Dynamic-prelink: An Enhanced Prelinking Mechanism without Modifying Shared Libraries

Hyungjo Yoon[1,2], **Changwoo Min**[1], **and Young Ik Eom**[2]
[1]Samsung Electronics, Suwon, Gyeonggi-do, Korea
[2]Sungkyunkwan University, Suwon, Gyeonggi-do, Korea

Abstract— *Prelink accelerates the speed of program startup by fixing the base address of shared libraries. However, prelink prevents the dynamic linker from loading shared libraries by using Address Space Layout Randomization (ASLR) in runtime because it modifies the program header in binary files directly.*

To resolve this problem, we introduce an enhanced prelinking mechanism, called dynamic-prelink, which separates the memory address layout per program as well as keeps high performance of prelink mechanism. Dynamic-prelink records prelinked contents to a file instead of modifying shared libraries. This makes dynamic linker be able to use both ASLR and prelink mechanism. Our experimental results show that the memory address layout of dynamic-prelinked programs is separated per program and the dynamic linker is able to randomly load shared libraries regardless of dynamic-prelinking. In addition, the startup time of dynamic-prelinked program becomes faster than common program in the dynamic linker, about 42% on average.

Keywords: Dynamic linker, Prelink, ASLR, Shared library, Program startup

1. Introduction

The dynamic linker loads shared libraries and executables, and relocates the memory address layout of each binary. Since it affects every program on the system, dynamic linking is critical in speeding up the program startup.

Prelink accelerates the booting time and program launching time in various operating systems. Jelínek[1] found that prelink reduces processing time of dynamic linking by about 83%, when GTK+ applications are evaluated. In Android, which is most popular system in mobile devices, prelink reduces the booting time by 5%, in practice [2].

But prelink has significant drawbacks. Prelink and *Address Space Layout Randomization* (ASLR) cannot be used simultaneously on a system [2][3][14][16]. ASLR randomizes the layout of memory including stack, heap, library, and executable for enhanced security level of system. According to this, it is difficult for an attacker to expect the randomized address of processes, and so attacks can be defeated ultimately. ASLR is valuable for defending control flow hijacking attacks and *return-to libc* (RTL) attacks [2][4][5][12].

In order to enhance the secure execution, the recent operating systems adapt ASLR rather than applying prelink. Android supported prelink up to the version of Ice-cream 4.0, but it supports ASLR in current version [10]. Several linux's distributions support PaX's implementation of ASLR by default [15]. Windows supports ASLR from Windows Vista [11], and Mac OS X and iOS supported the preload mechanism similar to prelink in the past, but ASLR is supported from the version of Mac OS X 10.8 and iOS 4.3 in the entire system [8][9].

Prelink is partially adopted per program or it is adopted in the whole system. In case that prelink is adapted per program, prelinked program should not influence the launching procedure of the other programs. But a prelinked program prevents the dynamic linker from loading common programs using ASLR on the operation system due to modifying shared libraries directly. Prelink switches the base address, which is described as PT_LOAD segment in program header, from zero to a new value [1]. Thereafter, the dynamic linker let mmap() load libraries into arbitrary address in the memory by referencing PT_LOAD segment. If PT_LOAD is not equal to zero and the indicated address is allocable memory space, the kernel allocates a binary in virtual address of PT_LOAD [6][7]. Therefore, the dynamic linker cannot always load shared libraries related to prelinking in random address if a prelinked program exists on the system. For example, we assume that a program is already prelinked. The dynamic linker always loads libc.so in the same address for every process due to fixed libc.so's program header, and then applying ASLR is limited. This causes a security problem under RTL attacks in which attacker injects malicious codes into the fixed address of libc.so [4].

To resolve this problem, we introduce novel prelinking mechanism, called *dynamic-prelink*, which separates the memory address layout per program as well as keeps high performance of prelinking mechanism. Our scheme separates the memory address layout per program and records prelinked contents to a file instead of modifying shared libraries. We make the following contributions in this paper.

- *Young Ik Eom is the corresponding author of this paper.*
- *This research was supported by the MSIP (Ministry of Science, ICT&Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2014(H0301-14-1020)) supervised by the NIPA (National IT Industry Promotion Agency).*

- We find an efficient mechanism that maintains the performance of prelink and supplements prelink's weaknesses.
- We design an enhanced prelinking mechanism to perform prelinking without modifying shared library.
- We discuss the challenges to handle the separate memory address layout for each prelinked program.

We evaluate our dynamic-prelinking mechanism by comparing the performance of dynamic linker during starting each application which applies dynamic-prelink, ASLR, and original prelink. We get a result that dynamic-prelink is better than ASLR, about 42% on average. Moreover, we also check the memory address layout of the loaded program that is generated by the dynamic-prelinking mechanism to prove that programs can be loaded using both dynamic-prelink and ASLR on a system.

In the rest of the paper, we analyze the overhead of dynamic linker and the effect of prelink in Section 2. Section 3 introduces the design idea of dynamic-prelink, and we address the implementation of dynamic-prelink in Section 4. Section 5 evaluates the implementation and discusses the effect of dynamic-prelink. Section 6 concludes and discusses future works.

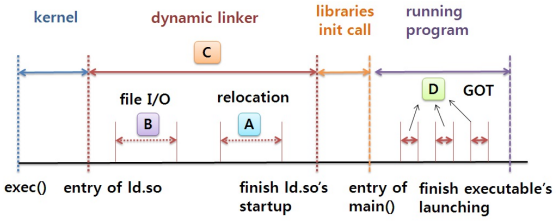

Fig. 1: Execution time during executable's launching.

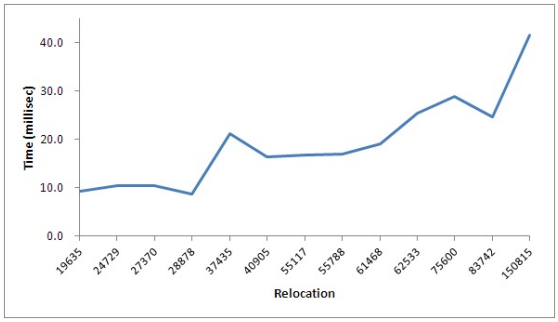

Fig. 2: Spent time in the dynamic linker according to the number of relocations.

2. Analysis of the dynamic linker and prelink

Most of time in the dynamic linker is spent in file I/O, relocations handling, and symbol lookups to load binaries

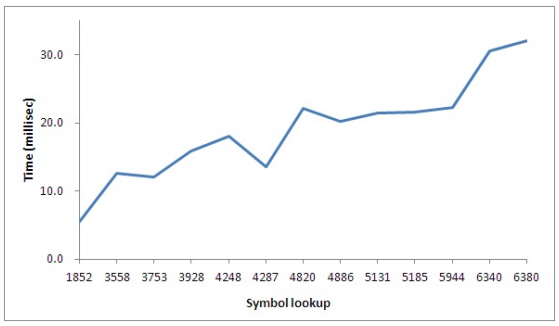

Fig. 3: Spent time in the dynamic linker according to the number of symbol lookups.

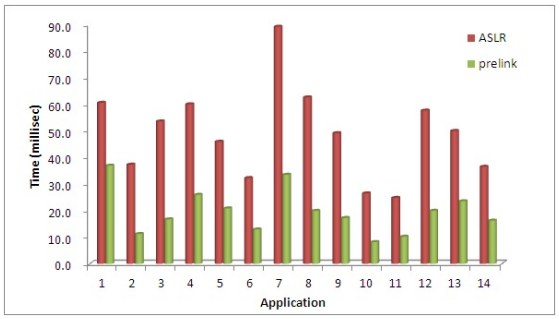

Fig. 4: Comparison of spent time in the dynamic linker during prelinked executable's launching and original executable's launching.

such as libraries and executables [1]. Every time, this operation is applied to memory pages which is written to be loaded into the memory when program is started through exec() in the dynamic linker [1]. The dynamic linker is hard to enhance the performance by controlling file I/O, but relocations handling and symbol lookups are able to be handled by the dynamic linker. So developing algorithm to reduce spent time of relocations handling and symbol lookups is rather efficient to enhance the performance of dynamic linker. Increase in the number of relocations and the number of shared libraries make the dynamic linker spend more time to search symbol scope and do symbol lookups [1]. Additional factor increasing cost is the length of symbol names mangled by C++. It makes the dynamic linker spend more time to find symbols due to increasing cost of comparing symbols. GUI programs become more and more important in recent most of the desktop platforms and the mobile platforms. Additionally, the complexity of program is also increasing. It means future programs will contain more libraries, larger relocations, more symbols and longer the length of symbol [1].

But there is limitation to enhance the performance of dynamic linker although efficient algorithm is designed. It cannot make the number of relocations and the number of symbol lookups be decreased. In terms of same ELF

binaries, it is hard to reduce the number of relocations and the number of symbol lookups.

Both Figure 2 and Figure 3 show how the number of relocations and the number of symbol lookups influence the performance of dynamic linker in Intel i3 dual core 1.2GHz. Those show that spent time in the dynamic linker linearly increases according to enlarging the number of relocations and the number of symbol lookups. Prelink executes relocation handling and the *Global Offset Table* (GOT) resolving in advance to enhance effectively the performance of dynamic linker. Prelink is able to complete relocations except some relocation related dlopen() and conflicted information [1]. It is absolutely better method in terms of only the performance. To verify the performance of prelink in current computing environment, we tried to compare the performance of prelinked program with original program. We divided the main activity of dynamic linker up into two types of sections: spent time in the dynamic linker before calling executable's main() (Figure 1c) and spent time for symbol lookup to resolve the GOT during starting the program (Figure 1d). Startup time (Figure 1c) in the dynamic linker is almost spent for file I/O (Figure 1b) and relocations handling (Figure 1a). We assumed that the total of spent time in the dynamic linker during starting the program is the sum of spent time of two type of sections (startup time in the dynamic linker and symbol lookup time). Figure 4 shows the speed of dynamic linker that loads prelinked program is faster than the speed for original program about 60% on average and the performance is enhanced in all of tested 14 applications. As a result, prelink is still effective method in terms of the performance of dynamic linker.

Table 1: The ratio of sections related the relocation in PIC

	libc.so(byte)	**libstdc++.so(byte)**
(A) Modified area	60260	104460
(B) Whole loaded area	1724620	904164
(C) Ratio (B/A)*100	3.4%	11.5%

3. Design of the dynamic-prelink

Most shared libraries are generated to the *position-independent code* (PIC) ELF. The objective of PIC is to maximize sharing the code of shared libraries and to save the memory space. The section of ELF binary built in the PIC is divided up into three types of sections; *the location-independent section*, *the location-sensitive section*, and *the relocation section* [13]. The relocation section holds information related the location-sensitive section. The dynamic linker adjusts the location-sensitive section using the relocation section during starting the program. Prelink finishes relocating operation by adjusting shared library's sections related the relocation ahead of time. It helps the dynamic linker save time in runtime because relocation and

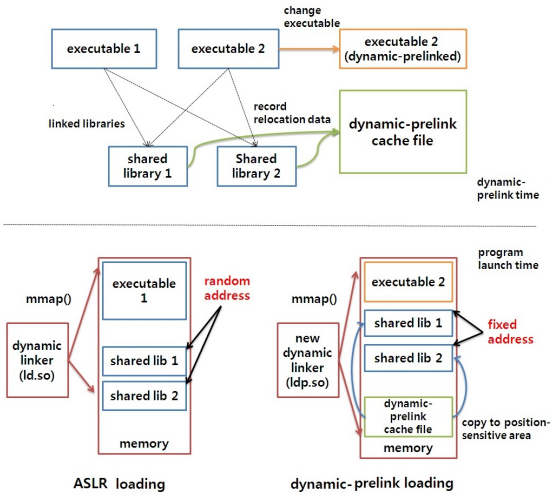

Fig. 5: The whole architecture of dynamic-prelink.

resolving the GOT are already finished. Table 1 shows the ratio of shared library's sections that are modified by prelink. The ratio of modified sections is not greater in and out 10% although there is difference depending on the property of shared library.

We get an idea from what the ratio of relocated section is very small. Our main idea is to record a relocated data to a new binary file instead of directly modifying shared libraries at prelinking time (Figure 5). So we design a new cache file, named *dynamic-prelink cache file*, to record relocated data per program. Dynamic-prelink can bring two benefits, compared with prelink. First, dynamic-prelink can create independent and random address layout per program although several programs are prelinked. Second, programs are not adopted by dynamic-prelink can be loaded normally using ASLR because dynamic-prelink do not modify shared libraries.

To support this idea, additional function is needed in the dynamic linker. So we design a new dynamic linker, called *ldp.so*, has the same function with original dynamic linker as well as new function supporting our idea. ldp.so is able to decode dynamic-prelink cache file and copy relocated data into the memory during the program startup. Moreover, ldp.so can distinguish dynamic-prelinked programs using dynamic-prelink cache file and determine the method for program loading (dynamic-prelink or ASLR) in the runtime.

4. Implementation

To verify the feasibility of dynamic-prelinking mechanism, we implemented dynamic-prelink and a new dynamic linker in the ubuntu 12.04. We contribute that dynamic-prelink independently creates the memory address layout of each prelinked program. It supports that non-prelinked program can be loaded using ASLR. Moreover, the new dynamic linker is developed to support dynamic-prelinking

mechanism. We start with a discussion about implementation for the dynamic-prelinking mechanism and the new dynamic linker.

4.1 Dynamic-prelinking

Dynamic-prelink has two important points about implementation. First, dynamic-prelink cache file is created to help dynamic linker load the program fast. Dynamic-prelink collects and records a prelinked data, called a *cached data*, to dynamic-prelink cache file when it executes prelinking. ldp.so copies the cached data to proper location using dynamic-prelink cache file during starting the program. Second, dynamic-prelink randomly makes the memory address layout per program. It is possible that dynamic-prelink does not modify shared libraries directly. So the base address of shared libraries is assigned randomly, and the memory address layout of each dynamic-prelinked program becomes unique on the system.

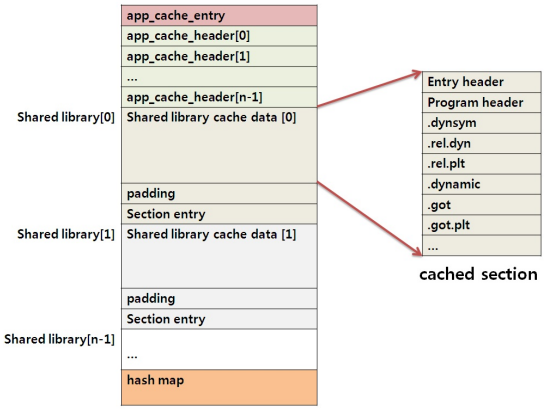

Fig. 6: The format of *dynamic-prelink cache file*.

Table 2: The list of *cached sections*

Section name	Type
.dynsym	SHT_DYNSYM
.rel.dyn	SHT_REL
.rel.plt	SHT_REL
.dynamic	SHT_DYNAMIC
.got	SHT_PROGBITS
.got.plt	SHT_PROGBITS
.data	SHT_PROGBITS
.data.rel.ro	SHT_PROGBITS
.__libc_thread_subfreeres	SHT_PROGBITS
.__libc_atexit	SHT_PROGBITS
.__libc_subfreeres	SHT_PROGBITS
.init_array	SHT_INIT_ARRAY
.fini_array	SHT_FINI_ARRAY
.tdata	SHT_PROGBITS

4.1.1 Dynamic-prelink cache file format

dynamic-prelink cache file is a binary which is recorded by the cached data of the location-sensitive section and the relocation section. This file is designed to manage the list of dependent shared libraries of the program and the cached data of each shared libraries. Figure 6 shows how dynamic-prelink cache file format is designed. It is composed to the field of *app_cache_entry*, *app_cache_entry*, *app_cache_header*, *section_header*, *cached section*, and *hash map*. The characteristic of each field is as follows.

- *app_cache_entry*: This structure gives the offset of first index of app_cache_header, the size of app_cache_header, and the number of app_cache_header. It helps dynamic linker find all of index of app_cache_header.
- *app_cache_header*: This structure gives a cached object of each shared library. Each object gives the offset of root header for cached sections and the number of cached sections, those help dynamic linker find all of index of section_header.
- *section_header*: This structure gives relative offset for the cached data and the copied virtual address. The format of structure is the same with *Elfxx_Shdr* of ELF.
- *cached section*: This is the set of cached data which is recorded by dynamic-prelink ahead of time. The dynamic linker can directly copy it into the memory instead of relocating operation.
- *hash map*: This element holds the offset of app_cache_header hash table to search fast object related shared object.

We try to distinguish the type of sections to select the cached data. The list of location-sensitive sections and the list of relocation sections are found by comparing original shared library with prelinked shared binary. Both *vimdiff* and *objdump*, which are utilities, are used to find the list of relocated sections. Table 2 shows the list of location-sensitive sections and relocation sections. Mainly, those sections are recorded to the cached data of dynamic-prelink cache file.

The dynamic linker does not adjust both position independent sections and irrelevant-relocation sections such as *.text*, *.eh_frame*. But common executable is not built in PIC, is the type of ET_EXEC. It means that all sections of executable are able to be adjusted by the dynamic linker during starting the program. It is impossible to keep the cached data to dynamic-prelink cache file. But the address of ET_EXEC's program header is fixed during the link processing at the compile time, the executable is loaded at a known location always. In other words, the executable is originally impossible to be loaded by using address randomization. So dynamic-prelink directly modifies the executable

in dynamic-prelinking time because it does not affect the other programs.

```
[ 7] .gnu.version_r    VERNEED          0003cbac 03cbac 000120 00
[ 8] .rel.dyn          REL              0003cccc 03cccc 0051c8 08
[ 9] .rel.plt          REL              00041e94 041e94 0014b8 08

[ 7] .gnu.version_r    VERNEED          43dccbac 03cbac 000120 00
[ 8] .rel.dyn          RELA             43dccccc 03cccc 007aac 0c
[ 9] .rel.plt          REL              43dd4778 044778 0014b8 08
```

Fig. 7: The size of *.rel.dyn* section is enlarged 1.5 times before and after prelinking due to changing REL to RELA and the relative offset of *rel.plt* is influenced by RELA.

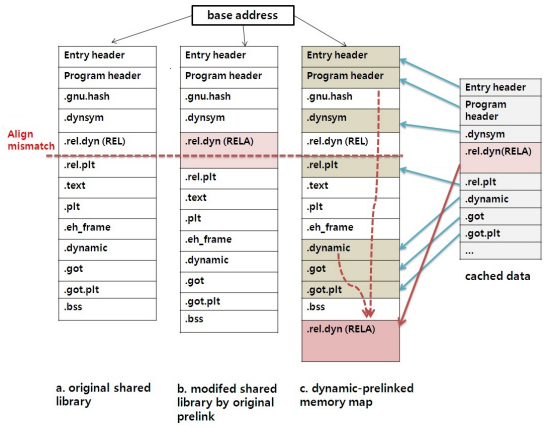

Fig. 8: Comparison with the memory address layout according to the loading mechanism. The location of RELA section is moved at the last address of loaded memory space of each shared libraries.

4.1.2 RELA section

we start with assumption that the data segments to be page-aligned and relative offset of shared library are identical before and after prelinking. If there becomes right assumption, the size of sections, which are loaded in the memory, is coincided before and after prelinking. But, in some cases, prelink changes DT_REL to DT_RELA which is the property of relocation section. RELA makes relocating operation be easy because it does not need the relocated contents in the memory [1]. *Elfxx_Rela* contains more a member which is an explicit addend compared with *Elfxx_Rel*, and the size of DT_RELA is 1.5 times larger than DT_REL. Moreover, DT_RELA is commonly located in an intermediate position of the binary file. Figure 7 shows the size of *.rel.dyn* section is enlarged after prelinking, and the relative offset of rel.plt is changed. It is big problem to mechanism of dynamic-prelink due to the page alignment.

But we make one important observation about the relocation section. The relocation section only contains information about which data is relocated in runtime, and it is independent against relative offset in the binary. Namely, it is not important where the relocation section is located. The dynamic linker commonly gets the location of relocation section from PT_GNU_RELRO of program header. A program can be normally operated if the location of relocation section is identical to the address of PT_GNU_RELRO regardless relative offset of relocation section. So we design that dynamic-prelink changes the address of PT_GNU_RELRO to the last address of loaded binary in case of generating DT_RELA. Figure 8 compares the section alignment of each loaded shared libraries (original shared library, prelinked shared library, dynamic-prelinked shared library) into the memory. Prelink makes the size of shared library's *.rel.dyn* be enlarged against original shared library. But dynamic-prelink can maintain identical alignment with original shared library by locating the RELA in the last address of mapped memory.

4.1.3 Randomization of dynamic-prelink

when dynamic-prelink does prelinking for any program, it is possible to randomly determine the base address of shared library using */dev/random*. Dynamic-prelink sets up various prelinked memory address layout per program. Because an address layout of program becomes unique on the system, the randomization of dynamic-prelink makes it more difficult to analysis the loaded address of shared libraries.

4.2 New dynamic linker: ldp.so

The new dynamic linker (ldp.so) supplements additional feature for the dynamic-prelinking mechanism. First, ldp.so is able to decode dynamic-prelink cache file format and copy the cached data to proper address instead of original dynamic linker mechanism. Second, the program, which is not adopted by dynamic-prelink, is loaded normally using ASLR following original dynamic linker mechanism. ldp.so is able to distinguish dynamic-prelinked programs by checking dynamic-prelink cache file. Moreover, in case of adding new rules, ldp.so is able to dynamically determine the loading mechanism (ASLR or dynamic-prelink) in dynamic-prelinked program startup.

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ldp.so]
  LOAD           0x000000 0x08048000 0x08048000 0x00bbd 0x00bbd R E 0x1000
  LOAD           0x000efc 0x08049efc 0x08049efc 0x0272c 0x0272c RW  0x1000
```

Fig. 9: Modified executable's *.interp* by dynamic-prelink.

```
yoon@yoon-X280:~$ sudo cat /proc/14158/maps | grep "ld-2.15.so" & sudo cat /proc/14158/maps | grep "ldp.so"
[9] 14174
00000000-00022000 r-xp 00000000 08:07 4064129    /lib/ldp.so
00022000-00023000 r--p 00021000 08:07 4064129    /lib/ldp.so
00023000-00024000 rw-p 00022000 08:07 4064129    /lib/ldp.so
[9]   Exit 1                  sudo cat /proc/14158/maps | grep --color=auto "ld-2.15.so"
yoon@yoon-X280:~$ sudo cat /proc/14103/maps | grep "ld-2.15.so" & sudo cat /proc/14103/maps | grep "ldp.so"
[9] 14181
b7732000-b7754000 r-xp 00000000 08:07 4063356    /lib/i386-linux-gnu/ld-2.15.so
b7754000-b7755000 r--p 00021000 08:07 4063356    /lib/i386-linux-gnu/ld-2.15.so
b7755000-b7756000 rw-p 00022000 08:07 4063356    /lib/i386-linux-gnu/ld-2.15.so
```

/lib/ldp.so

/lib/i386-linux-gnu/ld-2.15.so

Fig. 10: ldp.so is loaded in running dynamic-prelinked program, but ld.so is loaded in running common program.

When the kernel loads and executes a process in newly constructed address space, the kernel checks the dynamic linker first in executable's *.interp* section. In user-mode, first context of process is started in the entry point of dynamic linker. The dynamic linker is also the shared library (ld.so), but it is hard to control the memory address layout of ld.so using the dynamic-prelinking mechanism. It is because there is no chance to control the memory address layout of ld.so in user-mode. To solve this problem, we make important modification to the dynamic liker and the executable. First, a new dynamic linker (ldp.so) is created without modifying ld.so, and we induce that the system becomes to own two dynamic linkers (ld.so and ldp.so). ldp.so is directly modified in order to complete the relocation by dynamic-prelink in advance. Second, executable's *.interp* section is modified to make the kernel load ldp.so as the dynamic linker. Figure 9 shows executable's *.interp* section is changed to ldp.so by dynamic-prelink. After all, programs, which are not adopted by the dynamic-prelinking mechanism, are loaded by original dynamic linker (ld.so) and dynamic-prelinked programs are loaded by ldp.so as the dynamic linker. Figure 10 shows that two dynamic linkers are able to be loaded according to a mechanism adopted in program.

5. Evaluation

We start with a discussion to evaluate our dynamic-prelinking mechanism by comparing the performance of dynamic linker during starting the program and verifying the memory address layout of program which is generated by dynamic-prelink.

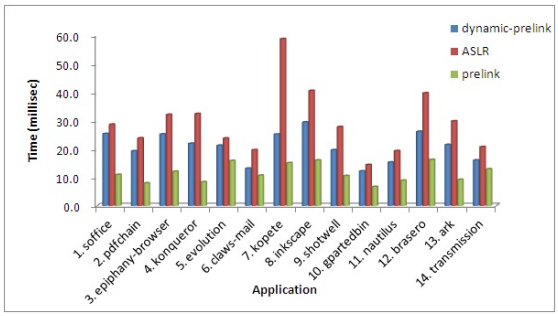

Fig. 11: Spent time before enter executable's main() (Figure 1c) in the dynamic linker during 14 applications startup: original prelinked program, dynamic-prelinked program, original program.

5.1 Performance

We verified the effect of dynamic-prelink by evaluating spent time in the dynamic linker in Intel i3 dual core 1.2GHz. We also divide the main activity of dynamic linker up into two type of sections: spent time in dynamic linker before calling executable's main() (Figure 1c) and spent time in symbol lookups to resolve the GOT during starting

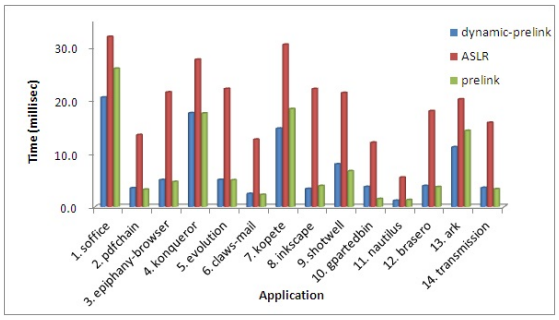

Fig. 12: Spent time to resolve the GOT (Figure 1d) in the dynamic linker during 14 applications startup: original prelinked program, dynamic-prelinked program, original program.

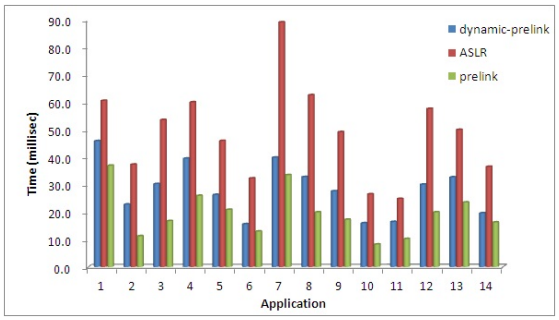

Fig. 13: Whole spent time in the dynamic linker during startup of original prelinked program, dynamic-prelinked program, and original program. This data is the sum of spent time before enter executable's main() (Figure 1c) and spent time to resolve the GOT (Figure 1d).

the program (Figure 1d). We compared the performance of original prelink, ASLR (RTLD_LAZY loading), and dynamic-prelink by adding to dual data (We think this is the performance of dynamic linker during starting the program). Figure 13 shows the speed of the dynamic linker using the dynamic-prelinking mechanism is faster than ASLR, about 42% on average, and it is slower than original prelink, about 32% on average. The performance of dynamic-prelink presents intermediate position between ASLR and original prelink. This result is influenced by dynamic-prelink cache file. Dynamic-prelink needs additional time for file I/O and memory copy to control dynamic-prelink cache file. Figure 11 shows spent time in the dynamic linker to load dynamic-prelinked program (Figure 1c) is increased more than original prelinked program. But we should concentrate on saved time in comparison with ASLR. Dynamic-prelink can save relocating time which is similar to original prelink. So the speed of dynamic-prelink is enhanced in both spent time before calling executable's main() (Figure 11) and spent time to resolve the GOT (Figure 12) in the dynamic linker. Especially, spent time to resolve the GOT of dynamic-prelink is similar to these of original prelink. (Although a program

is prelinked, dynamic linker should resolve the GOT due to dlopen()). Namely, the performance of dynamic-prelink is better than ASLR and is similar to prelink except the overhead of dynamic-prelink cache file.

Table 3: Fixed address layout of shared libraries of two dynamic-prelinked programs. Dynamic-prelinked programs own different address layout each other.

application	library	1st address	2nd address
soffice	ldp.so	0x80000000	0x80000000
	libc.so	0x41967000	0x41967000
	libstdc++.so	0x41fd1000	0x41fd1000
nautilus	ldp.so	0x80000000	0x80000000
	libc.so	0x4b6d5000	0x4b6d5000
	libstdc++.so	0x4be77000	0x4be77000

Table 4: The memory address layout of loaded programs using ASLR in running programs which are adopted by dynamic-prelinking mechanism.

application	library	1st address	2nd address
chrome	ld.so	0xb5efe000	0xb5f3a000
	libc.so	0xaf9f2000	0xafa2e000
	libstdc++.so	0xafbe6000	0xafc22000
firefox	ld.so	0xb7726000	0xb770c000
	libc.so	0xb740a000	0xb73f0000
	libstdc++.so	0xb7619000	0xb75ff000

5.2 Memory address space layout

We verified the memory address space layout of processes based on a dynamic-prelinked programs and non-dynamic-prelinked programs using */proc/[pid]/maps* to prove that two mechanisms (ASLR and dynamic-prelink) are able to be used on a system at the same time (Table 3, 4). And we checked to hold different memory address layout per program which is adopted by the dynamic-prelinking mechanism (Table 3). Table 3 shows libc.so and libstdc++.so are located in different address in running processes based on two dynamic-prelinked programs, and ldp.so is loaded instead of ld.so. It means that dynamic-prelink separates address layout of two programs. ldp.so is always located in the same address due to the effect of dynamic-prelink.

Another important point is that both libc.so and libstdc++.so of non-dynamic-prelinked programs are loaded using ASLR at same time in running processes based on dynamic-prelinked program (Table 4). As a result, the dynamic-prelinking mechanism does not affect the memory address layout of non-dynamic-prelinked program. Besides, the memory address layout of dynamic-prelinked program is separated with it of the other dynamic-prelinked programs.

6. Conclusion

Dynamic-prelink makes the dynamic linker be able to use mechanism of both ASLR and prelink at the same time. Moreover, it helps dynamic-prelinked program be loaded faster than common program in the dynamic linker, about 42% on average. Since dynamic-prelink supports independent prelinking without modifying the shared libraries, a dynamic-prelinked program does not affect the memory address layout of the other programs. And dynamic-prelink prevents exposing process's address space layout to the other programs. In addition, each program is able to be prelinked selectively according to program's importance by combining several security mechanisms to enhance system performance.

In this paper, we do not discuss *position independent executable* (PIE) for dynamic-prelink. Original prelink cannot support PIE prelinking due to modification of binary. But we have a plan to adapt dynamic-prelink to PIE. Because dynamic-prelink does not modify shared libraries, we expect that PIE can be prelinked by dynamic-prelink. In addition, we will solve a issue about the fixed base address of new dynamic linker (ldp.so). We anticipate that ldp.so is loaded into random address as well as dynamic linkers (ld.so, ldp.so) can be unified to one component by modifying kernel's exec() mechanism.

References

[1] Jelinek, Jakub. Prelink. Technical report, Red Hat, Inc., 2004. available at http://people.redhat.com/jakub/prelink. pdf, 2003.

[2] Bojinov, Hristo, et al. "Address space randomization for mobile devices."Proceedings of the fourth ACM conference on Wireless network security. ACM, 2011.

[3] van Veen, Sander Mathijs. "Concurrent Linking with the GNU Gold Linker." (2013).

[4] Shacham, Hovav, et al. "On the effectiveness of address-space randomization."Proceedings of the 11th ACM conference on Computer and communications security. ACM, 2004.

[5] Spengler, Brad. "Pax: The guaranteed end of arbitrary code execution." (2003).

[6] Loosemore, Sandra, et al. The GNU C libraryreference manual. Free software foundation, 2001.

[7] Chamberlain, Steve, and Ian Lance Taylor. "Using ld: the GNU Linker." (2003).

[8] "Apple OS X Mountain Lion Core Technologies Overview". June 2012. Retrieved 3 December 2013.

[9] Dai Zovi, Dino A. "Apple iOS 4 security evaluation." Black Hat USA (2011).

[10] "Android Security". Android Developers. Retrieved 9 December 2013.

[11] Windows, I. S. V. "Software Security Defenses." (2012).

[12] Payer, Mathias. "Too much PIE is bad for performance." (2012).

[13] Tool Interface Standards Committee. "Executable and Linkable Format (ELF)." Specification, Unix System Laboratories (2001).

[14] John Moser. Prelink and address space randomization, 2006. http://lwn.net/Articles/190139/.

[15] De Raadt, Theo. "Exploit mitigation techniques." (2005).

[16] Xu, Haizhi, and Steve J. Chapin. "Improving address space randomization with a dynamic offset randomization technique." Proceedings of the 2006 ACM symposium on Applied computing. ACM, 2006.