Gadge Me If You Can

Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM

Lucas Davi[1,2], Alexandra Dmitrienko[3], Stefan Nürnberger[2], Ahmad-Reza Sadeghi[1,2,3]

[1]Intel Collaborative Research Institute for Secure Computing (ICRI-SC) at TU-Darmstadt, Germany

[2]CASED/System Security Lab Technische Universität Darmstadt Darmstadt, Germany

[3]Fraunhofer Institute for Secure Information Technology Darmstadt, Germany

ABSTRACT

Code reuse attacks such as return-oriented programming are one of the most powerful threats to contemporary software. ASLR was introduced to impede these attacks by dispersing shared libraries and the executable in memory. However, in practice its entropy is rather low and, more importantly, the leakage of a single address reveals the position of a whole library in memory. The recent mitigation literature followed the route of randomization, applied it at different stages such as source code or the executable binary. However, the code segments still stay in one block. In contrast to previous work, our randomization solution, called *XIFER*, (1) disperses all code (executable and libraries) across the whole address space, (2) re-randomizes the address space for each run, (3) is compatible to code signing, and (4) does neither require offline static analysis nor source-code. Our prototype implementation supports the Linux ELF file format and covers both mainstream processor architectures x86 and ARM. Our evaluation demonstrates that *XIFER* performs efficiently at load- and during run-time (1.2% overhead).

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

software diversity; randomization; ASLR; return-oriented programming, return-into-libc

1. INTRODUCTION

Security-critical operations such as online banking are increasingly performed by widespread everyday-software. This makes them an appealing target for various attacks, in particular runtime attacks which subject a process to an adversary's control. Albeit control-flow attacks on software are known for about two decades, they are still one of the major threats to software on desktop PCs and mobile devices. The NIST vulnerability database reported 663 buffer errors in 2011, and 724 for 2012 [26]. The broad introduction of non-executable memory, write-xor-execute ($W \oplus X$) for short, successfully mitigates code injection attacks but gave rise to a form of attacks that re-use existing code by intelligently stitching small code fractions, so-called *gadgets*, together in order to execute arbitrary code. These gadgets are well selected so that they end in an instruction that transfers control to the next gadget, e.g. a `ret` (return) instruction which pops its target off the stack. Hence, the name *return-oriented programming* (ROP [31]).

Since these attacks rely on exact addresses of the instructions they want to abuse, Address Space Layout Randomization (ASLR, e.g. [29]) debuted as the next move in that cat-and-mouse game. By randomizing the base address of loaded code and data in memory, ASLR in theory makes it infeasible for an attacker to predict the location of gadgets in memory. However, low entropy [32] and the fact that a single leaked pointer makes it possible to calculate relative addresses called for yet another step of defense. Such a means of defense has come to light in the form of finer and finer code randomization that, in contrast to ASLR, also shuffles the code itself, not just its base address. Despite the fact that such randomization is a simple idea, its implementation is highly involved and several approaches exist in the literature, ranging from compiler-based solutions [6, 11, 18] to run-time solutions [16] that randomize the program either once or even constantly during its lifetime [13].

As we elaborate in Section 3, most of the existing works have at least one of the following drawbacks: they (i) need access to source code, (ii) do not cover the whole address space, e.g. no loaded libraries and the code segment stays in one block, (iii) do not re-randomize at each process start, or (iv) touch the executable file rendering them incompatible to code signing which is prevalent for commercial software and mandatory in modern app stores.

In order to compare and measure existing software diversity methods, we establish a set of properties that make a randomization solution *ideal*, i.e., featuring the best trade-off among these properties. These properties are: (1) mitigation of code reuse attacks (ROP and return-into-libc), (2) high

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.

diversification entropy, (3) efficient re-randomization, (4) required input information (e.g., source code or other side information), (5) code coverage (code parts that cannot be diversified), (6) compliance to code signing, (7) performance, (8) space consumption (memory and disk) (9) shared library support, and (10) generality vs. specificity, i.e. applicability or limit to certain hardware architectures.

To our surprise, recent software diversity and randomization approaches do *not* fulfil these criteria (as shown in the comparison in Table 1). In particular, they randomize the code only within its segment so that it stays as one block, which means a leaked pointer is always surrounded by the remainder of the code. Moreover, most of the existing work avoids re-randomization for each run, probably due to efficiency concerns. Hence, all binaries of a system remain unchanged leading to an increasing advantage for an adversary over time. Further, many existing solutions touch the main executable file which courrupts a potential digital signature that could be in place.

Our Contribution. In this paper, we present a novel tool, called *XIFER* that adequately addresses the aforementioned requirements for an ideal randomization tool:

- It achieves an instruction granularity randomization.
- It randomizes all sections of an executable and library (not just `.text`) and disperses fractions of the executable segments so that they do not stay in one block, changing *all* relative relations of code and data so that leaked pointers cannot be used to calculate relative addresses.
- The randomization takes place on-the-fly – not requiring an offline static analysis – leading to a different address space layout for each process.
- The high randomization performance needed for an on-the-fly solution stems from a technique we call *partial dis- and reassembly*. This technique leverages the fact that most of the instructions do not reference code or data and hence do not need to be dis- or re-assembled.

We evaluated our prototype for x86 and ARM by using the benchmark suite SPEC CPU2006. Our evaluation results (see Section 6) demonstrate that *XIFER* efficiently performs randomization so that the resulting runtime overhead is only 1.2%, and the linear load-time overhead achieves 5500 kBit/s.

2. BACKGROUND

A simplified view of a code reuse attack is shown in Figure 1. It shows an abstract memory layout of a vulnerable application, where an adversary hijacks the intended execution flow of the application by exploiting a memory-related vulnerability on the stack (e.g., a buffer overflow). In step 1, the adversary exploits the vulnerability to inject a number of pointers (Return Address 1 to 3) on the stack (step 1). Each of these addresses point to a certain code sequence residing in the linked libraries or the executable itself. The executable/library code segment contains a number of functions, where each function consists of several so-called basic blocks (BBLs). A BBL is a sequence of machine instructions with a single entry and exit instruction, where the latter one can be any branch instruction the processor supports (e.g., return, indirect/direct jump or call).

After the adversary has subverted the execution flow by, e.g. overwriting the program's return address, the execution is

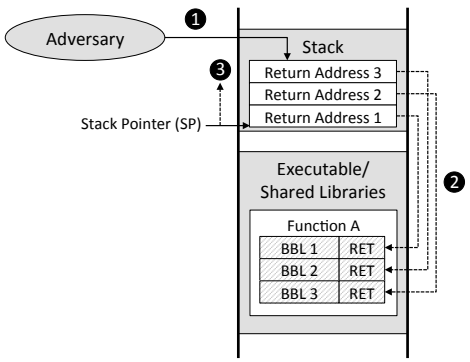

Figure 1: Memory view of a stack-based code reuse attack.

redirected to BBL 1 (step 2). After BBL 1 has executed, the terminating return instruction (RET) pops the next return address off the stack (Return Address 2), thereby increasing the stack pointer by one memory word, and redirects the execution to BBL 3 and so forth.

In practice, an application includes a large number of BBLs. While return-into-libc attacks would target execution of known *functions*, return-oriented programming enables the combination of arbitrary code sequences. Both attack techniques form a Turing-complete basis for building arbitrary (malicious) programs [31, 34]. Since code-reuse attacks have become the standard runtime attack vector against desktop and mobile computing platforms [35, 14, 30, 17, 25, 20, 7, 19], our goal in this paper is to present an efficient mitigation technique that entirely prevents these attacks.

3. RELATED WORK

To mitigate the threat of code reuse attacks, recent proposals apply various randomization techniques. We define a set of properties that we think are vital for secure, effective and efficient randomization and use them to motivate the design and implementation of our approach. In Table 1, these properties are also compared to existing randomization proposals realized as compiler extensions [6, 11, 18, 4], system-wide base address randomization (ASLR) [29], and binary rewriting tools [21, 28, 16, 36].

P1 – Effectiveness Against Code Reuse Attacks. The solution shall mitigate return-into-libc and ROP, since both have been shown to be Turing-complete [31, 34].

P2 – Entropy. A strong randomization scheme must provide enough entropy to render brute-force attacks infeasible and should change relative distances between code and data as well in order to render a leaked pointer futile.

P3 – Randomization Frequency. A randomization tool should be able to re-diversify a program for each execution. Otherwise, an adversary could acquire knowledge about the memory layout of the diversified program by running the program various times and launching brute-force attacks.

P4 – Input Information. The fact that a randomization solutions requires access to source code or to additional information like debug symbols or relocation information.

P5 – Code Coverage. The recent past has shown that only a piece of un-randomized code is in many cases sufficient to launch a code-reuse attack [12]. Hence, randomizing the whole address space is a necessity.

P6 – Compliance to Code Signing. Code signing is a mandatory feature in nearly all modern app stores [24, 15, 9]. A randomization tool should not touch the executable file in order to keep the signature intact.
P7 – Performance. Performance is influenced by the fact that (1) randomized code pieces must be connected to retain their original order and (2) the reduced locality of code is more prone to cache misses.
P8 – Memory and Disk Space. The randomization solution might need a static analysis or caches to be built and maintained in an offline phase or at run-time. A large size of either of these is undesirable.
P9 – Library Support. Libraries exist to ease application development and to save space on disk. As code-reuse attacks typically leverage code residing in libraries, it is crucial that a randomization tool can also be applied to libraries.
P10 – Target Hardware Architectures. A randomization tool should be general enough to be easily ported to another processor architecture.

Compiler-Based Randomization.

The original randomization approach targets and proposes a compiler-based solution [6]. Recently, Franz et al. [11, 18] have explored the feasibility of a compiler-based approach for large-scale software diversity in the mobile market. The authors suggest that app store providers integrate a multicompiler (diversifier) in the code production process. However, this approach has two shortcomings: App store providers have no access to the app source code. This requires the multicompiler to be deployed on the developer side, who has to deliver thousands of different app instances to the app store. Secondly, an app instance gets only randomized once. A former randomization work by Bhatkar et al. [4] does not suffer from this shortcoming, because it deploys a source code transformer that enables re-randomization for each run.

In general, compiler based solutions have the potential to provide among all randomization approaches the highest degree of entropy due to the access to source code. However, as argued above, source code is rarely available in practice and current app store models are not compatible to a multicompiler approach.

Binary Instrumentation Based Randomization.

These techniques directly operate on the application binary to perform code randomization. In particular, Kil et al. [21] introduce address space layout permutation (ASLP). The proposed scheme statically rewrites ELF executables to permute all functions and data objects of an executable. Moreover, the Linux kernel has been modified to increase the entropy for the base address randomization of shared libraries. Although, the presented scheme is efficient and supports re-randomization, it is not directly compatible to code signing, provides a lower randomization entropy compared to instruction or basic block (BBL) randomization, and does not apply code randomization to shared libraries.

Recently, Pappas et al. [28] introduced ORP, a static rewriting tool which randomizes instructions and registers within a BBL to mitigate ROP attacks. However, ORP cannot prevent return-into-libc attacks (which have shown to be Turing-complete [34]), since all functions remain at their original position. In contrast, ILR (instruction location randomization) [16] translates each address to a randomized version while executing in a process virtual machine. For this, a program needs to be analyzed and re-assembled during a static analysis phase which induces significant run-time performance and space overhead (the rewriting rules reserve on average 104 MB for only one benchmark of SPEC CPU). ILR also suffers from code coverage deficiencies due to imprecision of the static analysis phase. This concerns in particular the destination addresses of indirect jumps, indirect calls, and function returns. Specifically, ILR does not attempt to resolve destination addresses of indirect calls [16, 6] allowing an adversary to launch return-into-libc like attacks. Both ORP and ILR cannot re-randomize a binary for each program.

In contrast, STIR [36] randomizes and permutes BBLs for each execution and hence provides a higher randomization entropy compared to ORP and ILR. However, STIR still requires a static analysis and rewriting phase which is not compatible to code signing. Moreover, it suffers from a high space overhead, because the file size of a stirred program increases by 73 % on average.

Finally, Guiffrida et al. [13] propose a fine-grained randomization proposal for operating system kernels. Besides stack, heap and code randomization, it allows re-randomization of a module at runtime but is limited to microkernels.

4. DESIGN

In this section, we first present the design of our randomization tool *XIFER*, after which we elaborate on several technical challenges and present our solutions thereof.

4.1 High-Level Design of XIFER

As already alluded to, the goal of our tool is to fulfill all of the aforementioned properties and criteria by randomizing the *complete* address space for every start of a process. This randomization deliberately tears code apart subjecting relative distances within code to change. This is achieved by randomizing the position of each executable and library segment (such as `.text`, `.init`, `.ctors`, `.data`, `.bss`) in memory and additionally twirling the code so that leaked pointers do not reveal anything about the remainder of the code or data.

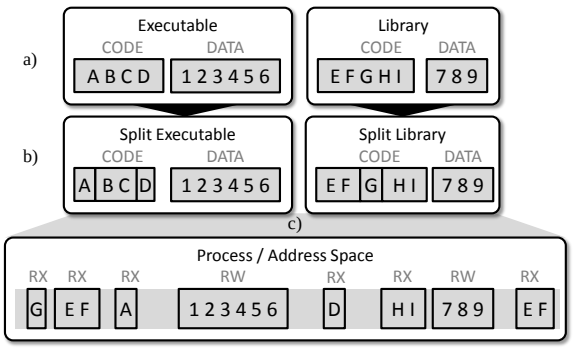

Figure 2: High-level process of the randomization

As depicted in Figure 2, the idea of our randomization approach is to cover the complete executable code of a process, which consists of the executable file itself and all loaded libraries. For that purpose, we apply the necessary randomization steps right before execution of a process starts, but after all the necessary code has been loaded into the address space by the linker (step *a* in Figure 2). In order to randomize the individual code segments and to intermix them

Criteria and Properties	Multicompiler [6, 11, 18]	Source Transformer [4]	ASLR e.g.,[29]	ASLP [21]	ORP [28]	ILR [16]	STIR [36]	*XIFER*
P1 - Effectiveness								
a) ROP	yes	yes	partially	yes	yes	yes	yes	yes
b) Return-into-Libc	yes	yes	partially	yes	no	no	partially	yes
P2 - Entropy	very high	medium	low	medium	medium	high	high	high
P3 - Frequency	one time	many	many	many	one time	many	many	many
P4 - Input Information	source code	source code	reloc	reloc	none	none	none	reloc
P5 - Code Coverage	high	high	high	high	medium	medium	high	high
P6 - Code Signing Comp.	partially	partially	yes	yes	yes	no	no	yes
P7 - Performance	no impl.*	≈ 11%	0%	0%	≈ 1%	≈ 13%	≈ 6.6%	≈ 1.2%
P8 - Memory Consumption								
a) Memory Footprint	no impl.*	low**	0%	low**	0%	high	≈ 37%	≈ 5%
b) Disk Space	no impl.*	low**	0	low**	0	≈ 104MB	73%	0/7%
P9 - Library Support								
a) Static Libraries	yes	yes	yes	yes	yes	yes	yes	yes
b) Shared Libraries	yes	partially	yes	partially	yes	no	no	yes
P10 - Target Architectures	no impl.*	x86 ELF	x86,ARM, ...	x86 ELF	x86 PE	x86 ELF	x86 ELF/PE	x86/ARM ELF

Table 1: Comparison of Existing Randomization Methods to *XIFER*

* In [6], only an experimental setup has been used. The author mentions that the approach induces negligible performance and space overhead. In contrast, [11, 18] provide no implementation and evaluation.
** No precise numbers provided in [4, 21].

(all library code and executable code), the code is cut into arbitrarily small pieces (step b). In the last step c, all code pieces are spread across the whole address space.

In order to make the individual steps work, we have to overcome several challenges. The most obvious one is the fact, that code cannot simply be re-arranged without breaking its semantics since all control flow information (e.g., branch addresses) is outdated. This challenge is addressed by building upon binary rewriting techniques, that is disassembling code, understanding its semantics and subjecting it to the desired changes and re-assembling it. The binary rewriter's duty is to make sure that all changes are reflected in the output code but also its original semantics are still preserved. With a rewriter in place, we can intelligently choose the points where we cut the code into pieces and the rewriting process preserves the original control flow despite its shuffled layout. A dynamic translation approach – mimicking the changes in a virtual machine – is out of the question because of its poor performance [23, 5, 16] due to its piecemeal and constant translation process.

The existing binary rewriting approaches do not feature load-time static rewriting and are not customizable to our needs. Further, a full-blown rewriter is over the top for our needs and hence does not deliver the performance for an ad hoc translation at process start-up. Hence, we built a rewriter from scratch as detailed in the implementation (Section 5). For the time being, we take the binary rewriter for granted and first explain our randomization solution.

4.2 Randomization

The randomization's goal is that no instruction remains at its original relative distance to any other instruction. This ensures that leaked pointers do not reveal any information about their surrounding code. The locality of code is further kept minimal, i.e. it is split apart, so that an attacker cannot guess anything about the surrounding of any byte of code. Other randomization solutions that keep the code segment as one block always reveal that for any leaked pointer at position x the remainder of the code must be in the interval $]x-s, x+s[$ while s is the size of the code. Our deliberate low locality on the other hand is achieved by splitting the code at certain positions right between two subsequent instructions. These cuts result in code that is broken in pieces whose order can be shuffled. The binary rewriter automatically takes care of keeping formerly subsequent instructions that have been severed in a sequential control flow (see Figure 3).

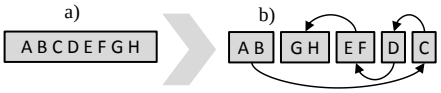

Figure 3: Splitting of code into several interconnected pieces.

4.3 Piece Size

Another challenge is the fact that splitting code in too many pieces imposes a lot of pressure on the processor's instruction cache (as can be seen in the evaluation section 6) since the locality of code has been destroyed. This is why we constrain the amount and position of the cuts:

Positions: When possible, we leverage already existing control transfer instructions (e.g. `jump, call`) as a splitting boundary. This has two advantages: First, there is no need to connect the severed control flow later, when the pieces have been moved away from each other, because there is already a control flow instruction that can be adjusted. Second, when using an already existing control flow instruction, the cache miss penalty is most likely to be identical to the original program.

Amount: The maximum possible entropy of a 32-bit user mode process however is limited to 2^{31} (2 GB). The entropy of 13 permutations is already larger than that ($13! = 6,227,020,800 \approx 2^{32.5}$). Hence, it actually makes no sense to split more than 12 times[1] for 32-bit system or more than 16 times for a 64-bit system (with 48 bit address user space).

[1] 12 times splitting makes 13 pieces, $log_2 13! \approx 32.54$ bits of entropy

4.4 Compliance

As described in Section 3, an ideal randomization tool needs to comply with several criteria. While not all of the criteria can be fulfilled at the same time, in this section, we explain why we think the best trade-off of them is met by *XIFER*.

P1 – Effectiveness. The most important property (P1) concerns the effectiveness of our solution against code reuse attacks, which is the main objective of this paper. We achieve complete mitigation of code-reuse attacks by diversifying the location of each instruction, so that the addresses needed to mount a successful code-reuse attack remain unknown. In addition, *XIFER* is not vulnerable to *disclosure attacks*, i.e. the address of a known function is leaked to the adversary allowing him to revert the memory structure of the executable, a library or even the whole application. This is due to the fact that all offsets between functions, basic blocks of code and even instructions have been randomly changed. Even if the permutation and the memory layout of one specific instance is known, the adversary cannot assume that the target device is using this instance, since our diversification is re-applied for each application run.

P2 – Entropy. Our diversification techniques also yield very high diversification entropy, i.e. every instruction is moved from its original location, and their relative distance to each other is completely random. With the permutation of arbitrarily small code pieces, we achieve an entropy of $n!$, while n denotes the number of pieces the code was divided into. While n can be arbitrarily large, a suitable position to cut is a boundary of a basic block. Our tests have shown that on average 15.5% of all instructions are such boundaries. This means that for a sample 1000-instruction-binary we end up with 156 code pieces to shuffle. This entropy of 156! is already higher than the ultimate ASLR solution that would provide an entropy of 2^{48} on a 64-bit system. Moreover, besides code transformation, we randomize the location of each data section to achieve a fully-randomized memory layout.

P3 – Frequency and P6 – Signing. Since *XIFER* performs its operation entirely at runtime, we are able to automatically randomize a program for each run (*P3*) while keeping compliance to code signing (*P6*).

P4 – Input Information. Instructions may reference other data or other code (control flow). These references need to be detected reliably in order to adjust them accordingly to the new randomized position. In most cases, references are encoded in the instruction (e.g. a branch) and can be detected automatically by disassembling the code. However, the value of a register at run-time could represent an address which is used to direct control flow (indirect jump) or to dereference data (pointer). Calculating the value of that register beforehand is a highly involved task. For instance, `mov $0x8067ab, %eax` might represent an arbitrary number being moved to `%eax` or it might be an address to which `eax` should point. Relocation information solves this problem by pointing to positions in code and data that represent addresses. Fortunately, all address references, including C++ vTables, and indirect jumps benefit from that identification and can be rewritten reliably. There exists literature about how to reliably disassemble and rewrite code when no relocation information is available [33]. However, these solutions suffer from significant space problems, and cannot accurately determine indirect jump targets. Hence, for the time being, our solution was implemented using relocation information.

Note that *P7-P10* are implementation-related and we will show how *XIFER* addresses these criteria in the implementation section (Section 5) and the performance evaluation (Section 6). To meet the code coverage property (*P5*) our solution faces and tackles several technical challenges that are addressed in the following subsection.

4.5 Technical Challenges

Our goal is to achieve all properties mentioned in Section 3. This poses several technical challenges that are mainly related to the code coverage criterion (*P5*). Without loss of generality, examples herein are given in x86 assembler.

CH1 – Function Returns.

The position of a split at the end of one and the beginning of the respective next basic block is not always a trivial case. Simple cases are instructions that unconditionally transfer control to another point in the program, e.g., a jump instruction (`jmp 0x1234`) for x86. These instructions can simply be rewritten to their new address in memory to which the original target has been moved. However, this does not hold for function calls or conditional branches as they feature an *implicit fall-through* control flow, i.e. the control flow will continue at the next instruction after returning from the call. The same is true for conditional branches which *might* continue at the next instruction depending on the outcome of a comparison. In either case, the subsequent instruction to which the control flow would implicitly fall through can potentially be moved away because it is now being part of a different code piece that has been shuffled away.

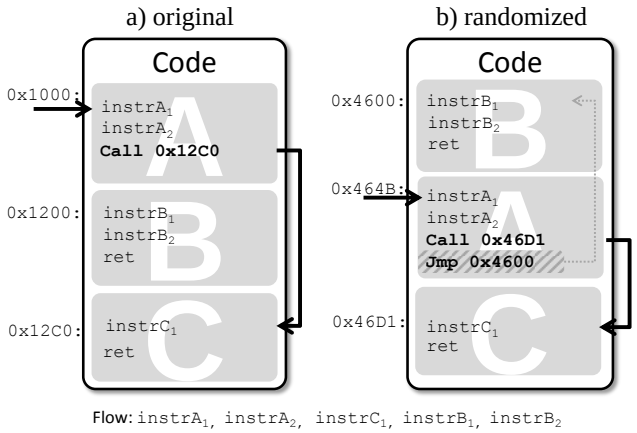

Figure 4: Implicit fall-through control flow requires the injection of jumps in order to retain the original sequence.

Consider the example shown in Figure 4. After the execution returns from the call to `0x12C0` (original), it will continue execution after the `call` instruction by falling through to code piece B (`instrB1, instrB2`). The randomized version would also return to the position right after the `call` where it left off. In contrast to the original, the code has been moved away and the control flow would fall through to code piece C, which is wrong. Consequently, we need to insert a jump in order to connect control flow with the original code piece that now resides at a different position.

CH2 – Position-Independent Code (PIC).

PIC can start execution immediately without the need for certain instructions to be adjusted. It became widely adopted with the introduction of ASLR in modern operating systems. The avoidance of addresses is achieved by using only relative addressing for code branches, function calls and even data. Consequently, the absolute addresses in memory may shift, i.e. the base address of code and data may change, but the relative distances within code and data must stay intact in order for the relative calculations not to become stale. Typically, the code calculates its own address in memory and references code and data relative to its current position. Data is referenced using the Global Offset Table (GOT) by knowing the relative distance to the GOT and indexing the GOT to read and write data.

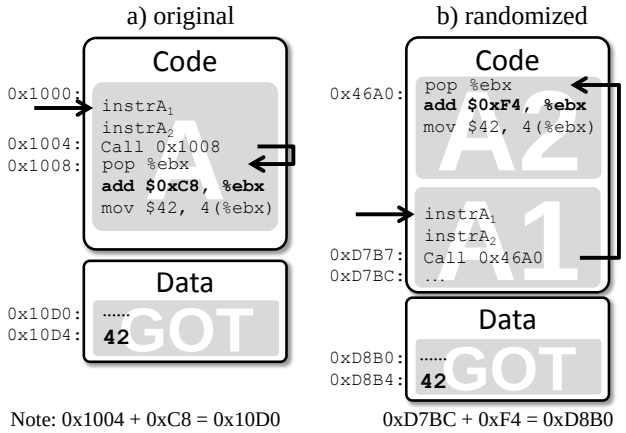

Figure 5: Position-independent code (PIC) needs to be adjusted as it assumes relative relations inside the code.

As we move around code pieces to deliberately change code layout and hence their relative distances to each other, the calculation of the GOT is no longer correct. This necessitates a detection of PIC and its correction in order for the data references to be still valid after the shuffling of code pieces. Consider the example in Figure 5 which depicts a common write operation to data residing in the GOT. In this example, a `call` instruction was issued by the compiler that targets the next instruction. This next instruction (`pop %ebx`) in turn pops the return address off the stack that has been placed there by the `call`. With this trick, register `ebx` now holds the absolute address in memory of where the call should return, in this case the absolute position of the `pop` instruction.

In the randomized version (Figure 5b), the instructions have not only been split apart but also the `pop %ebx` instruction no longer pops its *own absolute address* off the stack. To compensate for this effect, we correct the offset added to the register `ebx` in order to restore the reference to the GOT. To apply this correction in a general fashion, we detect call targets that do not return but rather save a position of the stack that holds the return address. We then follow the register to find the instruction that adds the offset to the found position. When the original offset plus its calculated own position equals the GOT, we have found such a case and correct that offset by rewriting the instruction (e.g. `add $0xF4, %ebx`).

CH3 – C++ Exceptions.

When the *GCC* compiler compiles C++ exceptions, a so-called *unwind frame* is created for every function in a special section called `.eh_frame`. It contains information about how to restore the stack and registers when returning to the point in code that actually catches the exception. This list works similar to a hash map which is indexed with the position in memory that has thrown the exception and provides information about which function catches that particular exception. The information about how to unwind the stack is stored as a stack machine bytecode that is interpreted at run-time while the exception is thrown. This *GCC*-specific machine language incorporates code load addresses and assumes relative offsets, e.g. how to get to the beginning of a function and where the frame pointer is stored. However, since we randomized the code this information is no longer valid. The `.eh_frame` would need to be rewritten according to the changed layout, which we actually did not implement, and leave as future work. Note that with the exception of ILR [16], *GCC* C++ exceptions are also not supported by other randomization approaches.

CH4 – Intermixed Code and Data.

Compilers often optimize code by aligning functions in memory so that they start at the beginning of a cache line. The inevitable gap before aligned functions is sometimes filled with data in the middle of code. A typical fall-through disassembly[2] is thus not possible, as it would interpret alignment zeros (garbage) or intentional data as instructions. To prevent this, code must not be disassembled in a linear fashion but should rather recursively follow the control-flow with respect to indicators for the start of an instruction. There are plenty of such indicators, e.g. targets of control-flow such as function calls, conditional and unconditional branches. We leverage this information to re-align the disassembly process based on discovered control-flow targets and sanity checks that ensure all references from and to code and data stay in their respective segment and target only the beginning of an instruction. We are aware of the fact that from a theoretical point of view, there might be corner cases in which such a disassembly might not be complete. However, we could not find any such cases and a reliable disassembly is not the main topic of this paper but a mere building block. However, it should be noted that an incorrect disassembly could lead to program crashes.

CH5 – Shared Libraries.

Typically, applications link by default to a number of shared libraries (e.g., the Unix standard library libc). Due to space and efficiency reasons, the code pages of these libraries are shared among the various simultaneously running processes. Hence, when every process gets a randomized version of the same shared library, the operating system can no longer benefit from code sharing across multiple processes because their content has changed. For flexibility reasons, we offer the user or system administrator to decide himself whether to favor security (i.e., allowing *XIFER* to generate multiple randomized library versions on-demand for each application) or benefit from shared library memory savings (i.e., randomizing a library once at boot or in a static offline phase). In

[2]The process of disassembling instructions from the beginning of the code in a linear fashion to its end.

particular, for the latter case, we have implemented the possibility to write the randomized libraries back to disk as ELF files. Since shared system libraries are typically not included in digital signatures for signed applications, randomizing them (e.g. every boot or in an offline phase) does not pose a threat to the signatures as long as the main executable remains untouched. Due to our support of shared libraries, we accurately address *P9*.

5. IMPLEMENTATION

The very heart of *XIFER* is the binary rewriter that is responsible for keeping the program semantics despite all the changes we subject the program to. We implemented the rewriter for both mainstream processor architectures Intel x86 and ARM for Linux and its Android derivative. Our prototype can be applied to all binaries that comply with the Linux ELF format which is the default file format for all Linux distributions and includes executable files as well as shared libraries. The code is entirely written in `C++` and contains 7183 source code lines (SLOC).

The binary rewriter builds the foundation for disassembling instructions, injecting instructions, relocating code and adjusting references. This is enabled by keeping an additional layer of references above the original x86/ARM instructions that keep track of references to/from code and to/from data. Transformation to achieve the desired randomization are applied directly on selected instructions. There code can teared apart and new addresses can be assigned to the individual newly generated code pieces. Lastly, the code pieces are transformed back to x86 or ARM instructions either on-the-fly in the address space of a process or can be written as an ELF file on disk, if need be. Most of the code is architecture-agnostic, only special cases of ARM and x86 are handled in small fractions of the total code. It is, however, not possible to transform a program from x86 to ARM due to the unique design that treats most of the code as a black-box because it does not operate on an intermediate language. Any other processor architecture can be easily supported when a disassembler for the architecture is available. Thus, *XIFER* accurately addresses property *P10* (see Section 3).

5.1 Internal Workflow

We explain the implementation of our on-the-fly rewriter with the running example of ARM code shown in Figure 6. The example also applies to Intel x86 instructions. This example assumes a very simplified program that consists of a branch (`beq`) whose target is the `mov` instruction located at address `837c`. This example explains why not all of the code needs to be disassembled while still guaranteeing an instruction-granularity randomization. In the following, the main steps involved in the rewriting process are explained:

1. *Loading* the executable.
2. *Disassembling* the bytecode on-the-fly.
3. Building a *reference graph* of the executable.
4. Applying *code transformation*.
5. Writing the executable back to memory (*fixation*) so that it can start executing

Step 1 — Executable Loading.

In general, a program is composed of different loadable (code and data) segments. These segments originate from different files like the executable file itself and shared libraries the program depends on. Therefore, we intercept the loading of the ELF executable file after all its dependent libraries have been loaded in the address space but before execution begins. To achieve that, we use the `LD_PRELOAD` environment variable that defines shared libraries that are forced to be present in an address space by the operating system. Originally intended to fix bugs of legacy software by allowing the override the symbol resolution of shared libraries, it is a perfect anchor point for our randomization solution. *XIFER* is compiled as a library (`librewrite.so`) to be automatically injected using `LD_PRELOAD` and features a special section called `.init` that is guaranteed to get executed before any other code. At that moment in time, the program that is to be randomized and all libraries the program depends on (including our injected `librewrite.so`) are now marked as loaded by the Linux kernel. This does not mean that they are actually loaded. Luckily, Linux uses a mechanism referred to as *lazy binding* for the resolution of symbols in shared libraries and on-demand paging for the loading of code in the main executable as well as all shared libraries. We use the second advantage of the `LD_PRELOAD` mechanism, i.e. overriding symbols, to replace the on-demand resolution of symbols with calls to our `librewrite.so`. This way, the rewriter and the standard Linux linker do not get in each other's way and do not do double the work.

By iterating over the mapped shared libraries and the executable file, we load file handlers to the particular ELF files representing the code and data of all loaded libraries. If available, we read the relocation information from the ELF file as well. However, relocation information provide *no* information about *local* data/code references within a segment, but *XIFER* still requires this information in order to accurately perform code transformations within a code segment, which makes a disassembly component a necessity.

Step 2 — Disassembly.

The main goal of the dis- and reassembly process is to make it partial as to avoid putting labor in instructions that are not modified in the rewriting process anyway. This is true for the majority of instructions as they do not reference memory addresses and hence their byte representation is not affected by the randomization.

ARM and x86 instructions are composed of a mandatory opcode, potentially followed by other information, most importantly an immediate value such as a memory address. This eases the processes of disassembly as we can simply use a look-up table for the opcodes to know whether a particular instruction uses an immediate value that encodes an address. If it is not, it is simply treated as a black-box – a blob of bytes. This feature does not only increase the performance[3] but due to inspecting only instructions of interest, it also comes in handy because *'uninteresting'* black-box instructions are treated no different from data. This is an advantage compared to classical disassemblers such as *IDA Pro* or *objdump* which need to decide whether some position is either code or data.

[3]Our experiments showed that our disassembler is faster by factor 10 compared to binutil's disassembler that objdump internally uses.

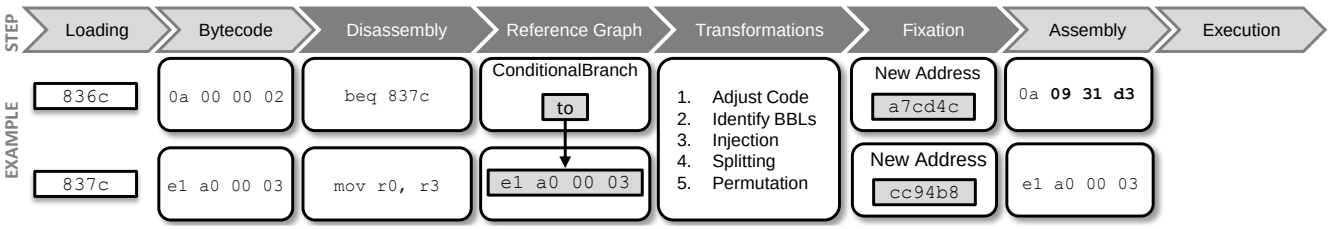

Figure 6: Processing Steps of the Rewriter

In contrast to ARM, which features a fixed-length instruction set, that of x86 is variable. That means, going through the code to find instructions is not a matter of indexing them by a multiple of 4 – which would be the instruction length for ARM. Hence, for x86 it is impossible to predict where an instruction begins without knowing where the preceding one ended. Again, we use the same look-up process to identify an opcode and look-up its length. The opcode also includes information about whether optional immediate values make the instruction longer. Once the length has been determined, the way is paved to start over at the next instruction and so forth.

All instructions that reference data or code are candidates that potentially need to be rewritten later and act as input to build the reference graph that provides information as to which other part of code or data an instruction refers to. The reference graph is similar to relocation information in the sense that it is architecture-agnostic and only saves which part of an instruction encodes an absolute or relative address. As an optimizing step, all instructions still have the original bytecode attached, so that they can be written back to memory and are either left untouched in case of a black-box instruction or are rewritten with aid of the reference graph that tells which part of the instruction encodes an immediate value that needs to be adjusted. Further, x86 and ARM are both generic enough, so that changing address-dependent information in a bytecode is limited to changing two's complement bits in a masked representation of the bytecode. This enables an efficient rewriting process as all of the original bytes are copied to their new memory position and are adjusted with the aid of the reference graph.

Step 3 — Building The Reference Graph.

To resolve code and data references, we build a reference graph by decoding only those instructions that are known to potentially refer to addresses. As an example decoding, we explain this process using the `beq` instruction in Figure 6. This instruction's bytecode is `0a 00 00 02`, while `0a` represents the opcode (`beq`) and `00 00 02` encodes the two's complement representation of a relative addressing. This two's complement representation of the decimal number 2 must be multiplied by 4 (as ARM instructions can only target addresses that are aligned by a multiple of 4). The `beq` instruction is stored at address `836c` which leads to an absolute target of $\texttt{836c} + 2 \cdot 4 = \texttt{8374}$. Moreover, due to the pipelined architecture of ARM, the program counter `PC` always points two pipeline stages (or 8 bytes) ahead, which leads to `837c` as absolute target for the `beq` instruction. This decoding is stored as additional *FastDecode* (see Table 2) information and is attached to the instruction, so that it can be used to generate appropriate bytecodes later. Particularly, *FastDecode* information includes whether it is signed or unsigned, a bit mask, a bit shift and a summand. This coding is generic enough to enable the rewriter to later write back addresses to an instruction without understanding the instruction itself, and is faster than assembling an instruction. In this example, the attached information would store the values depicted in Table 2 for the `beq` instruction.

Info	Value
Opcode	`0a`
Signedness	Signed
Bit Mask	`0x00ffffff`
Bit Shift	2 (left)
Summand	8

Table 2: *FastDecode* information codes how to write back a part of an instruction in an assembler-agnostic way

Using this information, we build the reference graph. The reference graph (see Figure 7) is built by introducing a layer of indirection. Each instruction is routed through the layer of indirection to the instruction it refers to. This is the most important step as it keeps references to the original instruction, even through they might be moved in memory.

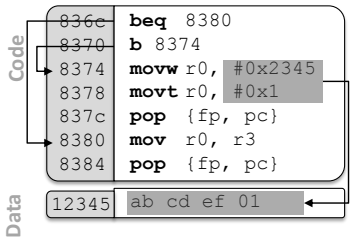

Figure 7: Building the Reference Graph from Instructions

Step 4 — Code Transformation.

The rewriter supports grouping instructions to *Instruction Sequences* that are later used to move code pieces together in memory. Inserting instructions in the *Instruction Sequence* is supported and used as a building block for the necessary explicit connection of severed code pieces (see challenge CH1) or for the insertion of `nop` instructions that do not change the program behavior.

The actual idea of randomization, or code piece permutation respectively, is implemented as operations that can be applied to *Instruction Sequences* or individual instructions. Prior to these operations, the challenge *CH1* of implicit control-flow needs to be tackled. We do this by introducing explicit branches (e.g., `jump 0x1234`) at the end of a code piece in case the original control-flow exhibited an implicit

fall-through as described in Section 4. Using this trick, we can then move the code to different memory positions and the reference graph (see Step 3) will later ensure that this injected jump points corresponding code piece that is ought to be connected to it. The number of those artificial code pieces can be specified by the security parameter.

Step 5 — Fixation & Assembly.

This step assigns a randomly chosen address to each code piece. To keep the number of wasted memory pages low, code pieces are grouped to sections that resemble the size of a memory page (e.g. 4 kB). Should an instruction or code piece not fit in (a multiple of) a memory page, it must go into the next one. The induced overhead of this procedure is rather low, as the aligned instructions on ARM allow exactly 1024 instructions to take place in a memory page without the need for padding. For x86, there is no standard instruction length or alignment, hence how many bytes need to be padded at the end of a memory page varies depending on the instruction. Our empirical studies have shown that the padding in the range of 2 to 4 bytes, albeit the theoretical maximum of one complete instruction that does not fit is 14 bytes[4].

We then assign randomly chosen addresses to the created sections (being a multiple of a page size in length). The number of sections, again, depends on the security parameter, as it is not necessary to introduce a higher entropy than the theoretical limit of either 32 bits of 64 bit address space.

Now every section has an address assigned. We then write back the instructions to their respective new addresses in memory while adjusting all code and data references with aid of the reference graph. This can be efficiently done, as the original bytecode of an instruction is still stored behind the layer of indirection by the reference graph. If we encounter an instruction that references code or data, a new bytecode with the adjusted address has to be emitted to the corresponding new location of that instruction with aid of the attached *FastDecode* information. If the instruction does not need any memory address corrections, the old bytecode is simply copied without the need to assemble a new instruction (see Figure 8).

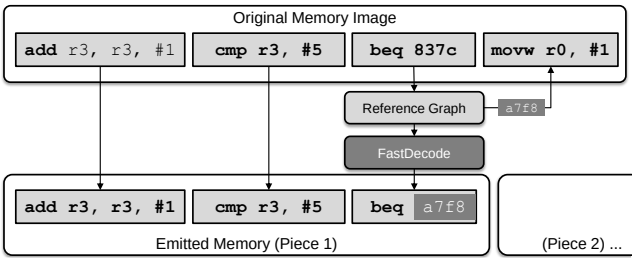

Figure 8: Emitting the Final Opcodes to Memory

Finishing.

Once the rewriting process is done and execution could start, we unload the rewriter by releasing the mapped memory that `librewrite.so` occupies. This is easily doable as the entire code of the rewriter resides in the `.init` section which is never executed again after it has been finished. This procedure is similar to how the Linux kernel frees memory after the initialization of modules has finished.

[4]The longest x86 instruction consumes 15 bytes

Debugging.

Debugging a software-diversified process is rather difficult in comparison to the normal process for two reasons: First, the debugger expects the instructions in memory to be in the same order as the program that is stored on disk. However, *XIFER* completely permuted the layout and addresses of the process and thereby renders the debug symbols obsolete. Since our randomization also shuffles the memory pages at process load time, this has to be propagated back to the debugger. In order to solve this problem, debug symbols according to the permutation of our tool are emitted when requested by the user. We chose to rewrite the debug information of an ELF file to a different file according to the randomization. Currently we support the common DWARF [1] file format which can be read e.g. by the 'gdb' debugger. The gdb debugger can then step through the code, inspect variables etc. as if the program were unmodified.

The second debugging issue is that two subsequent executions of a program result in completely different memory layouts, which makes it harder for humans to understand memory-related faults in the program in question. Further, symbolic debuggers usually allow to visually track data structures based on their addresses. If the addresses would change between every program run, finding the new addresses is labor-intensive. We avoided this issue by adding a debug flag to `librewrite.so` that indicates that the same random seed should be used for every execution of the program, resulting in the same memory layout for each executing with the debug flag enabled. The consequence is that every run of a particular program ends up in exactly the same address space layout of a process with every single instruction being at the exact same address across multiple runs. For all intends and purposes, this is against common sense of randomizing a process in the first place but greatly helps debugging a randomized process because all the variables reside at the very same address across different process runs or even reboots.

6. EVALUATION

In this section we evaluate the effectiveness of our randomization solution empirically as well as theoretically. In order to demonstrate the efficiency, we used industry standard performance benchmarks (SPEC CPU2006) as well as micro benchmarks for the important control flow instructions.

6.1 Practical Security Evaluation

In order to test the effectiveness, two experiments were performed: (1) Calculating the *gadget elimination*. A comparison of found gadgets before and after the randomization. (2) **Mitigation of an exploit** targeted to a vulnerable program.

Gadget Elimination. We used the 12 benchmark programs from the SPEC CPU2006 suite (see Table 3) to find ROP gadgets using the program *ROPgadget* [2]. After randomizing the code and writing it to an ELF file, ROPgadget was run again to check whether and how many gadgets have stayed at the original position.

Exploit. To demonstrate the effectiveness against an exploit, we constructed a sample program that is vulnerable to code-reuse attacks. The code is shown in Appendix A. The function `foo()` opens a file of which the path and length

Benchmark	ROP gadgets	Remaining gadgets
400.perlbench	67	0
401.bzip2	51	0
403.gcc	194	0
429.mcf	45	0
445.gobmk	105	0
456.hmmer	58	0
458.sjeng	57	0
462.libquantum	45	0
464.h264ref	79	0
471.omnetpp	168	0
473.astar	91	0
483.xalancbmk	460	0

Table 3: Overview of the SPEC CPU2006 integer benchmark suite. The C++ benchmarks are listed in gray.

are provided as parameters. Then, `fgets()` reads as many characters as specified by the `file_length` parameter and copies them into the local buffer `buf` without checking its bounds. This in turn allows an adversary to divert the control-flow by overflowing the buffer and eventually overwriting the return address of `foo()`, and inject a ROP payload on the stack. We used ROPgadget to find gadgets in the executable and then successfully mounted a shellcode exploit.

After the randomized program has been started the exploit failed. Even intentionally disclosing addresses with `printf ("%x", &foo)` no longer works as the relative offsets in the code segment have been changed.

6.2 Performance Evaluation

We evaluated the performance of *XIFER* on the Intel x86 platform and conducted micro benchmarks on the ARM platform. To evaluate the efficiency, we used the SPEC CPU2006 integer benchmark suite for the x86 version.

Runtime Overhead on Intel x86.

All benchmarks were performed on an Intel Core i7-2600 CPU running at 3.4 GHz with 8 GB of DDR3-SDRAM. We excluded two of the total twelve benchmarks because they use C++ exceptions (see section 4.5 CH3). We compiled all benchmarks using gcc-4.5.3 and the uClibc C library. All measurements include a *complete* randomization of the entire address space including the executable and all shared libraries. We examined three different randomization configurations: (Config-1) Maximum entropy of 52 bits, (Config-2) Forcing a split of code after exactly 15 instructions (Config-3) Strict BBL permutation: all found BBls are split into code pieces.

The results of our evaluation are summarized in Figure 9 and demonstrate that *XIFER* is highly efficient and hence addresses property *P7* (see Section 3). For *Config-1*, which already achieves an entropy of 52 bits, the overhead is only 5%.

Runtime Overhead on ARM.

In contrast to our evaluation on x86, we conducted micro benchmarks for ARM. In particular, we used an Android Nexus S device running Android version 4.0.3. To perform precise measurements, we leveraged the ARM hardware clock cycle counter (CCNT) which is part of the system co-processor (CP15). To measure the runtime overhead of our prototype, we developed an application that calculates 10,000 times the SHA-1 hash of a 1K buffer with padding. The second micro benchmark is a standard bubble sort algorithm being run on an array of 1024 reverse-ordered elements, so all elements in the array need to be touched (worst-case scenario). Again, measured 10,000 times and averaged. In average, the runtime overhead for the diversified executable is only 1.52% for the SHA-1 benchmark, and 1.92% for the bubble sort algorithm.

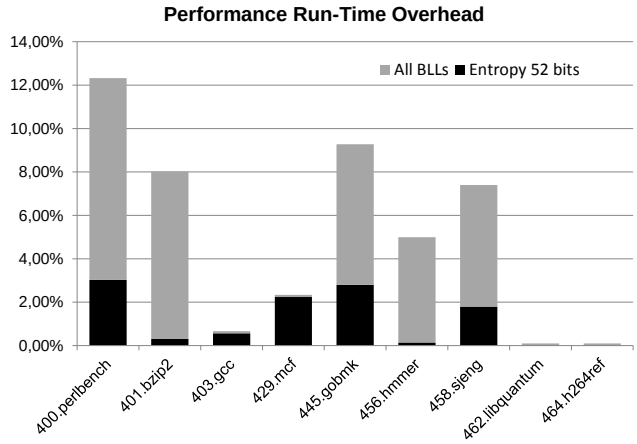

Figure 9: Runtime Measurements with SPEC CPUint2006

Cache Miss Penalty. We also evaluated the cache effects of *XIFER*. Since wild jumping in the code due to the randomization thwarts the locality of code that a processor cache assumes, it has negative impact. For this impact to be measured, we handcrafted code that consists of `add`-instructions whose input depends on the prior output. These instructions are aligned in memory so that they start at the beginning of a cache line and re-occur in memory so that every cache set and every cache line is filled after execution. The total number of instructions exactly fit the entire L1 cache of the Intel Core i7 CPU.

We then split the instruction sequences by inserting jumps between them while keeping the original number of inter-dependent `add` instructions. This effectively decreases the number of instructions that are executed per cache line before jumping to the next location. As the `jmp` instructions are inserted in equal distance to split the sequences, the total number of instructions to execute grows larger than the L1 cache, which leads to cache misses and lines being evicted from the cache in order to load new lines. The total runtime of all instructions in the cache was measured 100,000 times.

For our benchmark system, equipped with an Intel Core i7-2600 (32 KB L1 cache, 64 bytes per line), we found an acceptable minimum length of 6 `add` instructions (12 bytes) before a jump. This yields a negligible overhead of 0.4%. On the other hand, smaller sizes induce a significant number of cache misses. For instance, when we lower the maximum number of instructions between a jump to 2 instructions, we notice 90.3% overhead due to cache misses as every 3rd instruction is a jump instruction. Hence, we suggest not to set the granularity of randomization to ≤ 6 since this still achieves a very high entropy.

6.3 Rewriting Time

Based on the SPEC CPUint2006 benchmarks we also evaluated the time *XIFER* requires to rewrite and randomize a program. In average, the throughput of the rewriting is

5500 kBit/s which demonstrates the efficiency of our approach.

6.4 Memory Overhead

The possibility to write out ELF executable or shared library files might increase their file size compared to the original, because the code is more bloated and additional segment have been introduced to cope with the different load addresses.

File Size. Encapsulating each memory page in a separate segment in the ELF file requires the allocation of one section header and one program header per page. A section header is 40 bytes and the ELF program header is 32 bytes which leads to an overhead of 72 bytes per 4096 byte memory page, or $\approx 1.76\%$. Figure 10 depicts both, the increase of instructions due the static translation as well as the increase of the ELF section and program headers. `librewrite.so` itself occupies 72 kB when loaded.

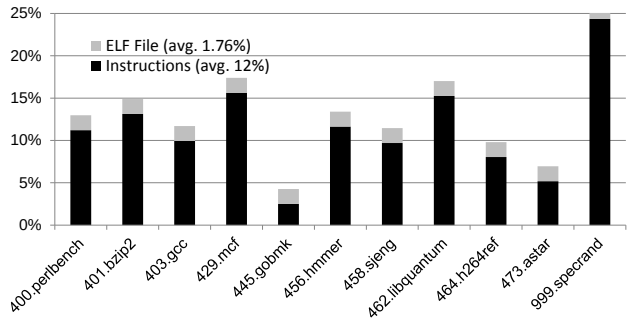

Figure 10: Memory overhead after static translation.

Run-Time. During run-time, the `librewrite.so` has to loaded once into the address space of a process. It is, however, unloaded after the `.init` section has finished and the target program has been randomized. The code size of `librewrite.so` that is temporarily mapped into an address space is $\approx 90kB$. The overhead due to the inserted instruction varies. On average, it increases the code by $\approx 5\%$.

In summary, our approach has a negligible impact on filesize and memory. Hence, we accurately address criterion *P8*.

7. OTHER MITIGATION TECHNIQUES

In this paper we focused on randomization-based countermeasures against code reuse attacks. The main advantage of these defenses resides in the fact that they typically require no access to source code, perform efficiently, and are already deployed in its basic form (ASLR) on today's commodity systems. Nevertheless, and for the sake of completeness, we briefly elaborate on the most well-known countermeasures against return-oriented programming like attacks in the following.

One of the first defense techniques against runtime attacks (that are based on corrupting return addresses) is StackGuard [8], a compiler extension which inserts random stack canaries before return addresses on the stack. A more comprehensive defense is provided by StackGhost [10] which encrypts return addresses on the stack and proposes the concept of return address stacks (i.e., shadow stacks) to keep valid copies of return addresses in a protected memory area. However, these defenses only focus on specific code reuse attacks, and can typically be circumvented by a sophisticated adversary. Another compiler-based solution against ROP attacks was proposed by Onarlioglu et al. [27]. The authors propose a compiler extension for Intel x86 to eliminate the so-called unintended instruction sequences of a program.

A very well-known binary based solution against code-reuse attacks is monitoring of the program flow which has been originally proposed by Kiriansky et al. [22]. In particular, control-flow integrity (CFI) [3] ensures that a program only follows legitimate execution paths. However, CFI induces more performance overhead than randomization-based proposals [3].

8. CONCLUSION

Runtime attacks that reuse existing code pieces (e.g., return-oriented programming and return-into-libc) are a prevalent attack vector against today's applications. In this paper, we tackle these attacks and present the design and implementation of an efficient mitigation technique that is inspired by the principle of *software diversity*. Our software diversity tool *XIFER* accurately mitigates code-reuse attacks by diversifying the structure of an application for each run by means of binary rewriting at the load-time of the application. At the heart of *XIFER* is our binary rewriter which disassembles application binaries on-the-fly, performs code transformations and assembles new application instances with new memory layouts, while still covering the entire semantics of the initial program. *XIFER* is fully dynamic, highly effective (provides a high randomization entropy based on a security parameter), and efficient (induces only 1.2% of runtime overhead in average). Moreover, it requires no access to source codes (which are rarely available in practice), and is compatible to application signatures. Our reference implementation targets both ARM and Intel x86 processors. In order to achieve a highly efficient and effective dynamic binary rewriter, we had to overcome a number of challenges which we highlighted in this paper.

9. REFERENCES

[1] Dwarf 2.0 debugging format standard. `http://www.dwarfstd.org/doc/dwarf-2.0.0.pdf`.

[2] ROPgadget. `http://shell-storm.org/project/ROPgadget/`.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[4] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*. USENIX Association, 2005.

[5] D. Bruenning. *Efficient, Transparent and Comprehensive Run-time Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[6] F. B. Cohen. Operating system protection through program evolution. *Computer & Security*, 12(6):565–584, Oct. 1993.

[7] comex. `http://www.jailbreakme.com//#`.

[8] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and

Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, 1998.

[9] R. Enderle. Windows 8: The app store to rule them all? http://www.conceivablytech.com/9973/products/windows-8-the-app-store-to-rule-them-all.

[10] M. Frantzen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security Symposium*, 2001.

[11] M. Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, pages 7–16. ACM, 2010.

[12] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically Returning to Randomized lib(c). In *ACSAC*, 2009.

[13] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium*, 2012.

[14] D. Goodin. Apple QuickTime backdoor creates code-execution peril. http://www.theregister.co.uk/2010/08/30/apple_quicktime_critical_vuln/, 2010.

[15] Google Play. https://play.google.com/store.

[16] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd My Gadgets Go? In *IEEE Symposium on Security and Privacy*, 2012.

[17] V. Iozzo and R. Weinmann. PWN2OWN contest. http://blog.zynamics.com/2010/03/24/ralf-philipp-weinmann-vincenzo-iozzo-own-the-iphone-at-pwn2own/, 2010.

[18] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. In *Moving Target Defense*. 2011.

[19] X. Jiang. GingerMaster: First android malware utilizing a root exploit on Android 2.3 (Gingerbread). http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/, 2011.

[20] M. Keith. Android 2.0-2.1 Reverse Shell Exploit, 2010. http://www.exploit-db.com/exploits/15423/.

[21] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *ACSAC*, 2006.

[22] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security Symposium*, 2002.

[23] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[24] Mac App Store. http://itunes.apple.com/us/app/apple-store/id375380948?mt=8.

[25] C. Miller and D. Blazakis. Pwn2Own contest. http://www.ditii.com/2011/03/10/pwn2own-iphone-4-running-ios-4-2-1-successfully-hacked/, 2011.

[26] National Institute of Standards and Technology. National vulnerability database statistics. http://web.nvd.nist.gov/view/vuln/search.

[27] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *ACSAC'10, Annual Computer Security Applications Conference*, Dec. 2010.

[28] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy*, 2012.

[29] PaX Team. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[30] M. J. Schwartz. Adobe Acrobat, Reader under attack from zero-day exploit. http://www.informationweek.com/news/security/vulnerabilities/227400016, 2010.

[31] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[32] H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. Boneh. On the Effectiveness of Address-space Randomization. In *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[33] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua. Binary rewriting without relocation information. Technical report, University of Maryland, 2010.

[34] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*. Springer-Verlag, 2011.

[35] P. Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf, 2010.

[36] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

APPENDIX

A. VULNERABLE PROGRAM

```
1 FILE *sFile;
2 void foo(char *path, file_length) {
3     char buf[8];
4     sFile = fopen(path, ''r'');
5     fgets(buf, file_length, sFile);
6     fclose(sFile);
7 }
```