possible approaches.

## 1 Introduction

Malicious software, or malware, is one of the main security challenges in today's information security. Malware uses knowledge about the identical interfaces of operating systems to achieve its goals. To access resources on a computer, a malicious program has to know the interface that provides the resources. Because of the prevailing operating system monoculture, an adversary can create a single malicious program that works for hundreds of millions of computers that use the same operating system.

The operation of malware would become considerably more difficult if it could not issue system calls and successfully use resources on a computer. Therefore,

---

This research has been funded by MATINE project 3301.

our approach is to make malware ineffective by using large-scale system call diversification. All software on a certain computer can be diversified so that it becomes very challenging for a malicious program to access resources. As a result of this, a large set of computers would have exactly the same functionality but differently diversified software layers and user applications. Because of the diversification of software layers, a piece of malware no more knows the “language” used in the system and becomes incompatible with its environment.

Even if a piece of malware would be able to find out how the resources are accessed on one computer, large-scale attacks are still very difficult, as malware knows the secret of applied diversification on one computer only. A costly analysis needs to be separately performed on each host. In other words, the diversification can be seen as a computer-specific secret. Our diversification scheme does not affect the work of a software developer because it is done on the binary level. Only some problematic cases, often found in libraries, may have to be dealt on source code level. The diversification of binaries does not change the user experience in any way, because the semantics and functionality of programs are preserved. Changing the system call numbers or mangled names of library routines does not affect performance either.

One part of this diversification process is to diversify the system calls that are used to access resources in an operating system. The idea of this paper is to study the diversification of system calls in Linux (see Sect. 2). More specifically, we discuss the challenge of recognizing and diversifying the direct system calls in Linux ELF (Executable and Linkable Format) binaries. We present a method for API diversification and describe a concrete tool used to achieve diversification. Based on the tests performed with this tool, we also present an experimental study of presence and distribution of system calls in Linux ELF binaries. We also discuss several possible solutions for the challenges faced when recognizing the system calls from binary files. By using our tool and these methods together, we believe 100 % accuracy in system call diversification can be achieved.

## 1.1 Our Goal

In this paper, our first goal is to characterize where the system calls are applied, what can be said about their distribution in the different software layers of an operating system that will be diversified. For example, how many binaries perform system calls and how often system calls are used in libraries and user applications. It is obvious that libraries perform the majority of system calls, but statistics of the role of direct system calls in application binaries is also important for considering schemes for securing a whole system. If applications often perform such calls, an automatic binary transformation tool seems necessary instead of just recompiling the diversified libraries from sources.

As the second goal, we want to find out how system calls are coded in ELF binaries. Understanding this is essential to successfully diversify all the system calls in binary files. We study the different ways system calls can be coded in ELF files. We chose to diversify applications and libraries on binary level rather than on source code level. This way, we do not need to have diversifiers for several

high-level programming languages. It is also easier to handle updates that arrive in binary form. Moreover, commercial applications or device drivers are usually not available in source code form.

The third goal is to present different ways to diversify the system calls in ELF binaries. Because system calls are not always presented in binaries in the same straightforward way, we may need to make use of several approaches to ensure that the system call diversification scheme is as perfect as possible.

## 1.2 Contributions and Structure of the Paper

To the best of our knowledge, our work is the first detailed study of static system call diversification in Linux ELF binaries. We also provide a concrete implementation of an automatic diversifier tool. This paper makes the idea of static system call number remapping, previously presented by Chew and Song [3], more concrete. We apply the system call diversification in practice to solve a more general problem of rendering malware useless. Unlike some earlier work on system call diversification (see for example [10, 13]), our tool performs the diversification statically after compilation and thus does not introduce any runtime performance loss. Compared to the earlier similar solutions, our approach is also more system-wide because it also diversifies the kernel. We implement a proof of concept diversifier and test our solution using two popular Linux distributions. We provide a detailed description of this tool and also present results on its accuracy.

One of the contributions of this paper is an empirical study of the presence of system calls in ELF binaries. We conclude that most system calls in two Linux distributions we tested are found in libraries like `libc`. Also, the vast majority of binaries do not contain direct system calls at all, which makes their diversification much easier. Another contribution of this paper is to present various solutions to diversify the differently coded system calls in ELF binaries. We believe that even though there are many challenges, diversifying 100 % of system calls is feasible by applying the different methods we consider in Sect. 6.

The rest of the paper is organized as follows. Section 2 presents an overview of our API diversification method for operating system protection. Section 3 covers some basic concepts needed to understand our diversification method and experimental tool in more concrete sense, like Linux layer structure and ELF binary files.

Section 4 discusses coding of system calls in ELF binaries and the challenges faced when trying to identify and diversify them. Section 5 presents our study of presence of system calls in the ELF binaries. We discuss our diversifier tool implementation and the experimental setting. We also present some results; where in the system are system calls located, and how many binaries contain direct system calls in the tested Linux distributions? Many possible solutions for these challenges and achieving 100 % diversification accuracy are covered in Sect. 6. Section 7 concludes the paper.

## 2 An Overview of Our API Diversification Scheme and Threat Scenarios

### 2.1 Our API Diversification Scheme

An operating system provides a variety of services that user application can utilize in a shared manner [25]. Therefore, the operating system and its system call interface can be thought as an abstraction between user applications and the services provided by hardware of a computer. System calls are a fundamental set of services in an operating system [23].

In order to interact with its environment, an application needs to use system calls. This can be achieved by either calling the system call interface directly or using libraries that provide wrappers for system calls. Therefore, in our view preventing a piece of malware from accessing the system call interface consists of two separate parts:

1. Diversify the system call numbers.
2. Diversify the system call implementations in the kernel and all the functions calling them directly or indirectly.

Diversification refers to meaning-preserving mapping in a programming language. That is, the program code is transformed to a different form, but its semantics are preserved so that the user of the program experiences no visible change when using the program. In this paper, system call diversification simply refers to changing the mapping of system call numbers.

The first part of our scheme means that we change the system call numbers defined in the operating system kernel. As a consequence, all code in libraries and user applications that call system calls directly using these numbers must be diversified accordingly as well, or they will stop working.

On the other hand, when a system call is not invoked directly, the call passes through several software layers before it reaches the system call implementation. In order to prevent a piece of malware from invoking system calls, we have to recursively diversify all the functions that make these system calls. We refer to the set of these functions directly or indirectly calling the system call implementations as the *transitive closure*. All these functions must be diversified (by changing their function signature, for example) to prevent a piece malware from using them to access a computer's resources. Trusted applications that are diversified correctly can still access the resources.

This paper deals with changing the system call numbers and diversifying all the direct system calls accordingly in user applications and libraries. Another part of our protection scheme, diversifying the transitive closure, has already been discussed in our other publication [17] and is not covered in this paper.

The trusted software layers and user applications are diversified with a secret diversification function which makes them compatible with new system call numbers defined in the kernel. As a result, the entry points that lead to the system calls are diversified in the whole system, preventing malware and any untrusted applications from using them to invoke system calls.

## 2.2 Threat Scenarios

Our solution is meant to protect computers from the malicious code that is either executed as its own process or as a part of another executing process. In the second case, the malicious code can observe the system calls its host program invokes and gradually learn the system call mappings used in the system. However, this would still require advanced analysis.

Without observing any program's actions, it is very difficult to guess correct system call numbers. Linux currently only uses around 320 system calls and their numbers are 32-bit, so there is a very high chance that our function can map calls so that malware cannot make valid system calls even by mistake. Of course, the mapping function should be designed so that it never maps an ID to itself. Illegal system calls could also be logged for further inspection. A program that randomly tries invoking large amount of system calls can be seen as suspicious.

It is important to note that in our approach we assume that a piece of malware has no access to the file system, for example, and has no way to analyze files or our secret diversification function that has been applied to binary files. File system access requires using system calls and thus an external malware does not have an easy access to the file system.

Our approach adopts a proactive view by preventing malware from harmfully interacting with its environment before its execution. As the number of malicious programs keeps growing and they keep transforming, traditional fingerprint-based antivirus software is becoming increasingly inefficient in the fight against malware [1]. Also, antivirus programs often only detect the threats they are already aware of. This is why complementary approaches are needed.

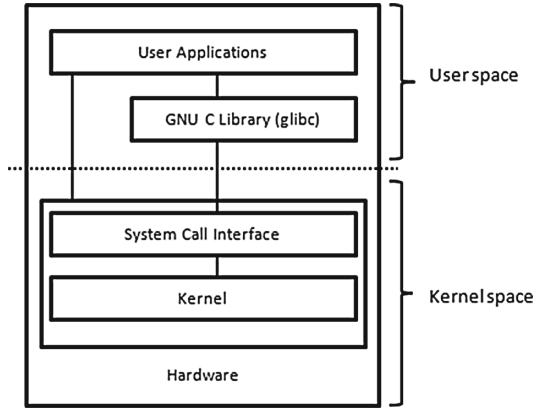
Together with diversifying the transitive closure, the system call number diversification should make it much harder for malicious programs from opening any resources on a computer. Untrusted programs do not know either the system call numbers nor the names of functions in other applications or libraries that lead to system calls.

## 3 Linux Layer Structure and ELF Binary Files

### 3.1 Linux Layer Structure

Linux system calls are implemented as named routines in the operating system kernel. In the user-space facing system call ABI, each call routine corresponds to a kernel-defined system call number. There are around 320 system calls in Linux, each with its own number [27].

The software layer structure of Linux is very roughly illustrated in Fig. 1. Linux contains wrappers in order to make it easier to issue system calls, which are implemented in different parts of kernel. However, as explained in Sect. 2, in this paper we are only interested in the cases where either libraries or user applications call the system call interface directly. We aim at recognizing and diversifying all these entry points in the binaries of the whole system.



**Fig. 1.** Linux layer structure.

The most important library making the system calls in Linux is `libc`, which most user applications use to access operating system services. However, several other libraries, many command line tools and even some web browsers also make direct system calls. The distribution of system calls will be examined closer in two test environments in Sect. 4.2.

### 3.2 Structure of ELF Files

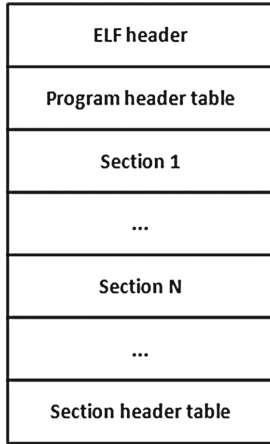
The Executable and Linkable Format (ELF) is a standard file format for executables, object code and shared libraries. It is used on many different platforms and in several operating systems. It is the standard binary file format for Unix-like systems.

Figure 2 shows the structure of an ELF file. An ELF header in the beginning of the ELF file describes the file's organization. The header also contains information on object file type and the instruction set architecture, for example.

Sections contain lots of miscellaneous information: instructions, data, symbol table, relocation information etc. Some sections are special, like sections for uninitialized and initialized data, sections holding debug information and comments, sections for read-only data and strings and sections for symbol tables [6].

A program header table is an optional part of ELF files. It tells the system how to create a process image and execute the program. Relocatable files do not need a process image. Information in a section header table describes the ELF file's sections. Each section has its own entry in the table. Every entry provides information such as the section name, the section size etc. Files used during linking always have a section header table, otherwise it is optional.

The sections in ELF files have a type attribute. For example, `PROGBITS`-type sections are reserved for program-specific data. This can be either executable code or other data. In order to diversify the system calls in ELF files, our diver-



**Fig. 2.** ELF file structure.

sifier tool analyzes the `PROGBITS`-type sections that are marked to be executable with `SHF_EXECINSTR` flag.

## 4 On Coding of System Calls in ELF Files

For the purposes of this paper, a system call can be seen consisting of two separate phases. As we have already seen, the first phase puts the value of the system call into a predefined register. The second phase transfers control to the operating system's system call handler. The exact mechanism for this is architecture and operating system dependent. On x86-64 Linux system calls are made using the `SYSCALL` instruction. For instance, the following commands are used to invoke `sys_write` (system call number 1):

```
mov $1, %eax
syscall
```

However, there are several factors that make identifying the system calls more difficult. For example, there might be a jump command between the two phases:

```
cmp $1, %eax
je equal
mov $0, %eax
jmp over
equal:
mov $1, %eax
over:
syscall
```

Here, system call 1 is invoked if the value in **EAX** register is 1, otherwise system call 0 is invoked. It is much harder for the analyzer to deduce what system call will be invoked.

Also, when the analysis is restricted to simple **mov** commands that directly move a value to a register where a system number is stored, many problems arise. For example, this leaves out a complicated setting where a value is moved to a register indirectly through other registers. Also, it seems compilers often write the binary so that the register value is first put to the memory and then into an appropriate register. These kind of indirect approaches are difficult to analyze without tracing the control flow of the program. There might also be other commands affecting the register values before the system call is made, say incrementing **EAX** register, for example.

Of course, it is interesting to ask whether these different ways to code the system calls in binaries have any visible reasons in the source code. It is pretty clear some of them do. For example, the jump we saw in the example earlier is probably created as a result of a conditional statement, like an if-statement, in the source code. It also seems that loop structures in the source code often create coding in the binary where the two phases the system call consists of are not consecutive. For example, we noticed that setting register values indirectly through other registers can be a result of a loop structure in the source code. Use of function pointers in the source code probably also affects the coding of the binary file.

On the other hand, all the binary codings of system calls do not seem to have clear explanations in the source code. For example, the fact that register values are sometimes circulated through memory seems very arbitrary and apparently associated with optimization made by the compiler. We noticed the version of compiler may greatly affect the coding of system calls in the binaries it creates. The compiler configuration and compiling environment probably also have an effect on this. Many compilers have switches that can be used to configure the level of optimization.

One more noteworthy problem is alignment, that is, the way data is arranged and accessed. Because our tool disassembles the binary file in a straightforward manner by processing it from the beginning to the end, any excessive data or empty space between instructions (zero bytes) lead to failure. There is no reliable way to detect when this failure takes place. In the worst case scenario, an erroneous system call could be found from a binary file. In practice, however, compilers usually should not produce this kind of faulty program code. Programs that somehow determine the system call at runtime based on the user input would naturally also be problematic, as it is impossible to identify the correct system call number in this case with static analysis. Some commercial products may also use different obfuscation methods in their binaries, which ironically makes our diversification task much harder. Moreover, different kinds of self-modifying programs are always difficult to handle for diversifiers.



## 5 Experimental Study on the Presence of System Calls in Linux Binaries

To find out how system calls are distributed between different parts and binaries in Linux distributions, we performed an experimental study on the presence of system calls in all binaries in tested systems.

### 5.1 Settings of Studied Linux Environments

We conducted our study using 64-bit Fedora and Gentoo Linux distributions. Fedora Linux was selected because it provided a full-fledged desktop environment with all the associated software out of the box. In contrast, the Gentoo installation we used was fairly minimal with only a few packages outside of the `default/linux/amd64/13.0` profile. Additionally, we conducted separate tests with C standard library implementation `glibc`. We concentrated on `glibc` specifically because it is one of the main libraries containing system calls. Characterization of test environment was deemed to be especially important since there are multiple factors that can cause results to vary considerably. When analyzing binary files, the most obvious source of differentiation is the compiler used to create the said binaries. Using a different compiler and even different version of the same compiler can lead to differences in produced binaries, which in turn might affect the results our system call analysis. Aside from the compiler version, the compiler settings used to produce the binaries play a central role. For example, we noticed that our analysis tool performed radically worse when the binaries were compiled with no optimizations at all. Because our tools performance is highly dependent upon how the register allocation is done, all compiler settings that might affect this can potentially alter our results.

Because we are analyzing all the installed applications and shared libraries, precisely describing our experimental setting would require us to list all the specific versions of the installed packages including potential distribution specific changes. Also we would have to record detailed information about the build environment, including compiler versions and settings.

Because of how compiler dependent our analysis is, conducting experimental studies using Linux distributions with binary based packaging, presents us with certain challenges. As we are using precompiled binaries we cannot know how they were produced, let alone control the specific compiler settings. This might make precisely replicating our results more complicated, but at the same time, it means that our test environment resembles a real-world test scenario more closely, since we assume that a typical end-user does not have control over how their binaries have been produced.

We used 64-bit Fedora Linux version 20 based installation (kernel version 3.14.4) as our test platform. We had also installed various other applications and libraries that were needed during the development of our analysis software. Of course, knowing only the release number of the distribution leaves out many details about the system, since the software has received various updates during

the release cycle. The Gentoo installation (kernel version 3.6.0) was considerably more minimal and contained relatively small number of packages. No desktop environment was installed for this system.

## 5.2 Distribution of System Calls in Binaries

We analyzed the direct system calls found in the binary files of two Linux distributions, Fedora and Gentoo. In addition to amount and distribution of system calls, we also wanted to see how well our diversification tool could identify the system calls in binaries.

In Fedora, 5649 binaries were analyzed. Only 18 of those contained any system calls. These binaries are shown in Table 1. For each binary, the amount of system calls successfully identified by our tool, the amount of unidentified calls and the total amount of system calls in that binary are shown. In this context, identifying refers to successfully recognizing the correct system call number. In unidentified cases, we find a `SYSCALL` command but cannot recognize a system call number associated with it.

**Table 1.** System calls found in binaries of Linux Fedora distribution.

Binary path	Identified	Not identified	Total calls
<i>/lib64/libunwind-x86_64.so.8.0.1</i>	1	0	1
<i>/lib64/libcrypt-2.18.so</i>	1	0	1
<i>/lib64/librt-2.18.so</i>	24	5	29
<i>/lib64/libc-2.18.so</i>	394	35	429
<i>/lib64/libanl-2.18.so</i>	3	3	6
<i>/lib64/libnss_db-2.18.so</i>	1	0	1
<i>/lib64/libgomp.so.1.0.0</i>	17	13	30
<i>/lib64/libaio.so.1.0.0</i>	5	0	5
<i>/lib64/libaio.so.1.0.1</i>	5	0	5
<i>/lib64/ld-2.18.so</i>	31	5	36
<i>/lib64/libunwind.so.8.0.1</i>	2	0	2
<i>/lib64/rtkaio/librtkaio-2.18.so</i>	45	14	59
<i>/lib64/xulrunner/crashreporter</i>	0	6	6
<i>/lib64/xulrunner/libxul.so</i>	3	56	59
<i>/lib64/firefox/crashreporter</i>	0	6	6
<i>/lib64/firefox/libxul.so</i>	3	56	59
<i>/sbin/ldconfig</i>	109	9	118
<i>/sbin/sln</i>	79	8	87
Total	<b>723</b>	<b>216</b>	<b>939</b>

We can see that large amount of system calls is located in `libc`, the C standard library. With this library, our diversifier performs well, recognizing over 90 % of calls. A few other libraries like `rtkaid` – a library used for asynchronous I/O – also contain direct system calls.

Some command line tools like `ld`, a dynamic linker, `ldconfig`, which is used to configure dynamic linker run-time bindings, and `sln`, symbolic link creator also seem to make quite many system calls. Our tool performs well with all of these binaries.

There are also a few problematic binaries, like Mozilla’s `libxul` library in this case. This binary encodes system calls in difficult ways, as it seems to favor using intermediate registers for passing values instead of direct assignments to appropriate registers to make a system call (see Sect. 4). Because of this problematic library, which appears in the system two times, our tool identifies about 70 % of the system calls in binaries in this system.

In the same way as with Fedora, binaries in Gentoo distribution were also analyzed. Only 9 of 569 binaries contained direct system calls. These binaries are shown in Table 2. There was no desktop environment installed in this system, which explains the smaller amount of binaries. Most system calls are in libraries, and about half of system calls are made in `libc`. Our tool performs well in this distribution, identifying 92 % of the system calls.

We can conclude from these results that even in rather large standard distributions, there are very few binaries with direct system calls. User application very rarely make direct system calls and use `libc` instead. This makes our diversification task easier. Especially in restricted environments with only a few user programs our diversifier would perform well.

However, even though our tool can recognize the system calls pretty well, there are still some problematic cases that were not identified correctly. We will look at some solutions to these problems in Sect. 6.

**Table 2.** System calls found in binaries of Linux Gentoo distribution.

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

### 5.3 A Closer Look at Diversification of System Calls in Libc

In the experiments, we also analyzed and diversified different versions of `libc` library. Because most of the system calls are located in standard libraries and not in the application's code (see Sect. 4.2), these libraries are a good target for analysis. User applications do not usually have any need to use the system call interface directly, and invoking the system calls indirectly using a standard library makes the application less dependent on certain operating system version by including an additional abstraction level.

We studied `glibc` version `686554bfff63dff0f8b20c84e9bdca45e643f9d9c`, which we compiled with `gcc` (GCC) 4.8.2 20131212 (Red Hat 4.8.2-7). This library was analyzed with our diversification tool both on Fedora 20 (64-bit) and on Gentoo. The results for both distributions are shown in Table 3.

**Table 3.** The system calls found in `libc`.

Distribution	Identified	Not identified	Total calls
<i>Fedora</i>	<i>380</i>	<i>35</i>	<i>415</i>
<i>Gentoo</i>	<i>398</i>	<i>18</i>	<i>416</i>

As, we can see, over 90 % of calls were successfully diversified in Fedora and over 95 % in Gentoo. Identifying the system calls succeeded well in our tests, because many routines in standard libraries are just simple wrappers for system calls. These routines simply take a set of parameters for system calls, put them into appropriate registers so that the system call can use them, and then invoke the system call with a predefined number.

However, some of the routines in `libc` include conditional execution of system calls. That is, these routines decide the system call to be invoked based on some external factor or invoke system calls as a part of a loop. These cases are of course more problematic. Additionally, standard libraries also usually contain routines that can be used to invoke an arbitrary system call. These routines take a system call number as their parameter, which makes it hard to rewrite them statically.

We also studied systems calls in `musl`, an implementation of C standard library. The version we used was `8a2d8719873a46d5cc5c54e688d47ea134c67c84`. This library was compiled with several different optimization settings, which demonstrates well the effects that optimization performed by the compiler may have on our diversifier tool. `musl` was tested on Fedora and the same compiler was used as in previous tests with `libc`.

`musl` was compiled using several different optimization options. Results are shown in Table 4. Switch `-O0` means no optimization. As shown in Table 4 results for the library with no optimization are really bad. However, with all optimized binaries, our tool performs well, identifying over 90 % of system calls. The switch

`-Os` means the binary is optimized for size. `-O1`, `-O2` and `-O3` refer to increasing level of optimization. Generally, it seems that optimization performed by a compiler is a big advantage for our tool.

We can also see that there are much less system calls in total in the binary that has not been optimized. It seems that leaving the optimization out results in more calls to the functions wrapping system calls in `libc` instead of inlining the `syscall` instruction in each function. This also explains why our tool does not perform that well with non-optimized binaries. This is because the wrappers circulate the system call number through the stack instead of putting it directly to a register, which causes problems for our diversifier tool.

It would be interesting to test more `libc` implementations with several versions of different compilers and see how well our tool performs when analyzing the compiled binaries.

**Table 4.** The effects of compiler optimization to diversification of musl library.

Optimization	Identified	Not identified	Total calls
<code>-O0</code>	7	291	298
<code>-Os</code>	372	27	399
<code>-O1</code>	379	27	406
<code>-O2</code>	373	29	402
<code>-O3</code>	375	31	406

## 6 Methods for System Call Diversification

Basically, the idea of system call diversification simply means that a system call number is replaced with another number. The easiest part in diversifying system calls is changing their numbers in the kernel code. How this is done depends on the architecture and kernel version. In x86-64 architecture, for example, the system call numbers are listed in `arch/x86/syscalls/syscall_64.tbl`. On compiling, definitions in this file are propagated to several header files.

The part that causes more problems is changing the system calls numbers in all binary files to correspond the new numbers we have set in the kernel. As we have seen, the system calls take place in several phases in the binary code, which causes several problems described in the previous section. In this section, we take a look at our own diversification tool and then present some solutions that would help to increase its accuracy to 100%.

### 6.1 Our Tool and Recognizing the System Calls

To demonstrate the feasibility of system call diversification, we implemented an experimental diversification tool as a proof of concept. Our tool rewrites

the system calls in x86-64 ELF-64 binaries by making use of a simple linear sweep algorithm [22]. This is a straightforward disassembly method that decodes everything appearing in sections of the executable that are typically reserved for machine code. We limit the analysis to executable `PROGBITS`-type sections in ELF binaries. The diversification is done after compile time before execution.

The tool tries to find system calls by walking through the program code sections linearly. It looks for `SYSCALL` commands used in x86-64 architecture. When such a command is found, it starts searching the system call number associated with this call. This is done by backtracking from the location of `SYSCALL` command and trying to find the command where the system call number is set. As the number of system call to be invoked is put into a register, our tool looks for commands that change values of `RAX`, `EAX`, `AX`, `AH` or `AL` registers.

Therefore, our diversifier tool uses the following two methods to identify the system calls:

1. *Recognize two consecutive phases.* As we have seen, in the simplest scenario we simply recognize two consecutive phases of the system call in the binary code. However, when there are other commands between these phases, this trivial approach will not work.
2. *Recognize two phases with a gap.* When the two phases of the system call are not consecutive, we have to find the command making the system call first and then backtrack to the call that puts the system call number into a register. Here, the potential jumps between the two phases should be somehow recognized and handled.

**Table 5.** Amount of gaps in system calls in Fedora and Gentoo.

Gap size	Fedora	Gentoo
0	722	736
1	34	29
2	15	19
3	17	8
4	42	6
5	4	7
Over 5	0	2
Total	834	807

Table 5 shows the amount of gaps found in Fedora and Gentoo distributions. In Fedora, 87 % of the system calls have no gaps and in Gentoo, 91 % of the system calls have no gaps. The vast majority of system calls are trivial in this sense. Fedora did not have any gaps bigger than 5 instructions. Gentoo has only two of these, the largest gap being 9 instructions.

When testing our tool, we used SysTap, a tool for real time analysis of running processes in user and kernel spaces. This way, we made sure that the programs that had been diversified with our tool worked correctly during this dynamic instrumentation – that is, they used the new system call numbers changed by our diversifier tool.

As seen from the results in Sect. 5.2, our tool still needs improvement. Next, we will take a look at many approaches that could be used to further improve our diversifier tool.

## 6.2 Challenges

During the development of our tool, we identified some problems in our approach to system call identification. Most of the challenges were linked to the use of a simple linear-sweep based disassembly algorithm. These problems are well known in literature and have for example been discussed by Schwarz et al. [22].

Our algorithm works by first disassembling the executable PROGBITS-type sections of the ELF files. After the initial disassembly we scan the binary for x86-64 specific SYSCALL instructions. If we find such instructions we stop the disassembly process and start backtracking. The backtracking process starts looking for preceding instructions that could assign a value to one of the accumulator registers RAX, EAX, AX, AH, AL. These registers are used for storing the system call number, and as our intention is to patch the system call numbers, we have to figure out what the original system call number was. The backtracking process might fail if it finds a control flow instruction or an instruction that might modify one of the accumulators in an unknown way. Also, we can only identify the system call numbers if the assignments assigning them use only immediate values for storing the numbers. This leaves out all cases where the system call number is assigned indirectly from memory or from another register.

There are various problems in this approach. First of all, the linear-sweep based disassembly process is susceptible to several hard to identify errors. If there is empty space or program data between instructions this might cause the disassembly to produce incorrect results. The fact that the malfunction might not be identified makes the situation even worse, the disassembly process might continue as if nothing unusual had happened producing false instructions or it might stop if the disassembler confronts an invalid instruction.

To solve this problem we would have to utilize a recursive disassembly algorithm. Such algorithm would first start the disassembly from a prespecified offset and continue until a control flow instruction is found. Then the algorithm would have to figure out the possible targets of the control flow instruction and continue the decoding process from there. This approach would solve some of the problems but increases the complexity of the tool considerably, because the recursive approach requires us to figure out the potential control flow paths. For example, if a jump target is specified to be in a certain register we have to figure out how that register gets its value. We would have to perform some form of data-flow analysis to be able to handle these kinds of indirect jumps. The situation is even more complicated. In order to build a control-flow graph we need a data-flow

graph which in turn requires a control-flow graph to be in place. Henrik Theiling [26] refers to this as a chicken and egg problem.

Reconstruction of control-flow graphs from binaries has been widely studied. Theiling presented a bottom-up approach for the flow graph approximation [26]. Cooper et al. introduced an algorithm for building a control flow graph approximation and then refines it [7]. Kinder et al. devised an abstract interpretation-based framework that produces the most precise overapproximation of the control-flow graph with respect to the used abstract domain [12].

Performing a proper data-flow analysis would also help us figure out how the system call numbers are assigned. With a data-flow graph in place we could backtrack through the indirect assignments and find out how the registers' values are formed. The control-flow graph would also help us solve challenges like the ones presented in Sect. 6.2.

### 6.3 Methods to Improve System Call Diversification

There are several methods we can use to improve the system call diversifier so that it can handle the remaining problematic system calls:

1. *Include the diversification calculation in the binary.* We can embed the diversification calculation – that is, the calculation determining the new diversified system call number – somewhere in the binary. However, this might cause some relocation problems. This approach would also make a potential leakage of diversified code quite dangerous. As a result, the secret new system call mappings defined by the diversification function would be revealed. However, considering we assume a piece of malware should not be able to perform system calls and get access to the file system in order to analyze the diversified binary code, this approach should be pretty safe. If the malware finds some way to get into the memory space of an executing process, however, it can try to analyze the meanings of diversified system calls.
2. *Change compiler or compiler switch settings.* Sometimes the order of commands in the machine code can be changed as an optimization made by the compiler and system call numbers can be circulated through registers and memory before they are put in the appropriate registers in order to make a system call. This could probably often be prevented with correct compiler settings. This method naturally has some problems. For example, we cannot expect all software developers – like the major browser suppliers – to compile their binaries for us using a certain compiler or some specific configuration. Many open source applications could be compiled from source codes on the target machine using a specific compiler, though.
3. *Rewriting parts of the source code.* Many problems faced in the binary code diversification process can probably be traced back to the source code. As a consequence, rewriting some of the source code differently might solve the problem. In many systems, rewriting would cause too much work if it would be done for all user applications. However, it is a possible solution for example for many standard libraries like `libc`.



4. *Hard-code the diversification.* For some of the most problematic code sections, the diversification could be hard-coded in binaries. While not usually a preferable solution, this could be done for some standard parts of the Linux operating system.

The various methods for more accurate diversification we have discussed in this section all have some challenges. However, they can still be successfully used at least for some standard set of libraries and applications. Also, these methods are very feasible in some more or less restricted environments. Systems used in industry or military and embedded systems in general are easier to adapt this way, and security is often a major concern for these systems. In these systems, we believe we can reach 100 % diversification accuracy.

## 7 Related Work

To better position our work in the Linux based software ecosystem, we shortly discuss existing related technologies in this section and provide a summary of related research in the field.

### 7.1 Related Technologies

The traditional UNIX point of view to security is based on a discretionary user and group based restriction of file/process privileges to perform operations, with the exception of a superuser with access to all such resources. The system is binary in nature, i.e. an operation is either prohibited or allowed to full extent. It was later extended with more flexible *access control lists* (ACL) and policy based controlling mechanisms (PolKit) [9].

Another way to control actions is sandboxing. The *chroot* mechanism [9] provides an isolated view of the file system. As the superuser is allowed to break out from the *chroot* “jail”, local root exploits pose a security threat. The *chroot* also has other attack vectors such as the *ptrace* system call. For mount points there is a *noexec* flag that prevents the execution of binaries from that file system, but will not prevent interpreting scripts from such locations.

Sandboxing is not limited to file systems. For example, Linux provides namespace isolation for process identifiers, network interfaces, firewall rules, routing, and inter-process communication and a related container framework (LXC) [11]. Other types of resource limits can be imposed via the *ulimits*, *sysctl*, and control group interfaces. The *Linux Secure Computing Mode* (*sec-comp*) mechanism can be used to isolate a process from system on system call level with only a very limited interface to outside system via already-open file descriptors.

A more disciplined approach to security is mandatory access control. Frameworks such as SELinux and AppArmor introduce a policy based mechanism to security with modular hooks directly on kernel level. The policy is enforced by kernel, but its definition comes from userspace, which also deals with logging

and informing about policy violations. The frameworks enable a fine-grained policy control with a small runtime overhead, and while the framework can be transparently set up on a system without changing the userspace applications, programs that are not designed for such a rigorous enforcement of permissions may trigger false warnings with careless resource usage patterns.

## 7.2 Related Research

In 1993, Cohen [4] introduced a general method of program diversity to protect operating systems. He proposed the exploitation of the evolutionary defenses to produce more complex and unique program instances. The higher complexity of the program increases the work an attacker has to do to understand the program's behavior in order to perform an attack. Moreover, with the uniqueness of the program, the attacker is no longer able to impact a substantial number of program versions with a single attack. This way, the attacker is forced to design individual attack versions for each of the program instances.

According to the classification of Collberg [5], there are various obfuscation techniques available: code obfuscation, data obfuscation, layout obfuscation, and preventing transformation. Based on the distribution format of the software, different techniques are applicable [15]. In [15] these techniques are used at the binary level.

Binary obfuscation makes reverse engineering the software significantly harder. In the reverse engineering process the machine code is disassembled into assembly code. The assembly code is then decompiled and the high-level code is recovered [14]. Linn and Debray [14] propose adding “junk bytes” into the instructions where the disassembler is expecting code. This method can disrupt the disassembly process to produce disassembly errors or at least make disassembled code more complex. The candidate instruction code should be incomplete (to confuse disassembler) and unreachable during the execution (to save the program's semantics). In [16], similar to [14], the goal is to make the disassembly of the machine code and thus the reverse engineering harder. They propose two different obfuscation techniques that make it more difficult for the disassembler to find the actual control flow of the binary code. One technique is to modify the control transfer instructions so that they cause traps and signals. The other technique is to add new bogus instructions (e.g., adding the conditional jumps that are disassembled but are never taken, or adding junk bytes that cause incorrect disassembly). Falcarin et al. [8] propose a novel binary obfuscation technique that is based on code mobility and code splitting at binary level. Their approach aims at obstructing the static and dynamic analysis and therefore the reverse engineering. Mimimorphism [28] is another binary obfuscation technique that the malware can use to hide itself from static and semantic analysis.

The idea of system call diversification was introduced by Chew and Song[3] for the first time, to mitigate the computer intrusions. In [3], the randomization is applied to operating system to defeat buffer overflows. One of their proposed methods is randomizing the system call mappings. Each system call is mapped

to a corresponding numbers in a table. By altering (randomizing) the mappings, the original system call will no longer work.

System call diversification has also been studied in [13]. The authors propose it as a countermeasure against injection code attacks. This work is continued in [10], where the authors apply instruction set randomization and address space layout randomization simultaneously. These papers advocate randomization (diversification) that happens dynamically at load time or run time, which causes some performance loss. They also require de-randomization, because the kernel is not diversified. We diversify binaries after they have been compiled so that the run-time performance is not affected. Unlike these earlier papers, our approach also provide system-wide protection by also diversifying the kernel.

Srivastava et al. [24] have designed an attack called Illusion. Illusion obfuscates the kernel's system calls that are used by the attacker to hide the actual operation of the malicious program. With the help of Illusion, attackers can stay invisible to the malware analyzers; since these analyzers rely on the standard system call interface to detect any changes and also the analyzers do not consider the actual execution behavior of the system call in the kernel. Moreover, Illusion is not detected by the tools checking the integrity of the kernel; because it does not make any alteration in data structure or code of the kernel. In addition, they have designed a detection system for detecting their attack.

The basic behavior of a program is recognizable by following its execution flow, i.e. by tracing the sequence of the system calls the program invokes in execution phase. Brusch et al. [2], proposed an obfuscator that works at kernel-level and randomizes the sequence of the invoked system calls. Randomization makes the program's execution flow unpredictable for attacks.

In our previous research we have studied the applicability of diversification techniques in different levels of software. In [17] we aim at concealing the system call interface in order to protect the operating system, while in [21] we focus on diversification at higher levels, i.e. Ajax applications. We propose a proxy-like obfuscator [21] to defeat the online banking Trojans. We implemented our approach in [19] and illustrated its efficiency. In two other papers [18,20] we consider the use of diversification techniques to mitigate the man-in-the-browser attacks.

## 8 Conclusions

In this paper, we have presented a scheme for large-scale system call diversification for operating system protection and also implemented a concrete diversifier tool to demonstrate feasibility of our approach. Our experiments show that a large majority of system calls is handled well by our tool, but there are still some challenges. Still, the numbers of unidentified calls were usually relatively small for analyzed binary files.

To overcome the challenges, we have also discussed several ways to increase the accuracy of our diversification scheme to 100%. Based on this, we believe system call diversification is a feasible approach for protecting operating systems

from malware. This is especially true for systems where a certain set of well-known libraries and applications is used and in many embedded systems that are more restricted in nature.

The small total amount of system calls also makes things easier. As we have seen, very few binaries in the tested Linux distributions contained direct system calls. Most direct system calls are in standard libraries and well-known command line tools, not in the ordinary applications.

There are still many open questions related to our diversification scheme. How and where do we store the system call number mapping as a secret? How would we invoke our diversification tool in an operating system? Would it run in the kernel or in user space? How is it protected? These details will be discussed in future work.

Only Linux has been covered in this paper. It would be interesting to also study our diversification scheme in other operating systems. Most likely, similar methods can be used and there are similar challenges present in the contexts of those systems, too.

## References

1. Apvrille, A., Strazzere, T.: Reducing the window of opportunity for android malware gotta catch 'em all. *Int. J. Ambient Comput. Intell.* **8**(1–2), 61–71 (2012)
2. Bruschi, D., Cavallaro, L., Lanzi, A.: An efficient technique for preventing mimicry and impossible paths execution attacks. In: *Performance, Computing, and Communications Conference, 2007, IPCCC 2007. IEEE International*, pp. 418–425, April 2007
3. Chew, M., Song, D.: Mitigating buffer overflows by operating system randomization (2002)
4. Cohen, F.B.: Operating system protection through program evolution. *Comput. Secur.* **12**(6), 565–584 (1993)
5. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscation transformations. Technical report 148, The University of Auckland (1997)
6. TIS Committee: Tool Interface Standard. Executable and Linking Format (ELF) Specification. Version 1.2. Submitted to *Journal of Information Security and Applications* (Elsevier), under evaluation (1995)
7. Cooper, K.D., Harvey, T.J., Waterman, T.: Building a control-flow graph from scheduled assembly code. Technical report 02–399, Rice University (2002)
8. Falcarin, P., Carlo, S.D., Cabutto, A., Garazzino, N., Barberis, D.: Exploiting code mobility for dynamic binary obfuscation. In *2011 World Congress on Internet Security (WorldCIS)*, pp. 114–120, February 2011
9. Jang, M.H., Jang, M.: *Security Strategies in Linux Platforms and Applications*. Jones & Bartlett Publishers, Burlington (2010)
10. Jiang, X., Wang, H.J., Xu, D., Wang, Y.-M.: Randsys: thwarting code injection attacks with system service interface randomization. In: *IEEE International Symposium on Reliable Distributed Systems, SRDS 2007*, pp. 209–218 (2007)
11. Kerrisk, M.: *The Linux Programming Interface*. No Starch Press, San Francisco (2010)
12. Kinder, J., Zuleger, F., Veith, H.: An abstract interpretation-based framework for control flow reconstruction from binaries. In: Jones, N.D., Müller-Olm, M. (eds.) *VMCAI 2009. LNCS*, vol. 5403, pp. 214–228. Springer, Heidelberg (2009)

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, pp. 584–587 (2009)
14. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, pp. 290–299. ACM, New York, USA (2003)
15. Madou, M., Anckaert, B., De Bus, B., De Bosschere, K., Cappaert, J., Preneel, B.: On the effectiveness of source code transformations for binary obfuscation. In: Proceedings of the International Conference on Software Engineering Research and Practice (SERP06), pp. 527–533. CSREA Press (2006)
16. Popov, I.V., Debray, S.K., Andrews, G.R.: Binary obfuscation using signals. In: USENIX Security (2007)
17. S. Rauti, J. Holvitie, and V. Leppänen. Towards a Diversification Framework for Operating System Protection. In: Proceedings of International Conference on Computer Systems and Technologies, CompSysTech 2014 (2014)
18. Rauti, S., Leppänen, V.: Browser extension-based man-in-the-browser attacks against Ajax applications with countermeasures. In: Proceedings of International Conference on Computer Systems and Technologies, CompSysTech 2012, pp. 251–258. ACM Press (2012)
19. Rauti, S., Leppänen, V.: A proxy-like obfuscator for web application protection. *Int. J. Inf. Technol. Secur.* **5**(1) (2014)
20. Lee, J.W., Lee, Y.J., Kim, H.K., Hwang, B., Ryu, K.H.: Discovering temporal relation rules mining from interval data. In: Shafazand, H., Tjoa, A.M. (eds.) *EurAsia-ICT 2002*. LNCS, vol. 2510, pp. 57–66. Springer, Heidelberg (2002)
21. Rauti, S., Leppänen, V.: Resilient code protection by JavaScript and HTML obfuscation for Ajax applications against man-in-the-browser attacks. Submitted to *Journal of Information Security and Applications* (Elsevier), under evaluation (2014)
22. Schwarz, B., Debray, S., Andrews, G.: Disassembly of executable code revisited. In: Proceedings of Ninth Working Conference on Reverse Engineering, pp. 45–54 (2002)
23. Sobell, M.G.: *A Practical Guide to Linux*. Addison-Wesley, Boston (1999)
24. Srivastava, A., Lanzi, A., Giffin, J., Balzarotti, D.: Operating system interface obfuscation and the revealing of hidden operations. In: Holz, T., Bos, H. (eds.) *DIMVA 2011*. LNCS, vol. 6739, pp. 214–233. Springer, Heidelberg (2011)
25. Tanenbaum, A.S.: *Modern Operating Systems*, 3rd edn. Prentice Hall Press, Upper Saddle River (2007)
26. Theiling, H.: Extracting safe and precise control flow from binaries. In: Proceedings of Seventh International Conference on Real-Time Computing Systems and Applications, pp. 23–30. IEEE (2000)
27. Wang, S.P.: *Mastering Linux*. CRC Press, Boca Raton (2011)
28. Wu, Z., Gianvecchio, S., Xie, M., Wang, H.: Mimimorphism: a new approach to binary code obfuscation. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, pp. 536–546. ACM, New York, USA (2010)

Trusted Systems

6th International Conference, INTRUST 2014, Beijing,  
China, December 16-17, 2014, Revised Selected Papers

Yung, M.; Zhu, L.; Yang, Y. (Eds.)

2015, XIII, 442 p. 83 illus. in color., Softcover

ISBN: 978-3-319-27997-8