International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 15, Number 2, pages 118–137, July 2025

Implementation and Evaluation of a System Call Moving Target Defense Applied Multiple Times at Runtime for Binary Injections

Yuta Minato
Graduate School of Information Science and Electrical Engineering, Kyushu University
744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan

Takeshi Masumoto
Graduate School of Information Science and Electrical Engineering, Kyushu University
744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan
Currently: NEC Corporation
5-7-1 Shiba, Minato-ku, Tokyo 108-8001, Japan

and

Hiroshi Koide
Research Institute for Information Technology, Kyushu University
744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan

Received: February 14, 2025
Revised: May 4, 2025
Accepted: June 10, 2025
Communicated by Yasuyuki Nogami

Abstract

We propose and evaluate a system call-based Moving Target Defense (MTD) mechanism as a countermeasure against code injection attacks that exploit unknown vulnerabilities. Although integrating the proposed MTD mechanism into the OS kernel would be more ideal, we implemented it in userland for this study in order to demonstrate its feasibility and evaluate its effectiveness. The proposed system randomizes the mapping between system call numbers and their corresponding functions, thereby invalidating system calls issued by injected malicious code. Since system calls serve as the primary interface through which user applications access system resources, this randomization prevents attackers from achieving their objectives, even if they successfully inject code into a process. This approach, categorized as an MTD technique, is particularly promising against zero-day attacks, where vulnerabilities are exploited before they are patched. By dynamically altering the mapping at each system call invocation, the system increases its runtime diversity and unpredictability. While kernel-level implementation remains a future goal, our evaluation—conducted by remapping system call invocations through a userland wrapper—demonstrates that the proposed method can detect and mitigate code injection attacks in a wide range of existing compiled programs, without requiring specialized hardware support.

Keywords: information security, Moving Target Defense, system call randomization, code injection attack

1 Introduction

This research aims to defend against cyber attacks on information systems. In particular, we focus on code injection attacks including zero-day attacks and propose a new defense method against them. If the system is vulnerable to code injection attacks, arbitrary programs prepared by attackers may be executed on it. Therefore, this vulnerability is highly serious, and countermeasures against it are essential [1]. Code injection attacks are typically performed using vulnerabilities, such as stack overflow vulnerabilities. Attackers first inject the program into memory on the target system and finally make the system jump the CPU execution point to the position of the injected code to make the system execute the program.

Data execution prevention (DEP) is one of the existing defense methods against injection attacks [2–4]. DEP prevents code execution from the stack or heap by marking memory areas in a process non-executable unless the area contains executable code. This raises an exception when attempting to execute these areas. Although DEP prevents many attacks, it has some shortcomings. First, DEP requires hardware support, making it difficult to use DEP when we select a CPU that does not support it. Second, all areas in the program should be clearly separated into the code and data areas before execution. In some programs, such as the JIT compiler and self-modifying code, we put executable codes in the data area, thus, DEPs sometimes cannot protect these programs. Third, there are a few cases where it is bypassed [5–7]. Given these circumstances, DEP is not a perfect solution for code injection attacks.

In this paper, we propose and evaluate a system call-based Moving Target Defense (MTD) [8] mechanism as a countermeasure against code injection attacks that exploit unknown vulnerabilities. Although integrating the proposed MTD mechanism into the OS kernel would be more ideal, we implemented it in userland for this study in order to demonstrate its feasibility and evaluate its effectiveness. The proposed system randomizes the mapping between system call numbers and their corresponding functions, thereby invalidating system calls issued by injected malicious code. Since system calls serve as the primary interface through which user applications access system resources, this randomization prevents attackers from achieving their objectives, even if they successfully inject code into a process. This approach, categorized as an MTD technique, is particularly promising against zero-day attacks, where vulnerabilities are exploited before they are patched. By dynamically altering the mapping at each system call invocation, the system increases its runtime diversity and unpredictability. While kernel-level implementation remains a future goal, our evaluation—conducted by remapping system call invocations through a userland wrapper—demonstrates that the proposed method can detect and mitigate code injection attacks in a wide range of existing compiled programs, without requiring specialized hardware support [9, 10].

The system call MTD is a randomization technique that disables code injection attacks by randomizing the mapping between system call numbers and functions. As system calls are the only way for user applications to access system resources, system call MTD limits the processing and resources that an injected by attackers program can perform and access. There are existing research projects on system call mapping randomization [11–15], which perform randomization once, before loading the program into memory. However, such methods have no effect when information about randomization is disclosed to attackers. We enhanced the system call mapping randomization by continuing to re-randomize multiple times at runtime to resist information disclosure. We have confirmed that our proposed method is an effective defense method against zero-day attacks, such as code injection attacks, through experiments using smashme [16], which contains a buffer overflow vulnerability, and a Bash program with the ShellShock [17] vulnerability, where code embedded in the value of an environment variable is interpreted and executed by Bash. In other experiments, we measured the performance overhead in the execution time of the system call MTD applied by our method. One of the unique features of this method is that it applies a binary patch to a legitimate user program so that correct system calls can be issued even in an execution environment where system call MTD is applied.

The remainder of this study is organized as follows: Section 2 describes related research; Section 3 formalizes the threats that the system call MTD mitigates, and Section 4 explains the design of the proposed method; Section 5 explains the two components that our method consists of; Section 6

describes how to apply the method by directly rewriting the compiled program; Section 7 describes the limitations of our method; Section 8 describes the experimentation performed to evaluate the effectiveness and performance; Section 9 describes the improvements in our method; and finally, Section 10 concludes.

2 Related Works

Code-injection attacks have been addressed using three primary defense strategies: memory protection, instruction randomization, and system call randomization.

The first category is Data Execution Prevention (DEP) [2–4], which implements page-level memory protection by marking memory regions as non-executable. For example, DEP prevents the execution of injected code by ensuring that the stack or data segments cannot be executed. This technique requires hardware support to generate an exception when an attempt is made to execute instructions from a non-executable page.

The second category is randomization-based defenses, also known as Moving Target Defense (MTD) techniques. These defenses enhance security by introducing uncertainty into program behavior or memory layout, thus making it harder for attackers to craft reliable exploits. A widely used example is Address Space Layout Randomization (ASLR) [18, 19], which randomizes memory addresses such as the stack, heap, and libraries at runtime. However, on resource-constrained systems, ASLR may lack sufficient entropy, allowing attackers to eventually infer critical memory locations [20].

Another approach in this category is Instruction Set Randomization (ISR) [21, 22], which randomizes the instruction set itself. This prevents the successful execution of injected code unless it conforms to the randomized instruction set. Although ISR can be implemented in software, it often incurs high overhead. Hardware-assisted ISR can mitigate this performance penalty.

The third strategy is System Call Randomization, particularly System Call MTD, which provides a lightweight alternative to ISR. Instead of randomizing the entire instruction set, it randomizes only system call numbers, thereby reducing overhead. Previous studies on system call randomization [11–15] typically applied the randomization once at process startup. This approach can be circumvented if attackers manage to probe the mapping through trial execution and behavior observation.

To address this limitation, our work introduces repeated randomization of system call mappings during runtime, ensuring that even if attackers discover the mapping at one point, it will become obsolete due to subsequent re-randomization. This technique provides resilience even under partial information disclosure, improving overall defense against runtime binary injection.

In our earlier work [23], system call MTD was demonstrated only with small, library-free programs due to implementation limitations at the time. As a result, the evaluation was restricted in scope. In this study, we overcome those limitations by enabling our MTD approach to support nearly all statically linked programs, and we evaluate its effectiveness and performance through extensive experiments.

3 Threat Model

We consider a strong threat model in which an attacker has already succeeded in injecting malicious code by exploiting vulnerabilities in the system. Despite this compromise, our system call MTD mechanism is designed to prevent the injected code from performing any meaningful actions and to facilitate the detection of such execution attempts. This assumption underscores that our approach is not restricted to any particular code injection technique, thereby ensuring broader applicability.

Furthermore, we assume the target programs follow a common fault-tolerant design pattern where a new worker process is spawned using `fork` whenever an existing worker crashes. This pattern is prevalent in real-world server applications. Under this model, we also consider a more persistent adversary capable of performing clone-probing attacks, where the attacker probes not only a single process but also continues probing across newly spawned processes.

4 Design

In this research, we make the assumption that the tracer program is safe. The assumed threat model is that an attacker can insert code from outside at any time, and we made it possible to experiment with a PoC (Proof of Concept) of the proposed method in the form of a tracer, a function that should originally be possessed by the OS. In other words, the assumed model is based on the premise that an unknown vulnerability exists that allows code to be injected from outside, and all user programs are executed using the tracer. Although this function should originally be possessed by the OS kernel, this time we made it easy to execute as a PoC, and demonstrated the effectiveness of the proposed method, that is, that it can defend against and detect injection attacks.

4.1 Re-randomization Points

Runtime randomization is divided into two classes based on randomization trigger conditions [24]. One randomization trigger condition is clock-timing-based randomization, where re-randomization is triggered at fixed intervals. However, this is inefficient in cases where it is clear what causes the information disclosure about randomization, or what is related to the progress of the attacker's probing. The other randomization trigger condition is risk-based on-demand runtime randomization, where re-randomization is triggered at risky operations that may cause information disclosure. This method of approaching the problem directly was more efficient.

Generally, which does not depend on how the system call MTD is implemented, the method of invalidating randomization is to call the system call, and then obtain information about the randomized mapping from its behavior. From this perspective, a system call itself is an action that may give attackers information about randomization. Therefore, it is considered appropriate to use the timing of system call invocation as the trigger condition for re-randomization.

4.2 Randomization Space

Several operating systems do not have many system calls; for example, Ubuntu 20.04 has 334 system calls. Therefore, when using the shuffling method (interchanging with each other in the existing system call number space) in randomization, the probability that an attacker will choose a certain intended system call in a randomized context is not sufficiently small and is a few hundredths. However, it is possible to increase randomization space. For example, in the x86_64 architecture, a system call is called by first storing the system call number in the `rax` register and then executing the `syscall` (`sysenter`) instruction. Because the size of the `rax` register is 64 bits, the actual size of the system call number space is 2^{64}. Certainly, in this case, an attacker has only an $\frac{1}{2^{64}}$ chance of calling the intended system call. Also, when n system calls are randomized, the probability that the attacker correctly selects the randomized system call is $1/(2^{64}(2^{64}-1)\cdots(2^{64}-(n-2))(2^{64}-(n-1)))$, which is very small. Therefore, the possibility of a successful attack can be greatly reduced by increasing the number of system calls to be randomized.

The `/dev/random` of the Linux 6.8.0 kernel used in the experiment has Secure Random Generators (CSPRNG), a cryptographically secure pseudorandom number generator built into the kernel, available, and is considered sufficient for general use [25].

4.3 Randomized System Calls

Fig. 1 shows the correspondence between the randomized system call table and the original table when only the `write` is randomized. The table on the left in Figure 1 is a "randomized system call table" sorted in ascending order with a maximum of $2^{64}-1$. The size is not 18 exabytes large. As mentioned in Section 4.2, the randomization space size is 2^{64}, and the numbers of system calls are 334; therefore, many numbers are not assigned to any system calls. The numbers that these attackers might use are assigned to the `getpid`. The `getpid` system call simply returns the process ID of the calling process, thus, it would be impossible for the execution of the `getpid` to harm the system.

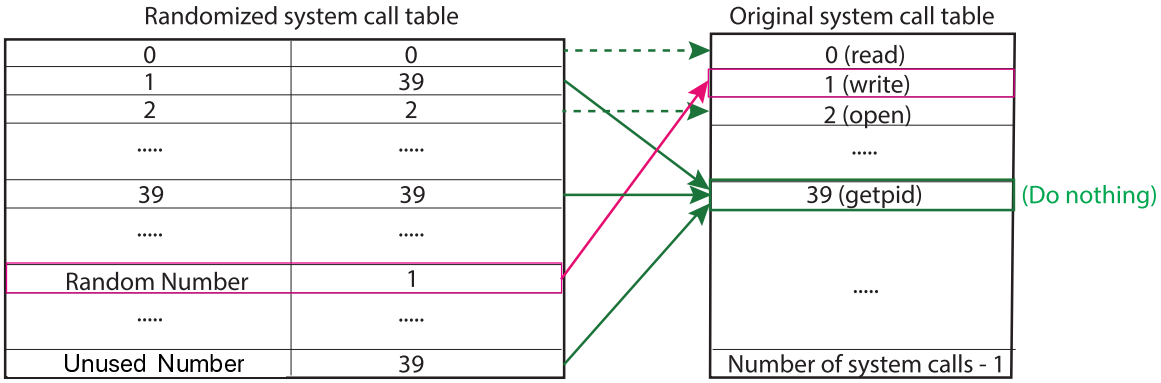


Figure 1: Correspondence between the randomized system call table and the original table.

Regarding the operation of getpid(), for example, in the case of Linux with amd64 architecture, the following number 39 (0x27) is set in the eax register and the syscall command is executed. This causes the system to switch to kernel mode operation.

```
0x00007ffff7cf5a90 <+0>: endbr64
0x00007ffff7cf5a94 <+4>: mov    $0x27,%eax
0x00007ffff7cf5a99 <+9>: syscall
0x00007ffff7cf5a9b <+11>: ret
```

In other words, it transfers CPU control from user mode (Ring3) to kernel mode (Ring0), automatically switches the stack, saves the values of the user mode instruction pointer (RIP) and flag register (RFLAGS), jumps to the address of the system call handler, and executes the system call corresponding to the system call number (value of the eax register). In the case of getpid, it copies the process number (PID) in a data structure called the process control block (PCB) inside the kernel to eax.

Not all system calls require to be barred from being executed by attackers. The overhead incurred by applying this technique can be reduced by minimizing the number of system calls to which randomization is applied. This study does not discuss which system calls must be prohibited from being executed by attackers.

5 Implementation

We implemented a system-call mapping MTD. The implementation assumes the x86_64 architecture and Linux OS as the execution environment.

5.1 Overview

As shown in Fig. 2, our prototype consists of two components: a target and a tracer program. The target program is the substance of the application, and the tracer program is a system call virtualizer that is added to apply the system call MTD to the target program. The target program has a correspondence table between system calls and randomized numbers in memory, which refers to when calling the system calls. The tracer intercepts the execution of a `syscall` instruction by the target, refers to the `rax` register (the number of called system calls), de-randomizes it, and stores the original system call number in the `rax` register. It then sends a new randomized system call table to the target program using inter-process communication (IPC) and resumes its execution. The target updates the old system call table with the new one received from the tracer. The above processing is performed each time the system calls to be randomized.

As aforementioned, the tracer sends a system call table that the target uses for randomization to the target using IPC, although this is tricky. Because the tracer and the target are designed

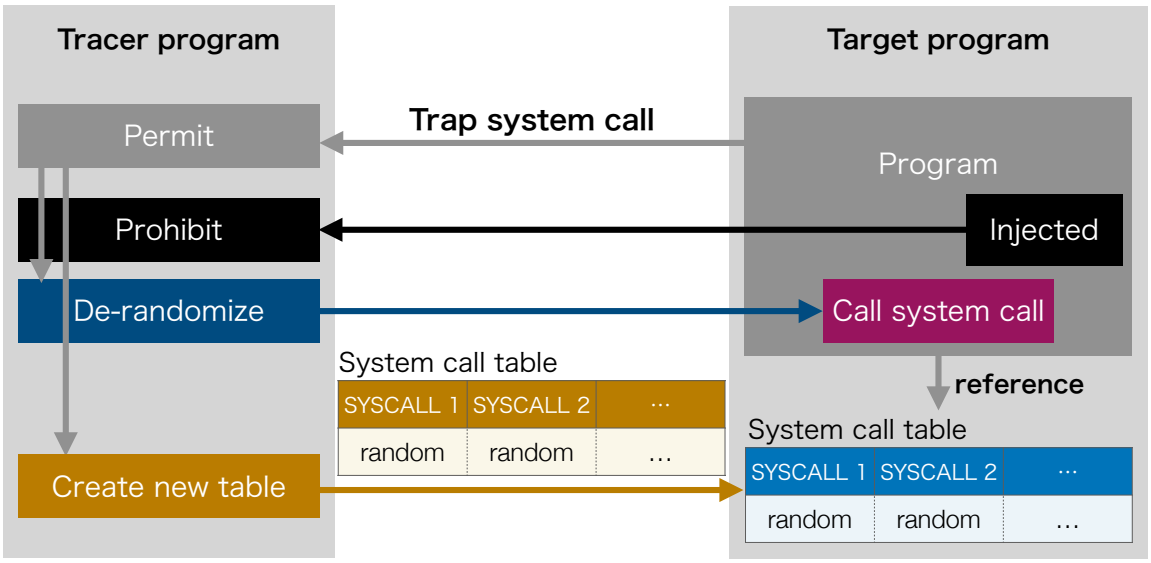


Figure 2: Outline of implementation of the proposed method.

to be separate programs, they cannot perform secure IPC using anonymous shared memory or the like. As we assumed the context in which attack codes can be injected, the IPC requires to be encrypted. However, this makes the security of our method dependent on the encryption algorithm used. Therefore, the tracer sends the encryption key that will be used for the next sending, along with the new system call table to the target. The format of the sending data is shown in Fig. 3. As shown in the figure, as the system call table is just a random number on the data, there is no need to add the key data after the system call table, and the decrypted bits can be the secret key to be used for the next sending. As the system call table is just a random number on the data, there is no need to add the key data after the system call table, and the decrypted bits can be the secret key to be used for the next sending.

5.2 Tracer Program

Algorithm 1 outlines the process of the tracer program. First, the tracer spawns a process to execute the target program, and the parent process traces this process using the `ptrace` system call. Next, it creates a system call randomizer corresponding to the initial process of the target program and sends the randomized system call table to this process. From here, the tracer program repeatedly captures and handles system call invocations from any process of the target program. The tracer program intercepts the system calls of the target program just before and immediately after their invocation. Before the invocation, it references the value in the `rax` register (the system call number), reverses the randomization, and stores the original system call number in the `rax` register. At this point, if a system call should have been randomized but is not, it is considered an attack, and the system call is neutralized as described in Section 4.3. Immediately after the invocation, the tracer updates and sends the corresponding randomized system call table to the process.

As shown in Fig. 4, because the target program may spawn multiple processes, the tracer should apply the system call mapping MTD to the target root process, child processes spawned from the root, and grandchild processes spawned from the child processes. If the invoked system call is `clone`, `clone3`, `fork`, or `vfork`, the tracer identifies the process ID of the new process spawned by the target program from the return value (the `rax` register value) and creates a system call randomizer corresponding to this ID. Consequently, even if the target program is multi-process, the system call MTD can be applied to each process, thereby preventing the previously mentioned clone probing attack.

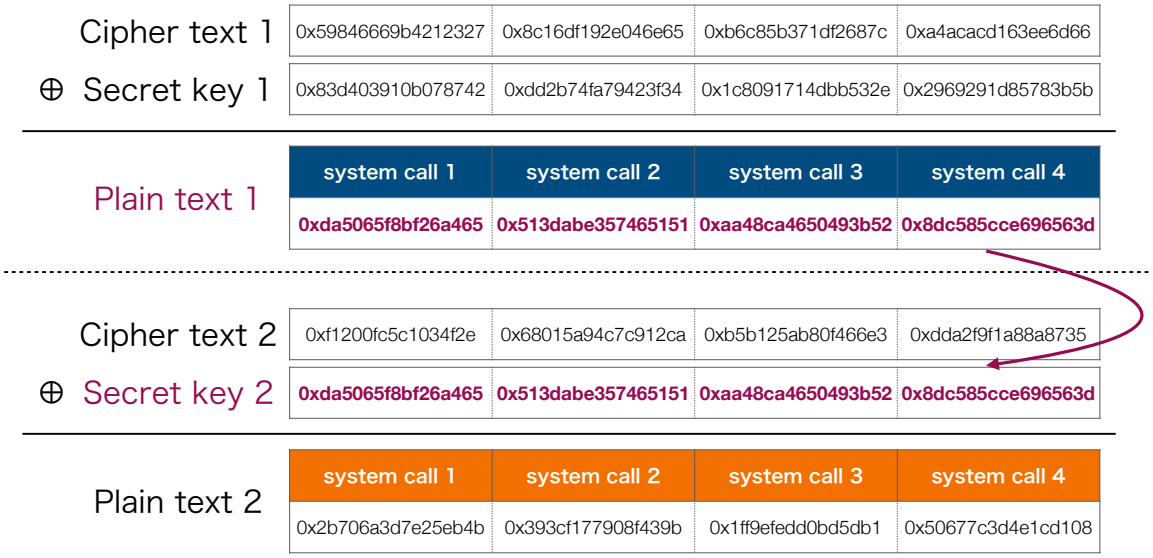


Figure 3: Format of data sent from the tracer program to the target program.

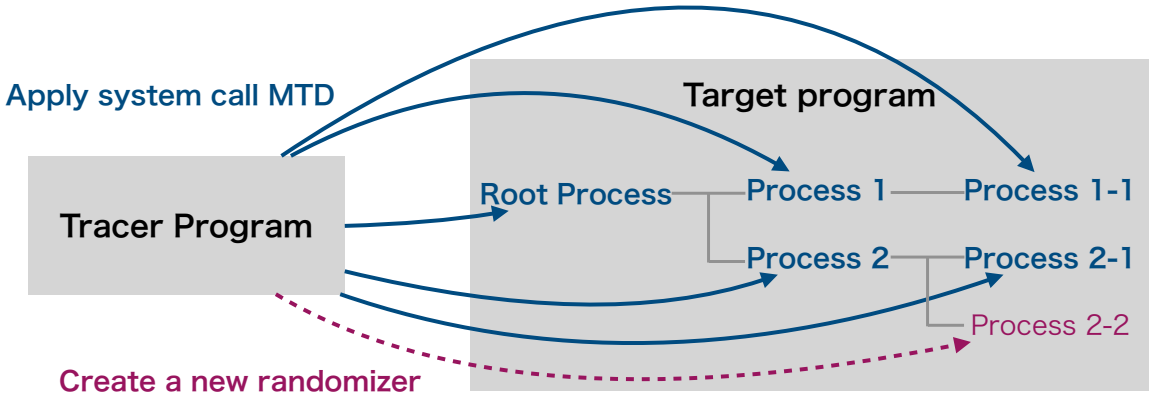


Figure 4: The target program may spawn multiple processes.

6 Application To Target Program

To change the numbers used for system calls to random numbers sent from the tracer program, it is necessary to rewrite the program of the original application. There are three main ways to rewrite the program: rewriting the source code, rewriting the compiler, and rewriting the binary. The method of binary rewriting has the advantage that it can apply the system call MTD even when the source code is unavailable, as is often the case with commercial off-the-shelf (COTS) software. We used E9Patch [26] to apply our method to a compiled program.

E9Patch is a tool for static binary rewriting without the need for control flow recovery and associated assumptions or heuristics. Most static binary rewriting tools require some form of control-flow recovery to adjust the set of jump targets in the rewritten binary. However, because recovering control-flow information from binary code is difficult in general [27, 28], they rely on a set of heuristics or assumptions, such as specific compilers, specific source languages, or binary file metadata information [29]. Because E9Patch is highly scalable by design, where it does not require that information, it can reliably rewrite large binaries including Google Chrome [30] and FireFox [31] web browsers.

Algorithm 2 The processing of the function to be inserted.

```
1: function FUNC(rax)
2:     if rax is TARGET_SYSCALL_NUM then
3:         tid ← GETTID()
4:         MUTEX_LOCK()
5:         randomizer ← GET_RANDOMIZER(tid)
6:         if IS_NOT_INITIALIZED(randomizer, tid) then
7:             INITIALIZE(randomizer, tid)
8:         end if
9:         UPDATE(randomizer)
10:        rax ← GET_SYSCALL_NUM(randomizer, rax)
11:        MUTEX_UNLOCK()
12:    else if rax = SYS_EXIT then
13:        tid ← GETTID()
14:        MUTEX_LOCK()
15:        DELETE_RANDOMIZER(tid)
16:        MUTEX_UNLOCK()
17:    end if
18: end function
```

Our method inserts function executions before every `syscall` instruction in the original target program using E9Patch. Algorithm 2 presents an overview of the function processing. As mentioned in Section 5.2, each randomizer is provided to each process (thread) using the tracer program. Therefore, in the target program, each thread requires its own randomizer. In the C language, we can use thread-local storage [32], which is static memory local to a thread, with the `__thread` or `thread_local` (since C11 [33]) keywords. However, it is impractical to insert thread-local variables that are not managed by the original target program. Instead, our implementation has randomizers on separate chaining [34] tables, which are often used in the implementation of a hash table, as the global shared variable. Therefore, Figure2 includes mutex locking and unlocking. Processing of function branches based on the value of the `rax` register. In the x86_64 architecture, the value of the `rax` register before executing the syscall instruction is the system call number. If the system call number is randomized, randomization should be performed. First, it obtains the randomizer using the thread ID. Then, if the randomizer is not initialized, that is, it is the first randomized system call in the newly spawned thread, it performs an initialization process, such as opening the IPC. It then receives a new randomized system call table through IPC, decrypts it, and updates the key. Finally, it stores a randomized number in the `rax` register. If the system call number is the number of `exit` system calls, it deletes the randomizer corresponding to the current thread.

```
$ ./e9compile.sh func.c
$ ./e9tool \
  --match 'asm=sys(?:enter|call)' \
  --patch 'call [before] func(&rax)@func' \
  target
```

Figure 5: Commands for applying system call MTD using E9Patch.

To apply the system call MTD to the "target," compile the function with `e9compile` and insert it with the `e9tool`(Fig. 5).

7 Limitation

In addition, in this evaluation experiment, we make the assumption that the tracer program itself cannot be attacked. This is due to convenience in implementing the PoC. Ideally, it would be

better to incorporate it into the system program as an OS function. This approach would result in less overhead and would make the tracer program itself unnecessary. Here, we implemented and evaluated the proposed method using the tracer program to confirm whether system call MTD can defend against and detect attacks when there is a binary injection in an information system that is unknown from the defense side.

First, because our method was implemented using E9Patch, E9Patch's limitations apply to our method. This means that if the original program has significantly large code or data segments, causing virtual address space shortages, it may fail to apply our method. The second limitation is that, in most cases, it cannot be applied to dynamically linked binaries. Particularly, the case in which `syscall` instructions are scattered in multiple binaries is not supported. Third, the system calls used in the randomization process shown in Algorithm 2 (`read`, `open`, `getpid`) cannot be randomized owing to implementation.

The second and third limitations have the potential to be remedied by devising an implementation.

8 Evaluation

In this section, we evaluate the effectiveness and performance of the proposed method. We conducted two experiments to apply the system call MTD to programs. These experiments were conducted on Intel(R) Xeon(R) w3-2423, and DDR5-4800 RAM 64GB.

8.1 Exploit Testing

8.1.1 Smashme

Listing 1: smashme.c.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
  char smashme[64];
  puts("Welcome to the Dr. Phil Show. Wanna smash?");
  fflush(stdout);
  gets(smashme);
  if(strstr(smashme, "Smash me outside, how bout dAAAAAAAAAAA")) {
    return 0;
  }
  exit(0);
}
```

```
$ gcc -fno-stackprotector -z execstack -static smashme.c
```

Figure 6: Compiling smashme.c.

To verify the effectiveness of our method, we tested whether the method could defeat a code injection attack. In this experiment, the system call MTD randomized only the `execve` system call. We executed exploits against both the program with the system call MTD applied and the original program. The program was the binary of *smashme* [16], which is one of the challenges in DEF CON CTF Qualifier 2017 (Capture The Flag competitions). The source code is listed in Listing1. It was compiled using the command shown in Fig. 6. The *smashme* program was compiled by disabling the stack protector, allowing an executable stack, and linking it statically using GCC. This means

that it lacks security protection, and the stack was executable. The program read some bytes from stdin and had buffer overflow vulnerability, which allowed us to inject code, [35] that executes a shell.

```
Welcome to the Dr. P
hil Show. Wanna smas
h?
$ ls
flag smashme
$ cat flag
The flag is: ...
```

(a) (a) Performance without system call MTD.

```
Welcome to the Dr. P
hil Show. Wanna smas
h?
Invalid system call
```

(b) (b) Performance with system call MTD.

Figure 7: Getting a shell by exploiting *smashme*.

The results are shown in Fig. 7. In the case of *smashme* without the system call MTD, the exploit successfully obtained a shell and captured the flag by executing commands in it (Fig. 7(a)). However, when the system call MTD was applied to *smashme*, the exploit failed and could not obtain a shell or capture the flag (Fig. 7(b)). The output *"Invalid system call"* in the figure is from the tracer program, and indicates that the system call was called with an unassigned number and was invalidated. The shellcode tried to call `execve("/bin/sh", ...)` but could not, because the `execve` was randomized.

8.1.2 Shellshock

```
$ ./configure CC="gcc -static" CFLAGS="-static" LDFLAGS=-Bstatic
$ make
$ make install
```

Figure 8: Compiling bash-4.3.

The second experiment involved evaluating the application of MTD to Bash, an OSS (Open Source Software) commonly used in enterprise systems. It was compiled using the command shown in Fig. 8. Bash was compiled as a statically linked binary by configuring GCC to use static linking, followed by the execution of the usual `make` and `make install` commands. In this experiment, the system call MTD randomized only the `write` system call. The vulnerability was *ShellShock* [17]. *ShellShock* is a vulnerability wherein code embedded within environment variables is interpreted and executed by Bash.

```
$ env x='()  :;; echo VULNERABLE' bash-4.3 -c "echo This is a test"
VULNERABLE
This is a test
```

Figure 9: Shellshock.

When using a Bash version affected by the *ShellShock* vulnerability, unintended code execution occurs, resulting in the output "VULNERABLE" (Figure9). Furthermore, when executing commands in a Bash environment with system call MTD applied, it is necessary to apply system call MTD to those commands as well. In this experiment, we obtained the precompiled BusyBox binary (AMD64 version) [36], applied MTD to it, and conducted evaluation experiments. BusyBox [37] combines tiny versions of many common UNIX utilities into a single small executable. By applying MTD to BusyBox, it is possible to collectively apply MTD to commands executed in a Bash environment.

```
$ ./tracer ./bash-4.3-test
Invalid system call
Invalid system call
Invalid system call
Invalid system call
Invalid system call
$ echo This is a test
This is a test
$ ls
Invalid system call
$ ./busybox-test ls
busybox-test tracer e9compile.sh func.c
bash-4.3-test e9tool
```

(a) (a) Command execution by Bash with MTD applied.

```
$ env x='()  :;; ls' ./tracer ./bash-4.3-test -c "echo This is a test"
Invalid system call
Invalid system call
Invalid system call
Invalid system call
Invalid system call
Invalid system call
Invalid system call
Invalid system call
```

(b) (b) Execution of "ls" by ShellShock.

```
$ env x='()  :;; ./busybox-test ls' ./tracer ./bash-4.3-test -c "echo This is a test"
busybox-test tracer e9compile.sh func.c
bash-4.3-test e9tool
Invalid system call
Invalid system call
Invalid system call
Invalid system call
Invalid system call
```

(c) (c) Execution of "busybox-test ls" by ShellShock.

Figure 10: Experiments with Bash.

The results are shown in Fig. 10. Both bash-4.3-test and busybox-test have system call MTD applied. In Figure10(a), the message "Invalid system call" is output during the startup process of bash-4.3-test due to the use of an external write system call, although the startup itself is successful. The echo command functions correctly since it is an internal Bash function, similar to pwd and cd, but the ls command fails to execute because it is an external function. Conversely, the internal ls command within BusyBox executes successfully because system call MTD has been applied to BusyBox. In Figure10(b), an attempt is made to embed and execute ls within the value of an environment variable using the ShellShock vulnerability, resulting in the output "Invalid system call". This indicates that the attempted call to ls, being an external Bash function, failed because the write system call had been randomized. In Figure10(c), the execution of the command "busybox-test ls" embedded in the value of an environment variable using the ShellShock vulnerability is successful. From these results, it can be concluded that although the ShellShock vulnerability remains, randomizing the write system call prevents the exploitation of external write system calls.

These experiments demonstrated the effectiveness of the system call MTD against buffer overflow vulnerabilities and the Shellshock vulnerability in Bash. It is reasonable to assume that this method can also be applied to other OSS software. When the system call MTD is applied to software with other typical vulnerabilities or even unknown vulnerabilities, similar defensive effects can be expected, as the system calls invoked by an attacker would correspond to unassigned numbers.

8.2 Performance

Our method introduced an increase in execution time owing to the execution of the inserted binary and context switching caused by `ptrace` and mutex locking. We measured the performance overhead of our method by testing the performance of programs to which the system call MTD was applied and the original programs.

Listing 2: threads.c.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define NUM_THREADS 1
#define NUM_LOOPS 1

void *thread_func(void *arg)
{
  for (int i = 0; i < NUM_LOOPS; i++) getpid();
}

int main()
{
  pthread_t v[NUM_THREADS];
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&v[i], NULL, thread_func, NULL) != 0) {
      perror("create\n");
      return -1;
    }
  }
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(v[i], NULL) != 0) {
      perror("join\n");
      return -1;
    }
  }
  return 0;
}
```

The source code is listed in Listing2. The main thread creates `NUM_THREADS` threads, and each thread executes the `getpid` system call `NUM_LOOPS` times in parallel. In this experiment, the system call MTD randomized the `getpid` system call. If several threads executed the inserted function simultaneously, the possibility of incurring additional overhead may have increased because of mutex locking. The execution time in real time was measured while varying the number of threads (`NUM_THREADS`) and `getpid` system call invocations (`NUM_LOOPS`). We chose the `getpid` system call to measure this overhead. Because the `getpid` system call is one of the fastest system calls, a large part of this program was occupied by the processing of the inserted function. This forces more threads to enter the critical section at the same time. We measured the real-time execution time while changing the number of times the system call was called and the number of threads. When the number of randomized system calls is two or more, one of them is the `getpid` system call, and the rest specify numbers from `600` onwards, for which the functionality is not defined.

Fig. 11 and Fig. 12 show the result of the execution time performance test. Clearly from these figures, the execution time decreases when the number of threads increases in the case with the system call MTD as in the case without the system call MTD. However, in programs to which the system call MTD was applied, the rate of diminution in execution time decreased as the number of threads increased. We assumed that this was because mutex locking in the target program and `ptrace` by the tracer program serialized part of the processing. From this graph, it can be observed that even with an increase in the number of randomized system calls, the increase in execution time was not particularly significant.

We conducted an additional experiment to analyze the causes of the execution time overhead and the amount of overhead. In the case that the target program had only one thread, the factors

that affected the execution time were mainly traced by the tracer program, system call hooking in the target program, and randomization processing. Randomization processing involves the IPC and the inserted program to obtain the randomizer and some processing for randomization. In the additional experiment, `NUM_THREADS` was fixed at one and `NUM_LOOPS` was fixed at 500,000, and we measured the execution times of the program to which only `ptrace` and system call hooking were attached, (without randomization), respectively.

Table 1: Number of system calls to be randomized and execution time.

	1	2	4	8	16
original	0.068 sec	0.068 sec	0.068 sec	0.068 sec	0.068 sec
original + `ptrace` + hooks	5.11 sec	5.11 sec	5.11 sec	5.11 sec	5.11 sec
original + system call MTD	15.0 sec	20.1 sec	29.6 sec	48.7 sec	88.1 sec

Table1 shows the results of this experiment. The `ptrace` and system call hooks are the overheads that were equally applied to all system calls contained in the original target program. Therefore, the overhead when applying the system call MTD to a certain program is expressed by the following formula:

$$\begin{aligned} overhead = (ptrace\_OH + hooks\_OH) \times num\_syscalls \\ + randomization\_OH \times num\_randomized\_syscalls \end{aligned}$$

Where $*\_OH$ denotes the overhead per system call. From this formula and Table1, we obtain the formula below. $*\_n$ denotes the number of system calls to be randomized.

$$\begin{aligned} ptrace\_1\_OH + hooks\_1\_OH &= \frac{5.11 - 0.068}{500000} = 1.01 \times 10^{-5}\,(\text{sec}) \\ randomization\_1\_OH &= \frac{15.0 - 5.11}{500000} = 1.98 \times 10^{-5}\,(\text{sec}) \end{aligned}$$

$$\begin{aligned} ptrace\_2\_OH + hooks\_2\_OH &= \frac{5.11 - 0.068}{500000} = 1.01 \times 10^{-5}\,(\text{sec}) \\ randomization\_\_OH &= \frac{20.1 - 5.11}{500000} = 3.00 \times 10^{-5}\,(\text{sec}) \end{aligned}$$

$$\begin{aligned} ptrace\_4\_OH + hooks\_4\_OH &= \frac{5.11 - 0.068}{500000} = 1.01 \times 10^{-5}\,(\text{sec}) \\ randomization\_4\_OH &= \frac{29.6 - 5.11}{500000} = 4.90 \times 10^{-5}\,(\text{sec}) \end{aligned}$$

$$\begin{aligned} ptrace\_8\_OH + hooks\_8\_OH &= \frac{5.11 - 0.068}{500000} = 1.01 \times 10^{-5}\,(\text{sec}) \\ randomization\_8\_OH &= \frac{48.7 - 5.11}{500000} = 8.73 \times 10^{-5}\,(\text{sec}) \end{aligned}$$

$$ptrace\_16\_OH + hooks\_16\_OH = \frac{5.11 - 0.068}{500000} = 1.01 \times 10^{-5}\,(\text{sec})$$
$$randomization\_16\_OH = \frac{88.1 - 5.11}{500000} = 1.66 \times 10^{-4}\,(\text{sec})$$

From this formula and Table1, it was found that a non-randomized system call incurs an overhead of 1.01×10^{-5}s per call, while for randomized system calls, the overheads per call were determined to be 1.98×10^{-5}s, 3.00×10^{-5}s, 4.90×10^{-5}s, 8.73×10^{-5}s, and 1.66×10^{-4}s when the number of randomized system calls was 1, 2, 4, 8, and 16, respectively. In summary, the results are as shown in Figure13. The graph indicates that the increase in overhead becomes less significant as the number of randomized system calls increases.

9 Future Work

The main problem we have to solve is the removal of the limitations in Section 7. Among them, it is important to apply the system call MTD to dynamically linked programs. This will allow our method to support almost all the existing programs.

Implementing the tracer program using `ptrace` introduces performance overhead in the execution time of system calls and parallel threads. Improving the implementation and design to reduce the overhead will make this method more practical.

10 Conclusion

In this study, we proposed an MTD for system call mapping as a defense against code injection attacks. Our method can be applied to almost all existing statically linked compiled programs.

Our method consisted of a tracer program and a target program. The target program is the system to which the method is applied, and the tracer program traces the system calls of the target program and provides randomization to the target program. Randomization is performed many times at runtime to prevent an attacker from obtaining information about randomization by investigation. The tracer program uses IPC to send a randomized system call table to the target program.

In the experiment, we applied a system call MTD to the program *smashme* containing the vulnerabilities. Consequently, the exploit could not execute shell programs. Additionally, we conducted an experiment applying the system call MTD to Bash containing the *ShellShock* vulnerability. In this Bash, commands using BusyBox with the system call MTD were successful, whereas commands using external system calls failed. These results indicate that our method can safely execute programs that include code injection vulnerabilities. In other experiments, we measured the performance overhead in the execution time of the system call MTD applied by our method. As a next step, we should reduce this overhead for our method to be practical.

Acknowledgment

This research was supported by JST K Program (Grant Number: JPMJKP24K3) and Hitachi Systems, Ltd.

References

[1] National Institute of Standards and Technology (NIST). National Vulnerability Database. `https://nvd.nist.gov/`. (Accessed: 2023-02-10).

[2] Microsoft. Data Execution Prevention. `https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention`, May 2018. (Accessed: 2023-02-10).

[3] Gaurav S Kc and Angelos D Keromytis. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 15–pp. IEEE, 2005.

[4] Arjan van de Ven. New security enhancements in red hat enterprise linux v. 3, update 3. *Raleigh, North Carolina, USA: Red Hat*, 2004. (Accessed: 2023-02-10).

[5] Ștefan Nicula and Răzvan Daniel Zota. Exploiting stack-based buffer overflow using modern day techniques. *Procedia Computer Science*, 160:9–14, 2019.

[6] Dion Blazakis. Interpreter exploitation: Pointer inference and JIT spraying. *BlackHat DC*, 2010.

[7] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[8] Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, and X Sean Wang. *Moving target defense: creating asymmetric uncertainty for cyber threats*, volume 54. Springer Science & Business Media, 2011.

[9] Sailik Sengupta, Ankur Chowdhary, Abdulhakim Sabur, Adel Alshamrani, Dijiang Huang, and Subbarao Kambhampati. A survey of moving target defenses for network security. *IEEE Communications Surveys & Tutorials*, 22(3):1909–1941, 2020.

[10] Jafar Haadi Jafarian, Ehab Al-Shaer, and Qi Duan. Openflow random host mutation: transparent moving target defense using software defined networking. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 127–132, 2012.

[11] Lynette Qu Nguyen, Tufan Demir, Jeff Rowe, Francis Hsu, and Karl Levitt. A framework for diversifying windows native APIs to tolerate code injection attacks. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 392–394, 2007.

[12] Zhaohui Liang, Bin Liang, and Lupin Li. A system call randomization based method for countering code injection attacks. In *International Conference on Networks Security, Wireless Communications and Trusted Computing, NSWCTC*, pages 584–587, 2009.

[13] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. 2002.

[14] Xuxian Jiang, Helen J Wangz, Dongyan Xu, and Yi-Min Wang. Randsys: Thwarting code injection attacks with system service interface randomization. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 209–218. IEEE, 2007.

[15] Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing*, 8(4):588–601, 2011.

[16] Legitimate Business Syndicate. smashme. `https://github.com/legitbs/quals-2017/tree/master/smashme`, 2017. GitHub repository.

[17] MITRE. CVE-2014-6271. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271`, 2014. (Accessed: 2024-05-28).

[18] PaX Team. Address Space Layout Randomization (ASLR) Documentation. `https://pax.grsecurity.net/docs/aslr.txt`, 2003.

[19] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[20] Jonathan Ganz and Sean Peisert. *ASLR: How robust is the randomness?* IEEE, 2017.

[21] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, 2003.

[22] San Du, Hui Shu, Fei Kang, Xiaobing Xiong, and Zheng Wang. Hardware-based instruction set randomization against code injection attacks. In *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pages 1426–1433. IEEE, 2017.

[23] Takeshi Masumoto, Wai Kyi Kyi Oo, and Hiroshi Koide. MTD: Run-time System Call Mapping Randomization. In *2021 International Symposium on Computer Science and Intelligent Controls (ISCSIC)*, pages 257–263. IEEE, 2021.

[24] Zhidong Shen and Weiying Chen. A Survey of Research on Runtime Rerandomization Under Memory Disclosure. *IEEE Access*, 7:105432–105440, 2019.

[25] Filippo Valsorda. The linux csprng is now good. `https://words.filippo.io/dispatches/linux-csprng/`, Feb 2020. (Accessed:2025-03-03).

[26] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary Rewriting without Control Flow Recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 151–163, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Xiaozhu Meng and Barton P. Miller. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 24–35, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.

[29] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *NDSS*, 2018.

[30] Google. Google Chrome. `https://www.google.com/chrome/`. (Accessed: 2023-02-10).

[31] Mozilla. Firefox. `https://www.mozilla.org/`. (Accessed: 2023-02-10).

[32] Ulrich Drepper. Elf handling for thread-local storage. Technical report, Technical report, Red Hat, Inc., 2003.

[33] Thomas Plum. *C11: The New C Standard*, 2013.

[34] Ebrahim Malalla. Two-way hashing with separate chaining and linear probing. 2005.

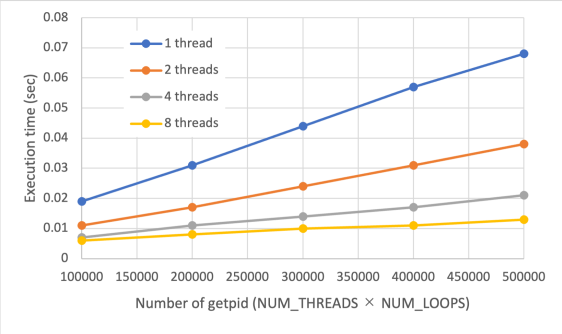
[35] Shell-Storm. Linux/x86-64 - Execute /bin/sh - 27 bytes. `http://shell-storm.org/shellcode/files/shellcode-806.php`. (Accessed: 2023-02-10).

[36] EXALAB. Busybox-static. `https://github.com/EXALAB/Busybox-static`. (Accessed: 2024-06-04).

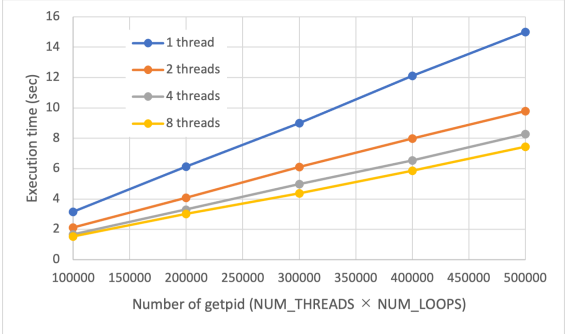
[37] BusyBox Team. Busybox: The swiss army knife of embedded linux. `https://git.busybox.net/busybox`. (Accessed: 2024-06-04).

Algorithm 1 Outline of tracer program processing.

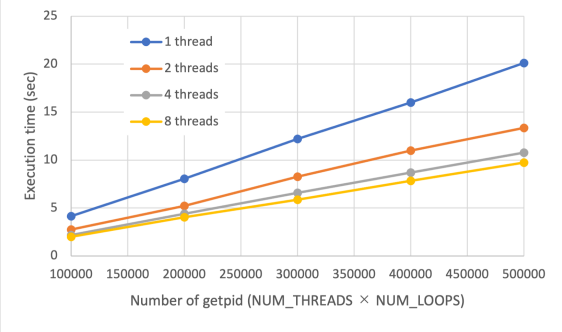
```
1: function MAIN()
2:     pid ← FORK()
3:     if pid = 0 then
4:         EXECV(argv)
5:         return
6:     end if
7:     PTRACE(ATTACH, pid)
8:     PTRACE(SYSCALL, pid)
9:     CREATE_RANDOMIZER(pid)
10:    UPDATE_AND_SEND_SYSTABLE(pid)
11:    loop
12:        pid, status ← WAITPID(−1, __WALL)
13:        switch status do
14:            case PTRACE_SYSCALL
15:                regs ← PTRACE(GETREGS, pid)
16:                if IS_ENTERSTOP(regs) then
17:                    if IS_RANDOMIZED(pid, regs) then
18:                        regs ← DERANDOMIZE(pid, regs)
19:                        PTRACE(SETREGS, regs)
20:                    else if IS_ATTACK(pid, regs) then
21:                        regs ← INVALIDATE(regs)
22:                        PTRACE(SETREGS, regs)
23:                    end if
24:                else
25:                    if IS_FORK(regs) then
26:                        newprocess_pid ← GETRAX(regs)
27:                        CREATE_RANDOMIZER(newprocess_pid)
28:                        UPDATE_AND_SEND_SYSTABLE(newprocess_pid)
29:                    end if
30:                    if WAS_RANDOMIZED(pid, regs) then
31:                        UPDATE_AND_SEND_SYSTABLE(pid)
32:                    end if
33:                end if
34:                PTRACE(SYSCALL, pid)
35:            case EXITED
36:                DELETE_RANDOMIZER(pid)
37:                if NUMBER_OF_RANDOMIZERS() = 0 then
38:                    return
39:                end if
40:        end switch
41:    end loop
42: end function
```



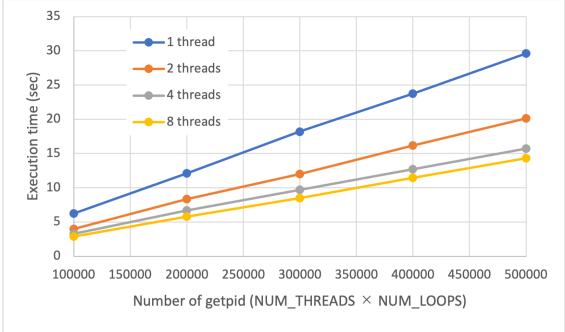
(a) Performance without system call MTD.



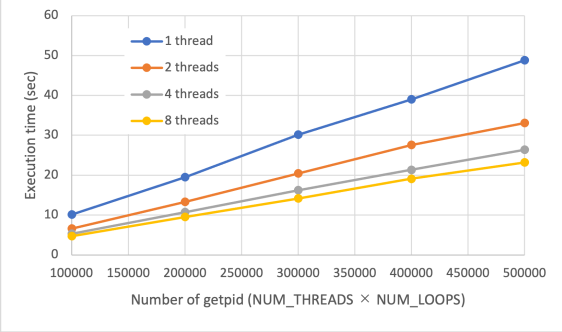
(b) Performance with 1 system call randomized.



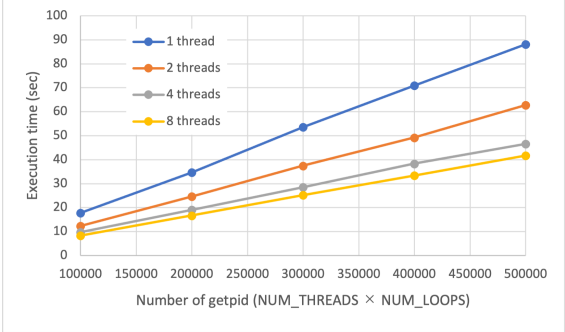
(c) Performance with 2 system calls randomized.



(d) Performance with 4 system calls randomized.

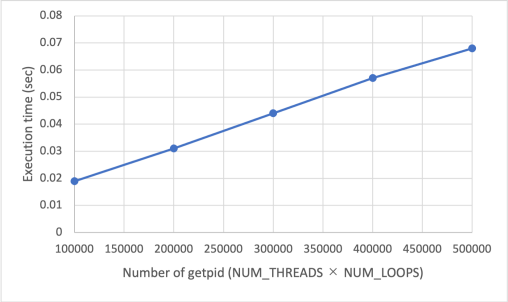


(e) Performance with 8 system calls randomized.

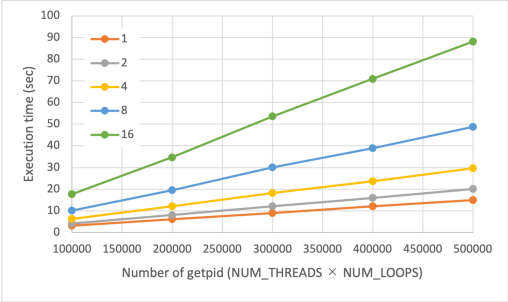


(f) Performance with 16 system calls randomized.

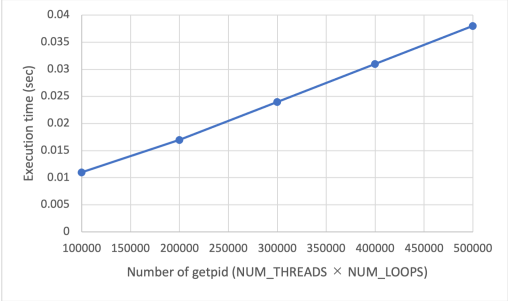
Figure 11: Comparing the execution time without and with system call MTD for different numbers of randomized system calls.



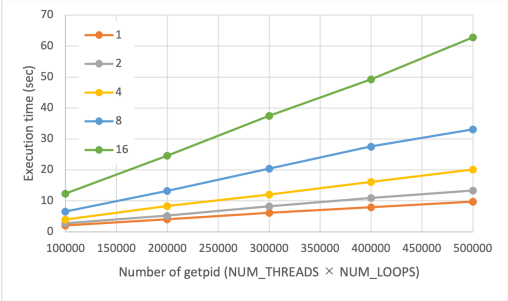
(a) Performance without MTD for 1 thread.



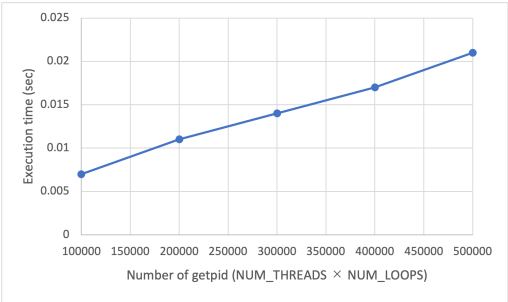
(b) Performance with MTD for 1 thread.



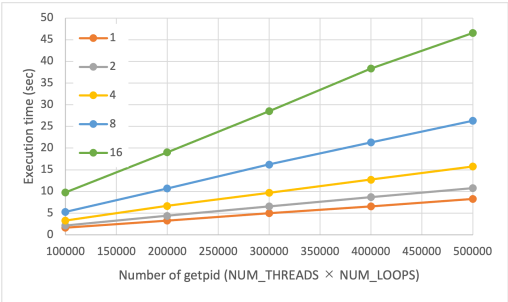
(c) Performance without MTD for 2 threads.



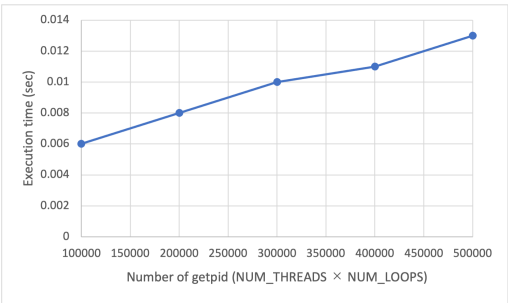
(d) Performance with MTD for 2 threads.



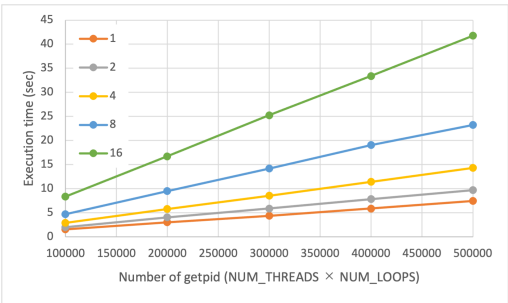
(e) Performance without MTD for 4 threads.



(f) Performance with MTD for 4 threads.



(g) Performance without MTD for 8 threads.



(h) Performance with MTD for 8 threads.

Figure 12: Comparing the execution time without and with system call MTD for various thread counts.

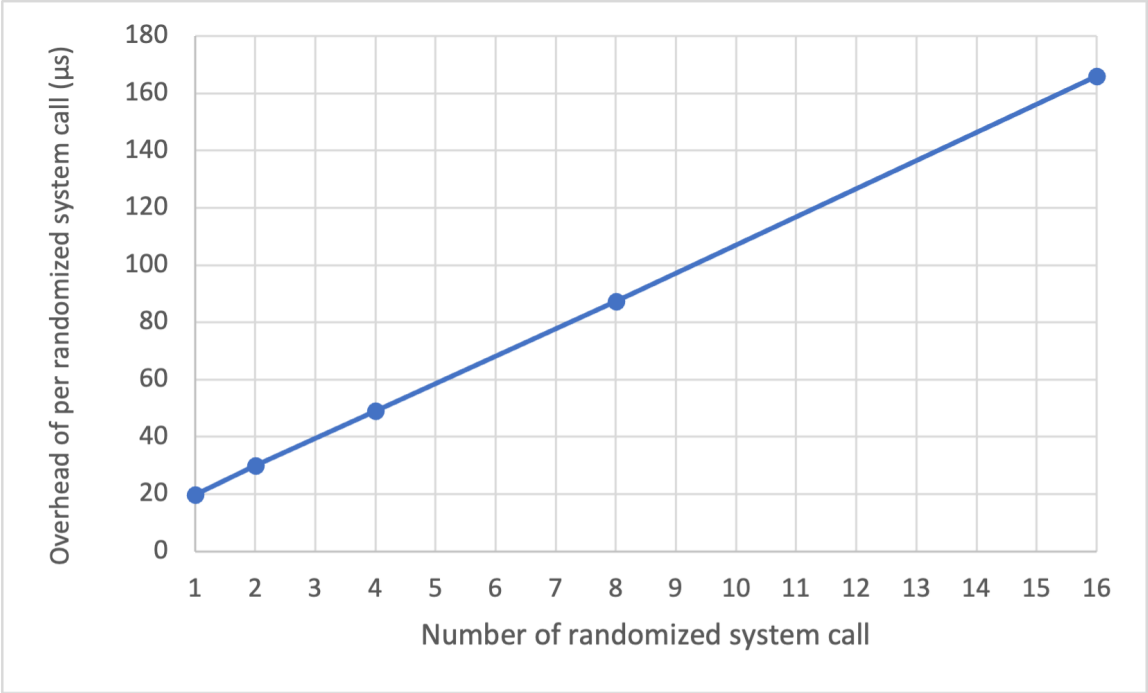


Figure 13: Number of system calls to be randomized and overhead per system call to be randomized.