How to Make ASLR Win the Clone Wars: Runtime Re-Randomization

Kangjie Lu[†], Stefan Nürnberger[‡§], Michael Backes[‡¶], and Wenke Lee[†]
[†]Georgia Institute of Technology, [‡]CISPA, Saarland University, [§]DFKI, [¶]MPI-SWS
kjlu@gatech.edu, {nuernberger, backes}@cs.uni-saarland.de, wenke@cc.gatech.edu

***Abstract*—Existing techniques for memory randomization such as the widely explored Address Space Layout Randomization (ASLR) perform a single, per-process randomization that is applied before or at the process' load-time. The efficacy of such upfront randomizations crucially relies on the assumption that an attacker has only one chance to guess the randomized address, and that this attack succeeds only with a very low probability. Recent research results have shown that this assumption is not valid in many scenarios, e.g., daemon servers fork child processes that inherent the state – and if applicable: the randomization – of their parents, and thereby create clones with the same memory layout. This enables the so-called *clone-probing* attacks where an adversary repeatedly probes different clones in order to increase its knowledge about their shared memory layout.**

In this paper, we propose RUNTIMEASLR – the first approach that prevents clone-probing attacks without altering the intended semantics of child process forking. The paper makes the following three contributions. First, we propose a *semantics-preserving* and *runtime-based approach* for preventing clone-probing attacks by re-randomizing the address space of every child after `fork()` at runtime while keeping the parent's state. We achieve this by devising a novel, automated pointer tracking policy generation process that has to be run just once, followed by a pointer tracking mechanism that is only applied to the parent process. Second, we propose a systematic and holistic *pointer tracking mechanism* that correctly identifies pointers inside memory space. This mechanism constitutes the central technical building block of our approach. Third, we provide an open-source *implementation* of our approach based on Intel's Pin on an x86-64 Linux platform, which supports COTS server binaries directly. We have also evaluated our system on Nginx web server. The results show that RUNTIMEASLR identifies all pointers, effectively prevents clone-probing attacks. Although it takes a longer time for RUNTIMEASLR to start the server program (e.g., 35 seconds for Nginx), RUNTIMEASLR imposes no runtime performance overhead to the worker processes that provide actual services.

I. INTRODUCTION

In the arms race of remote code execution, the introduction of non-executable memory has relocated the battlefield: attackers are now forced to identify and suitably reuse existing code snippets, while protective technologies aim at hiding the exact memory location of these code snippets by means of various forms of memory randomization. As a result, a variety of different memory randomization techniques have been proposed that strive to impede, or ideally to prevent, the precise localization or prediction where specific code resides [29], [22], [4], [8], [33], [49]. Address Space Layout Randomization (ASLR) [44], [43] currently stands out as the most widely adopted, efficient such kind of technique.

All existing techniques for memory randomization including ASLR are conceptually designed to perform a single, once-and-for-all randomization before or at the process' load-time. The efficacy of such upfront randomizations hence crucially relies on the assumption that an attacker has only one chance to guess the randomized address of a process to launch attack, and that this attack succeeds only with a very low probability. The underlying justification for this assumption is typically that an unsuccessful probing of the attacker will cause the process to crash, so that it has to be restarted and hence will be randomized again. Technically, the assumption is that each attacker's guess corresponds to an independent Bernoulli trial.

In reality, however, this assumption is not valid in many scenarios: daemon servers (e.g., Apache or Nginx web servers, and OpenSSH) `fork()` child processes at runtime which then inherit the randomization of their parents, and thereby creating clones with the same memory layout. Consequently, if a child crashes after unsuccessful probing, a clone with the same address space layout is created again. Therefore, an unsuccessful probing does not require an attacker to start from scratch, instead he can reuse the knowledge gained in his previous probings. In the following, we refer to such attacks as *clone-probing attacks*.

A fundamental limitation with existing ASLR implementations is that they are only performed at load-time. That is, any process that is created without going through loading will not be randomized. For example, all forked child processes always share exactly the same address space layout. As a result, clone-probing attacks enable an adversary to bypass current memory randomization techniques by brute-force exploration of possibilities. Moreover, practical scenarios often exhibit side-channel information that can be used to drastically reduce the search space. In these cases, the consequences are far worse, as impressively demonstrated by recent sophisticated attacks against memory randomization. For instance, Blind ROP [7] exploits a buffer overflow vulnerability to overwrite return pointers only by one byte each time, and thereby reducing the search space to mere 256 possibilities since the remaining bytes of the return pointer are left unmodified. After a successful return, i.e., if the program does not crash or exhibit unexpected

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23173

behavior, the first byte of a valid pointer has been correctly discovered. The technique is then used repeatedly to discover remaining bytes. On average, the probing of a byte is successful after 128 tries. This approach hence reduces the brute-force complexity from 2^{63} to $128 \cdot 8 = 1,024$ tries on average (for 64-bit systems). A successful clone-probing attack can hence be leveraged to bypass ASLR, which is a general prerequisite for further, more severe attacks, including code reuse attacks, privilege-escalation by resetting `uid`, and sensitive data leaks.

The root cause of clone-probing attacks is that the forked child processes are not re-randomized. Starting a process from its entry point by calling `execve()` in the child after `fork()` [7], however, alters the intended semantics of child forking: `fork` is intentionally designed to inherit the parent's state (variables, stack, allocated objects etc.), whereas `execve()` starts another, fresh instance of a program so that the execution starts at the program's entry point without inheriting any information from its parent. It is *possible* to make the forked child process independent from the parent process so that `execve()` can be used to easily re-randomize the child process; however, the child process will not be able to benefit from the semantic-preserving and resource-sharing (e.g., sharing file descriptors and network connections) `fork()`, and the target server program has to be carefully restructured and rewritten to make the child process not depend on any resource or data of the parent process. For example, the complicated dependencies (e.g., on program structure, file descriptors, shared memory, and global variables) made us give up rewriting Nginx (about 140KLOC) to use `execve()`. As a result, we aim to propose a practical (e.g., negligible performance overhead) and easy-to-use (e.g., supporting COTS binary directly without modifying source code) re-randomization mechanism to prevent clone-probing attacks in a semantic-preserving manner.

A. Contributions

In this paper, we propose RUNTIMEASLR – the first semantics-preserving approach that prevents clone-probing attacks, by consistently re-randomizing the address space of children after `fork()` at runtime while keeping the parent's state. More specifically, the paper makes the following contributions: (1) a *semantics-preserving and runtime-based approach* for preventing clone-probing attacks; (2) a systematic and holistic *pointer tracking mechanism* as our approach's central technical building block; and (3) an open-source *implementation* of our approach and a corresponding *evaluation* on Nginx web server.

The RUNTIMEASLR *approach.* Contemporary implementations of the `fork` system call simply clone the parent's address space in order to create a child that inherits the same state. Cloning is cheap in terms of complexity and benefits from inherited open file descriptors and network connections. Instead, RUNTIMEASLR re-randomizes the address space of children after `fork()` at runtime while inheriting their parent's state, as shown in Figure 1. This consistent re-randomization imposes formidable challenges since the state of a program typically incorporates references to a variety of objects, including code, data, stack and allocated objects on the heap. A successful randomization in particular has to ensure smooth continued execution at the new address; hence, it has to relocate code, data and all references consistently at runtime. RUNTIMEASLR

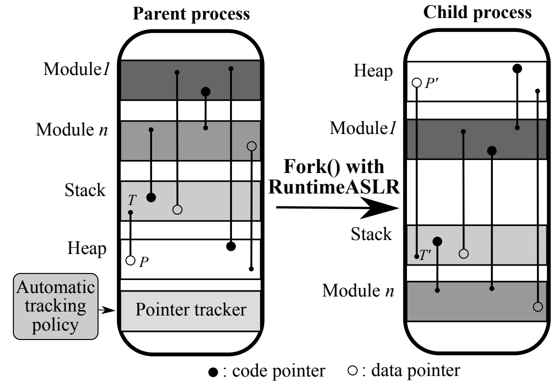


Fig. 1: RUNTIMEASLR approach for re-randomization with fork(). Pointer tracker accurately tracks all pointers based on automatically generated tracking policy set. Module-level re-randomization is performed and all pointers are updated for `fork()`. For example, in parent process, a pointer P in heap points to T in stack. After the re-randomization, stack is moved and thus T is moved to T'. Accordingly, P is patched to P'.

meets this challenges by first conducting a pointer tracking policy generation process that has to be run once, followed by a novel pointer tracking mechanism that is only applied to the parent process. By treating a pointer as a tainted value, the pointer tracking problem is naturally transformed into a taint tracking problem: the initial pointers prepared by OS are used as taint sources, pointer tracking is done using taint propagation, and the necessary propagation policy is automatically created. To generate a pointer tracking policy, instead of manually identifying which CPU instructions typically carry on tainted pointers and which ones remove taints [46], [11], [15], we opted for an automatic approach: the policy generation process executes the target program and sufficient sample programs using Intel's *Pin* instrumentation tool and inspects the semantics of each executed instruction with regard to pointer creation, manipulation or deletion. The result is a policy that describes exactly which instructions deal with pointers in memory and registers.

Pointer tracking mechanism. We use the generated policy to track pointers during run-time. Whenever the parent `forks` a child, RUNTIMEASLR performs address space re-randomization in the cloned child process and patches pointers according to the tracked pointers of the parent process. As a result, the child processes have mutually different address space layouts without facing any instrumentation overhead. Hence, RUNTIMEASLR imposes no overhead to the services provided by the worker processes of daemon servers. Effectiveness requires us to correctly identify all pointers inside *all* dynamically allocated structures, such as stack, heap, `.bss` or even uncharted `mmap`'ed memory. To this end, RUNTIMEASLR first identifies all pointer sources, which we classify into three categories. First, initial pointers prepared by the OS (e.g., the stack pointer `rsp`) because their location is deterministic and fixed; second, program counters (e.g., `rip`); and third, return values of certain syscalls (e.g., `mmap`, `mremap`, and `brk`). In an ASLR-enabled program, no static pointer is allowed, thus any other pointer must be directly or indirectly derived from these pointer sources. In order to address taint propagation that occurs in the kernel, we identify all syscalls that may propagate

pointers and that adapt memory mappings. Our findings show that only a limited number of pointer-propagating syscalls need to be taken into account, and that the memory-related syscalls (e.g., `mmap`, `munmap`, and `mprotect`) can be hooked in order to track changes in memory maps. With taint policies and appropriate syscall modeling in place, pointer tracking is realized by consistently updating hash tables storing taints.

Implementation and evaluation. We have implemented RUNTIMEASLR based on Intel's Pin on an x86-64 Linux platform. RUNTIMEASLR is implemented as a combination of three tools: a taint policy generator, a pointer tracker, and a randomizer. The taint policy generator and the pointer tracker are implemented as two Pintools; the randomizer is implemented as a shared library that is dynamically loaded by the pointer tracker in the child process. Note that, the current implementation of RUNTIMEASLR is dedicated to server programs that pre-fork worker processes at begining. For other programs that do not adopt the pre-fork scheme, different implementation approaches (e.g., compiler-based code instrumentation) may be preferred instead of dynamic code instrumentation. More details will be discussed in section VIII. To evaluate the effectiveness of RUNTIMEASLR, we applied it to the Nginx web server because it is one of the most popular web servers and has been an attractive attack target in the past [7], [21], [35], [9], [20]. Our evaluation shows that RUNTIMEASLR identifies all pointers, effectively prevents clone-probing attacks. Due to the dynamic instrumentation based, heavy-weight pointer tracking, it takes a longer time for RUNTIMEASLR to start the server programs (e.g., 35 seconds for starting Nginx web server), however, the one-time overhead is amortized over the long run-time, and more importantly, RUNTIMEASLR imposes no performance overhead to the provided services after the worker processes are pre-forked.

B. Organization of the paper

In the rest of the paper, we introduce the approach overview of RUNTIMEASLR in Section II, the design of the three key components of RUNTIMEASLR in Section III, IV, and V. We explain the implementation of RUNTIMEASLR in Section VI, and evaluate the correctness, effectiveness and efficiency of RUNTIMEASLR in Section VII. We discuss the limitation and future work of RUNTIMEASLR in Section VIII, compare it with related work in Section IX and conclude in Section X.

II. OVERVIEW OF RUNTIMEASLR

RUNTIMEASLR aims to prevent clone-probing attacks by re-randomizing the address space of a process at runtime. More specifically, RUNTIMEASLR protects daemon servers that fork multiple child (worker) processes for responding to users' requests.

A. Threat Model

The attacker's goal is to launch attacks against a daemon server with ASLR enabled, e.g., code re-use attack like return-oriented programming, privilege-escalation by overwriting the `uid`, or stealing sensitive data stored at a certain address. We assume the daemon server consists of a daemon process and multiple worker processes forked by the daemon process. For the sake of robustness, if a worker process is crashed, a new worker process is created by the daemon process with `fork`. This model is widely adopted by daemon servers, e.g., Apache web server, Nginx web server, and OpenSSH. Bypassing ASLR is a general prerequisite of these attacks, as they usually need to access the code or data (at a particular address) in the process memory. To bypass ASLR, the attacker can mount a clone-probing attack to iteratively recover the address space layout, which is shared amongst all children of the daemon process.

We assume that the operating system running the daemon server realizes the standard protection mechanisms, e.g., $W \oplus X$ and ASLR, and the daemon server is compiled with `-PIE` or `-fPIC`, i.e. the server is compiled to benefit from ASLR. However, the daemon binary may not be diversified, and hence the attacker may have exactly the same copy of it. As a result, attackers can perform both static and dynamic analyses on the daemon program, e.g., scanning it for vulnerabilities. We assume there exists at least one exploitable buffer overflow vulnerability, i.e., a buffer can be overwritten arbitrarily long including return pointers and frame pointers.

RUNTIMEASLR focuses on preventing clone-probing attacks that indirectly infer the address space layout by repeatedly probing worker processes of daemon servers. Other programs that do not adopt the pre-fork scheme usually do not suffer from the clone-probing attacks, and thus are out of scope. Direct leaks, e.g., a leaked pointer in a malformatted `printf` or a memory disclosure vulnerability [45], are out of scope for this paper, but have been covered in existing work [13], [3], [36], [14]. Physical attacks, e.g., cold boot attacks, are out of scope. We assume the OS is trusted, so attacks exploiting kernel vulnerabilities are also out of scope.

B. The RUNTIMEASLR Approach

To defeat clone-probing attacks, one can either reload the child process or re-randomize the address space of the child process by patching all pointers. As reloading the process will not inherit any state or semantics from the parent process, it would constitute a great loss as a programming paradigm. Moreover, reloading requires the existing server programs to be thoroughly restructured and rewritten to make the child process not depend on any resource or data of parent process. Therefore, RUNTIMEASLR instead keeps the semantics of its parent process and re-randomizes the address space at runtime.

Since we want RUNTIMEASLR to be a practical tool, it is designed to use user mode *on-the-fly* translation of programs and hence does not require any source code or OS modifications. Since daemon servers usually respond to a multitude of requests at the same time, performance is also a primary goal of RUNTIMEASLR. Those two goals are brought together by dynamic program instrumentation using Intel's *Pin Tool*: it allows to monitor and modify arbitrary programs on-the-fly and can be detached after randomization in order not to impose any performance overhead.

The high-level overview of RUNTIMEASLR is depicted in Figure 2. We first provide an overview of its three main components. Then, technical details will be given in the subsequent sections.

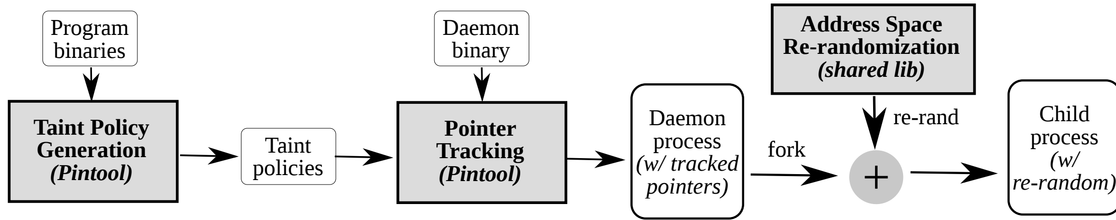


Fig. 2: An overview of RUNTIMEASLR's architecture. It consists of three components: taint policy generation, pointer tracking, and address space re-randomization. Program binaries include the daemon binary.

a) Taint Policy Generation: The first phase of RUNTIMEASLR– taint policy generation – automatically generates pointer tracking policies by inspecting lots of input programs. The generated policy grows with every input program by "learning" more behaviors of Intel x86-64 instructions. At some point, all possible usages of instructions have been seen and the policy does not grow any further. The generated policy file can then be used in the second phase, pointer tracking, to accurately track pointers of arbitrary programs. Although the identification of pointers is not a new topic, existing approaches lack a systematic approach that can vouch that the used tainting policy is sound. Additionally, prior approaches often aimed at catching out-of-bounds access, which is a slightly different problem as it also needs to know the size of the object pointed to by the pointer [27], [40], [39], [30], [46], [11], [15]. These approaches either rely on type-based analysis, or on heuristic-based taint analysis to identify pointers. Type-based analysis can only identify pointers with pointer-type. However, integers that are also used as pointers, such as base addresses, cannot be identified. Heuristics, on the other hand, do not provide any sound guarantee, since they might incorrectly track identified pointers or fail to identify some pointers in the first place. For example, previous approaches [46], [11] that work on instruction-level tracking only specified a handful of operations (e.g., assignment, addition, subtraction, and bitwise-AND) to carry on taints if the input was tainted. However, as we show in Table II, there are not only many more instructions that either carry on a taint or remove a taint, but many of them come as a surprise and can only be discovered by a systematic approach rather than empirical heuristics. For example, the `CPUID` and `RDTSC` instructions modify general purpose registers, thereby potentially overwriting stored pointers.

To tackle this problem, we propose an automated taint policy generation mechanism that *automatically* discovers all kinds of instructions that have direct or indirect effect on stored pointers. The basic concept for creating a pointer tracking policy is to inspect all instructions with respect to whether they create, update, or remove a taint. To this end, we check if an instruction writes to a register or to a memory location and whether that written value constitutes a pointer. Then a pointer is determined based on whether its value points inside any currently mapped memory. This approach does not miss any pointers, i.e., there are no false negatives as long as they point into valid memory. However, this approach might coincidently identify a random integer value as a valid pointer (false positive). For a 64-bit address space such false positive probability is very low since mapped memory segments are sparse in relation to the huge address space. Nevertheless, we further reduce that probability by running the program multiple times with ASLR enabled, and then compare the results. Correctly identified pointers occur in all runs, whereas false positives do not. We use this technique to automatically create a policy that identifies all instructions that will generate or remove a pointer.

b) Pointer Tracking: In the second step, the generated policy is used to perform pointer tracking. Pointer tracking itself consists of three parts: (1) it collects all initial pointers prepared by OS; (2) it performs taint tracking for pointers based on the taint policies generated by the first component; (3) since RUNTIMEASLR is explicitly designed to not modify the OS, some syscalls are modeled for taint propagation. The output of this component is the list of all pointers in the process.

c) Address Space Re-randomization: Using the tracked pointer list, the third component of RUNTIMEASLR– address space re-randomization – then performs module-level memory remapping for all currently mapped memory segments in the child process. As Pin is no longer necessary in the child process, the detachment of Pin is immediately performed after `fork`, and then the re-randomization is triggered as a callback of Pin detachment. After the re-randomization, the dangling pointers are corrected using the list of identified pointers. As a last step, the control is transferred back to the native child process code. Since Pin is already detached, there is no performance overhead in the child processes associated to dynamic instrumentation (see section VII for performance results).

III. AUTOMATED TAINT POLICY GENERATION FOR POINTERS

As alluded to earlier, the existing approaches for taint tracking on binary code are not complete. To tackle this challenging problem, we propose a novel pointer tracking mechanism that accurately identifies pointers (tainting) and tracks pointers throughout their lifecycle (taint propagation). In general, taint analysis can be categorized into *static* taint analysis and *dynamic* taint analysis. In theory, static taint analysis can identify all pointers without false negatives. However, false positives may be unavoidable [19] because of higher level compound type definitions such as `structs` and `unions`, which are not necessarily pointers. As a building block for re-randomization, however, we must neither allow any false positives nor false negatives. Otherwise, an overlooked pointer (false negative) will reference an outdated memory location and will most likely cause a program crash due to an illegal memory access. Similarly, tagging innocent integer values as pointers (false positive) changes data when patching the believed-to-be pointer resulting in erratic program behavior.

Given the limitations of static taint analysis and the strict requirements of our runtime re-randomization, we choose to

leverage dynamic taint analysis to identify pointers. Dynamic taint analysis, however, needs a policy that describes which instructions of an entire instruction set architecture (ISA) modify a pointer, either directly or through side effects, and how the pointer is modified. This is possible by studying the 1,513-pages Intel instruction set architecture manual [25] and hoping that one did not overlook a side effect of an obscure instruction. Alternatively, we opted for an automatic learning process that creates annotations for each observed instruction of the ISA by monitoring what each instruction does during runtime. This is possible by leveraging dynamic instrumentation based on Intel's Pin, which makes the execution of each instruction tangible. By carefully inspecting how and which memory and registers an instruction with different operands modifies, a policy is generated that describes how each instruction either creates a taint, removes a taint, does not modify a taint, or even propagates a taint to another register or into memory. This policy grows with every taught program and it eventually can be used in the second phase, the *Pointer Tracking* phase, to actually track which parts of memory or registers store pointers. The latter part is described in the next section.

d) Workflow: The sample programs used for generating the taint policies also contain the target program. We use Pintool to hook each instruction that writes to at least one register or one memory address (Pin provides an API to iterate all operands, including the implicit ones, e.g., operand `EAX` in `RDTSC`). Then, Pintool provides us with the metadata information for that particular instruction. This includes the opcode, count of operands, types of operands (register, memory, or immediate), widths of operands, read/write flags of operands, and data of operands (e.g., register number or immediate value). For conditional instructions (e.g., `cmovnz`), we further include the flag bits in register `rflags`. As an additional step, we hook syscalls to identify the ones that create pointers, e.g. `mmap()`. Also, memory-related syscalls such as `mmap()` are monitored for the requested memory range they allocate because our algorithm needs to know which memory is currently valid and which is not, for pointer verification.

The actual policy generation executes each instruction and compares the state of all registers and accessed memory before execution to after the execution of each instruction. This way, side effects of an instruction can be detected even though the register was not specified as an operand. For example, `RDTSC` reads the CPU's internal Time Stamp Counter (TSC) into `EAX:EDX`, thereby overwriting what was stored in `RAX` and `RDX`. This can be detected by checking each register for being a pointer using multi-run pointer verification (III-A). If an executed instruction modifies "pointer"-ness of a register or memory location, it is reported as a new policy for that instruction. This policy includes a description whether a pointer was created, removed or copied somewhere else.

e) Example: The instruction `mov RDI, RSP` (in Intel syntax) is always the first instruction that gets executed in every program. We first extract the opcode (that is `384` in Pin), the types, widths, and read/write flags of its operands. The operands are two 64-bit registers with the first one being destination and the second one being source. Then we analyze whether any of those operands (`RDI` or `RSP`) is a pointer using multi-run pointer verification (III-A). In this case, `RSP` is a pointer, while `RDI` is still zero after initialization of the program. After executing the instruction, the operands are checked again for "pointer"-ness. In this case, `RDI` becomes a pointer after execution. Therefore, we generate a pointer tracking policy as follows. Given an instruction with opcode = `384` and exactly two 64 bit registers as operands, the first operand will be tainted after execution if the second operand was before execution.

A. Realizing multi-run pointer verification

To ensure that only actual pointers are tracked, it is crucial that the pointer detection does not mistakenly classify an integer as a pointer, simply because its numerical value coincidentally represents a valid, mapped memory address. To address this problem, we propose *multi-run pointer verification* to check if a value indeed constitutes a real pointer.

The idea of multi-run pointer verification is inspired by the fact that in a 64-bit address space, the mapped memory is sparsely allocated, and it is unlikely for a random data to point into mapped memory. So by checking if a value points into mapped memory, we can determine a pointer with a high probability. To further decrease the false positive rate, we execute the program multiple times at different load addresses (with ASLR-enabled). The workflow of multi-run pointer verification is shown in Figure 3. In the first run, we output discovered policies with metadata, including the relative location (relative address into the base address of corresponding loaded module) of the targets of generated pointers. For each of the next runs, only the intersection of policies is kept, which identifies those pointers whose targets have the same relative locations.

The probability that a non-pointer value passes this check and is hence classified as a valid pointer is extremely low. On 64-bit x86 machines, Linux reserves the lower 47 bits for user mode programs[1]. Pointing inside the user mode address space is thus possible if the upper 17 bits of a random 64-bit number are set to zero. Additionally, the lower 47 bits must point into valid mapped memory that has been chosen randomly by ASLR. For the combination of all loaded modules with a total size of b bytes, the probability for a 64-bit integer pointing inside any loaded module is $b \cdot 2^{-64}$. For each run of the multi-run pointer verification, the probability decreases drastically. In the next run, the probability is not only $b \cdot 2^{-64}$ to be inside any loaded module, but the same instruction must produce an integer whose value has the same offset into the same module to be still considered a (false positive) pointer. Hence, only one valid address for the given randomization remains, which has a probability of only 2^{-64} for randomly chosen integers. For `n` runs, the final false positive rate is deceased to $b \cdot 2^{-64 \cdot n}$. The running Nginx web server for example has a total of approximately 22 MB of mapped valid memory. This would result in a false positive rate of $\approx 2^{-103}$ in a `2`-runs verification.

1) Multi-thread Support: In order to identify the same pointer across different runs, it is necessary to deterministically enumerate all pointer creations in a reliable way – one way would be the order of pointer creation, another would be the memory position of the pointer itself. In a single-threaded program, the position of pointers and the order of their creation are deterministic. However, multi-threaded programs might

[1] `arch/x86/include/asm/processor.h:881` of Kernel 4.1 defines `#define TASK_SIZE_MAX ((1UL << 47) - PAGE_SIZE)`

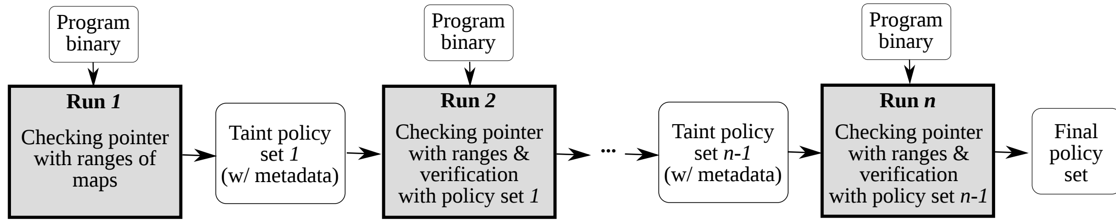


Fig. 3: The work-flow of multi-run pointer verification. For each policy, the metadata indicates which registers/memory is written to. Only pointers that point to the same relative address throughout multiple runs are kept.

create pointers in a different order, due to that threads share a common heap, which also influences the pointer positions inside the heap. To overcome this problem, we also consider where pointers are stored. For pointers that are stored on the heap, the base address of the heap object the pointer resides in and its relative position into the base of heap object are recorded and used to uniquely refer to that particular pointer. Since loadable sections (e.g. `.data` or `.bss`) have their objects at fixed locations and each thread has its own stack, they are not affected by multi-threading.

IV. Taint Tracking for Pointers

Using the taint policy that has been created *once* in the taint policy generation step, the program can be run again with pointer tracking enabled based on the created policy. The first step is to *identify* a pointer when it is created. It is crucial to identify exactly *when* a pointer is created to be able to track it from its inception. Otherwise, it might have been copied to other places already without noticing.

A. Pointer Sources

In ASLR-enabled programs, no static pointer (known at compile time) can exist because all code and data are loaded at an unpredictable address. All pointers are either derived from the current address of execution (`RIP` register), the stack (`RSP` regsiter), the heap (call to `mmap()`, `mremap()` or `brk()`) or they are injected by the OS. For example, `lea RAX,[RIP+0x2007bc]` derives a pointer based on current instruction pointer (`RIP`), and saves it in the register `RAX`. In fact, `RIP` and `RSP` are also injected by the Linux kernel before the program starts. In chronological order, whenever a new process is loaded after calling `execve()`, the Linux kernel inserts pointers into the process. As per the Linux 4.1 source, `execve()` first clears all registers[2] and then sets the first instruction to execute by modifying the `RIP` register and setting the stack register `RSP` to the end of the stack. Then, initial environment variables and program arguments are pushed to the stack before the first instruction of the newly created process begins execution. The initial data in stack prepared by OS also contains some pointers (e.g., entry point). As OS routinely stores these initial pointers in stack in the same order at load-time, their relative locations are fixed, we also apply the idea of multi-run pointer verification to accurately identify them. Once all of these pointer sources are found and tainted, the discovered taint policy of the earlier step can then accurately tracks other pointers derived from these pointer sources.

[2]Macro `ELF_PLAT_INIT(regs, load_addr)` of file arch/x86/um/asm/elf.h

B. Syscall Modeling

The only other way to introduce new pointers is as a result of a syscall. Since we do not want to track pointers in kernel mode, we rather use well-established method *syscall modeling* to mimic the behavior of the kernel with respect to pointer tainting in user mode. Our analysis of all side effects of syscalls has revealed that the only syscalls that modify mapped memory and hence create new pointers are `mmap`, `mremap` and `brk`. Those syscalls are monitored during execution by hooking them inside the Pintool. The return values of those syscalls are then tainted as pointers accordingly. As a side effect, the gained knowledge about memory mappings is used as a plausibility check that data tainted as pointers indeed points into valid, mapped memory.

C. Bookkeeping

The internal bookkeeping of which register or memory location is tainted is stored in a simple but fast hash map. Indexing the memory hash map with an address will return taint information about that address if applicable. Untainted memory will of course not be stored in the hash map and returns `null`. According to the used taint policy, each executed instruction may propagate a taint into another register or memory location (e.g., `mov` or `add`) or remove the taint (e.g., `xor RAX, RAX`). The taint policy contains tuples describing matches based on their opcode, operand type (i.e., register, memory, or immediate), operand size, flag bits (for conditional instructions), and whether operands are already tainted. Using this tuple, the policy is queried on the expected action for the modified registers or memory. If a matching policy is found, it is applied, i.e., the expected registers or memory locations are tainted or un-tainted. Otherwise, no pointer propagation effect is expected and no taints are changed.

V. Address Space Re-randomization

Even with the knowledge of exact pointer locations, re-randomization of a running process is not straight-forward. In fact, we faced the following challenges:

- C1: The child process also inherits the dynamic instrumentation of the Pin by `fork()`. For performance reasons, Pin in child process is unnecessary, thus should be detached.
- C2: Unmapping and clearing of Pintool is a chicken-and-egg problem since the instrumented `fork()` code would need to unmap `itself` while executing.
- C3: Remapping a set of memory blocks is not an atomic process and hence during the intermediate state of remapping, no valid stack and no valid library functions are accessible as they are already remapped.

To overcome those challenges, we designed our re-randomization mechanism as a separate shared library as shown in Figure 4, which is triggered by the callback of Pin's detaching. This way, it can (1) be attached to child process, (2) perform the re-randomization and patch all pointers,(3) clear the Pin code/data, and finally (4) load updated registers and perform a context switch to the newly `fork`'ed child.

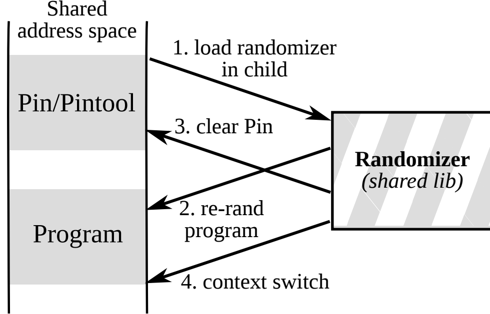


Fig. 4: The workflow of re-randomization.

A. Re-randomization Granularity

The default ASLR provided by Linux OS performs randomization in process-level, which means offsets between modules are fixed. RUNTIMEASLR chooses to perform module-level randomization by specifying random base to `mmap`, which achieves a better effective entropy [23]. However, finer-grained randomizations [22], [29], [4], [48], [24], [18] can be further applied thanks to the tracked pointer information.

VI. IMPLEMENTATION

RUNTIMEASLR is implemented as a combination of three tools: the taint policy generator, pointer tracker, and randomizer. The taint policy generator and pointer tracker are implemented as two Pintools for `pin-2.14-71313`, while the randomizer is implemented as a shared library that is dynamically loaded by the pointer tracker into the child process.

A. Policy Generator

1) Instruction Abstraction: In the automated policy generation phase, our Pintool learns how x86 instructions "behave" given a certain combination of operands. For this purpose, we are only interested in how they behave with respect to pointer creation, modification, erasure, and propagation. Therefore, the created policy is an abstraction of instruction behavior that is just enough to accomplish accurate taint tracking. Our experiments have shown that, in most cases, it suffices to abstract an instruction to only its opcode, number of operands, type of operands (immediate, register, memory), width of each operand (in bits), flag bits (only for conditional instructions), and which operands were tainted before execution of the instruction. For this exact combination, an entry in the policy describes how it behaves with respect to tainted registers and memory after executing the instruction. For example, let the observed instruction be `mov RDI, RSP` with two operands of 64 bit register type. The second operand `RSP` is tainted, because it is a stack pointer. One specific policy entry for this exact combination (opcode=`mov`, operands=`register64,register64`, taint=false, true) specifies that after execution the first operand will be tainted. This makes sense, as `mov` actually copies the value of the second operand into the first operand.

#	Instruction	Input	Output	Category
1	`add reg1, reg2`	reg1=1 and reg2=0	reg1?=1	arithmetic
2	`sub reg1, mem1`	reg1=1 and mem1=0	reg1?=1	arithmetic
3	`add mem1, reg1`	mem1=0 and reg1=0	mem1?=1	arithmetic
4	`add reg1, mem1`	reg1=0 and reg2=0	reg1?=1	arithmetic
5	`and reg1, imm1`	reg1=1 and reg2=0	reg1?=1	bitwise

TABLE I: Ambiguous policies found in tested programs. An implicit instruction may either generate or remove a pointer (i.e., its output is not deterministic), even the provided taintness inputs are the same. 1: pointer; 0: non-pointer

2) Ambiguous Policy: There are special cases for which the aforementioned tuples of operand types, amount of operands, operand sizes and taints are not unambiguous enough to make a statement about taints after the execution of an instruction. Table I lists the ambiguous cases found in the tested programs. Case 5 is easy to understand, since a bitwise `and` operation could be used to compute either the base address (i.e., a pointer) or an offset (i.e., a non-pointer) depending on the value of the other operand. For example, assume `RAX` in `and RAX, 0x????????` contains a pointer and is therefore tainted. If the bitmask of the second operand exhibits many set MSBs, e.g., `and RAX, 0xfffffffffffff000`, the result will be the base address of the provided pointer in `RAX`. Vice versa, setting a few LSBs (e.g., `0x0000000000000fff`) calculates the offset into something. Our experiments with real-world programs show that the only use of a bitwise `and` instruction with a pointer is to either calculate the base or an offset. This confirms similar findings by [11] and [46].

Case 1 - case 4 indicate that ambiguous policies also exist for arithmetic instructions – a pointer could be added or subtracted by an offset to generate either a new pointer (e.g., relocation) or a non-pointer. To understand these special cases, we performed an in-depth analysis (on Ubuntu `14.04.2`). Note that RUNTIMEASLR provides a debugging mode that can print context information of executed instructions to ease the analysis.

Case 1. At the beginning of execution in loader, instruction `add r12, rcx` adds the pointer saved in `rcx` with a constant `0x3800003d8`, which generates a "non-pointer" (it does not point to mapped memory). However, we found that this non-pointer is added back to generate a new pointer with constant `-6FFFFEF5`×`8` in instruction `mov qword ptr [r12+rax`×`8], rdx`.

Case 2 - 4. We found that there is an instruction `sub rdx, qword ptr [r12]` that destroys a pointer by subtracting it with an offset. Before execution, `rdx` points to the base of `vDSO`, which is then subtracted by constant `0xffffffffff700000` to generate a value pointing to unmapped memory. Interestingly, we found case 3 and case 4 are paired to case 2, which re-assemble the pointers from the destroyed one in case 4. For example, case 3 is the instruction `add qword ptr [r12+0x348], rdx)` that adds the destroyed pointer by constant `0xffffffffff700fbd` to generate a new pointer still pointing to `vDSO`. We believe the arithmetic

operations in case 2, 3, and 4 are used to perform a very simple pointer protection employed by vDSO, which is however not found in other normal programs.

Based on the analysis, we generally cope with these ambiguous instructions by the simple range-checking approach: we check whether the result after executing the instruction points into valid memory, if it does, it is a pointer; otherwise, it is a non-pointer. The check only needs to be lazily performed for those ambiguous cases that are clearly marked in the policy file. For ambiguous cases with bitwise operations, range-checking can easily differentiate between a base address and an offset. For ambiguous ones with arithmetic operations, no matter what unrecognizable representation a pointer is transformed into, it will be recognized again by range-checking if it is changed back to the pointer. Therefore, the simple range-checking approach can generally handle all these ambiguous cases in a lazy manner.

3) Hidden Pointers in SSE Registers: Streaming SIMD Extensions (SSE) is an x86 instruction set extension that provides 128 bit registers `xmm0-xmm15` and and 256 bit registers `ymm0-ymm15`. In theory and good practice, 64 bit pointers of the 64 bit Intel x86 architecture should be stored in its general purpose registers (`RAX, RBX, ..., r1, r2, ...r15`) or in segment registers (`FS, GS`). However, our policy generation showed that these 128 bit SSE registers may also be used to store pointers, e.g., Nginx web server. Fortunately, those pointers were always stored either in the first 64 of the 128 bits (upper half) or in the last 64 bit (lower half). Based on this observation, RUNTIMEASLR treats a SSE register as a multiple of non-overlapping 64 bit general purpose integer registers. Whenever these SSE register are used, each 64-bit of their content is checked by RUNTIMEASLR for 64 bit pointers.

4) Mangled Pointers: Interestingly, the GNU standard C library (glibc) that is linked to every C or C++ program performs pointer obfuscation for some syscalls. It is implemented as a simple XOR operation against a random but fixed value stored relative to the segment register `FS` (see Figure 5).

```
#define PTR_MANGLE(reg) \
  xor %fs:POINTER_GUARD, reg; \
  rol $2*LP_SIZE+1, reg

#define PTR_DEMANGLE(reg) \
  ror $2*LP_SIZE+1, reg; \
  xor %fs:POINTER_GUARD, reg
```

Fig. 5: Pointer mangling and demangling in libc. The XORing key is hidden by using `fs` segment register. LP_SIZE is 8 on 64 bit platform.

To cope with mangled pointers, we demangle the pointers and check if the demangled values point inside valid mapped memory. This is possible because the XOR value is accessible by the process itself (stored at `fs:POINTER_GUIARD`), but not accessible for attackers residing outside the process. Should the demangled value reveal a pointer, the stored taint is augmented by meta information indicating a mangling according to glibc's method. This meta information leaves room for future, other methods of pointer obfuscation to be implemented. Currently, all tested programs compiled with gcc only exhibit the pointer obfuscation using the `PTR_MANGLE` macro shown in Figure 5.

B. Pointer Tracker

The pointer tracker itself is also implemented as a Pintool. As alluded to earlier, the only initial pointers after process creation are `RSP`, `RIP`, and some initial pointers in stack. In particular, we found `71` initial pointers in stack with the help of multi-run pointer verification. As the kernel routinely loads programs in the same way, we confirmed that these initial pointers have fixed relative locations in stack. From these initial pointers, further pointers are derived using the generated policy, which is applied to the affected instructions specified in the policy set. The policy set is loaded into an STL `list` container. We then use Pin's functionality to hook instructions based on their opcode, operand types and widths of operands. The hooking is realized in Pin by inserting a call before and after each instruction in question. Control flow is then diverted from the main executable code or library code into Pin and thereby into our pointer tracking mechanism. The pointer tracking mechanism fetches information about the used operands (register and/or memory), especially whether they are tainted and their current value. After applying the policy, which results in further or fewer taints for registers or memory, execution continues normally at the original position in memory. Unmatched instructions will not be hooked and execute normally.

1) Syscall Handling: Since RUNTIMEASLR is designed as a user space tool, all syscalls are a black box operation to RUNTIMEASLR. However, we are only interested in whether a syscall modifies a pointer or creates a new one. We found that most syscalls actually do not generate new pointers or propagate pointers. We manually evaluated the visible effects of the 313 Linux syscalls in user mode. Only `15` of them generate or propagate user mode pointers. Some representative ones are detailed as below.

- `mmap` creates new memory space and returns the base. We taint its return value when it returns.
- `munmap` unmaps the given memory space. In this case, we untaint the corresponding pointers that point to the unmapped memory.
- `mremap` moves an existing memory space to a new one and returns the new base address. In this case, we update the corresponding pointers pointing to the old memory space with the moved offset. Also we taint the returned value.
- `mprotect` changes the access permissions of memory. We update the tracked pointers according to how permissions are changed. For example, if the memory is changed from readable to non-readable, we untaint the corresponding pointers.
- `brk` either obtains current program break (by providing the first parameter with 0) or changes the location of program break. In the former, we simply taint its return value; while in the latter, if it shrinks the memory, we untaint the corresponding pointers and also taint its return value.
- `arch_prctl, etc.` There are also some syscalls that may load existing pointers of user space. For example, `arch_prctl` can be used to get base values in segment

register `fs` or `gs`. For these cases, we taint the obtained pointers in return value or parameters.

2) Pointer Scope and Memory Deallocation: Whenever a called function returns, the local variables of the callee are no longer valid, i.e., they have left their scope. Such local variables may also contain pointers. Hence, it is vital to remove any associated taint of pointers that have gone out of scope. The memory region that contains variables that have gone out of scope is considered uninitialized memory, since reading from that memory is non-deterministic as it depends on which function was called last. A good compiler will catch that type of uninitialized access at compile-time and warn the developer about it.

To prevent unpredicted behaviors, we detect out-of-scope pointers when a function returns so that they can be untainted. To this end, we hook the `ret` instruction and on each return calculate the currently valid stack frame by inspecting `RSP`. By the x86-64 calling convention, the stack pointer `RSP` must point to the top of the valid stack after returning from a function, therefore, all data stored above `RSP` can be untainted.

The same applies to objects on the heap: after a call to `free()` (`delete()` also calls `free()`), pointers stored in that now deallocated area become invalid. We therefore remove all taints of pointers that are stored in a `free`'ed area. A caveat is that the call to `free` does not include any information about the size of the to-be-deleted object. However, the memory in question was once allocated using `malloc` with a specific size. We therefore use a map that associates a length obtained from hooking `malloc` to each heap pointer. Whenever, `free()` is called, we look up the size for the specified base pointer and untaint all pointers stored in the deallocated area.

C. Randomizer

The randomization is done in module-level, which is better than the "process-level" randomization provided by Linux' ASLR implementation. Finer-grained randomization is possible, as we have all pointer information. However, even position-independent code assumes fixed offsets between code and data. Finer-grained randomization may require patching these offsets, and thus introduces more overhead. The actual remapping from an old address (cloned from the parent) to a new random address is achieved by calling the syscall `mremap` with a random base address. We obtain the cryptographically secure randomness from `/dev/random`. In the current implementation, we set the default entropy to the one provided by default Linux OS, which is 28 bits. However, we also provide a configuration switch for the entropy so it can be set up to 48 bits (the full canonical address space on Intel x86-64 platforms). Note that when `heap` is remapped, we also need to adjust `program break` by calling the syscall `sbrk()`. The protection flags and size of the new mapping are simply taken from the old one to preserve all semantics. After remapping, we patch the pointers accordingly (in both memory and registers).

The final step is context switching from Pin to program. The context includes not only general registers but also SSE registers (e.g., `xmm0-xmm15`) and segment registers (e.g., `fs and gs`). As all pointers are patched when the address space is re-randomized, the context is naturally updated accordingly. Before Pin's mappings are unmapped, the context data is copied to the stack of the program. To load the context into the corresponding registers (after unmapping Pin), we have to use handwritten assembly, as no library functions are available and we cannot even use Pin's stack and heap. Special care must be taken for loading values from stack into SSE registers (i.e., `xmm0-xmm15`) as those memory addresses must be 16-byte-aligned in order to not trigger a general protection fault (`#GP`). Finally, we use an indirect `jmp` to transfer the control back to program and continue the original execution of child process. At this point, Pin is completely unmapped and as the child process is no longer instrumented, it runs with its native performance.

1) Recursive Forking: So far, we have described that `fork`'ed children are no longer instrumented in order not to suffer from performance penalties. However, one cannot know if child processes will or will not fork their own children. If they do, there would not be any pointer tracking information because it was intentionally disabled for performance reasons. To overcome this, we provided a feature that records which layer of processes of a program typically `fork` their own new processes. For example, the Nginx web server first forks a daemon process, and then that daemon process forks worker processes for each CPU core. Therefore, we can configure which layer of children will disable their instrumentation which will not.

VII. Evaluation

In this section we evaluate the **correctness**, **effectiveness**, and **efficiency** of RuntimeASLR. To this end, we have developed the following four evaluations:

1) A *theoretical* and *practical* analysis of the accuracy of identifying pointers.
2) A memory *snapshot* analysis to compare the correctness of re-randomization with the help of load-time randomization.
3) An empirical test of real clone-probing attack prevention.
4) A performance evaluation of re-randomized child processes and microbenchmarks of `fork` overhead and pointer tracking in the parent process.

Experiment setup. We applied RuntimeASLR to the Nginx web server, as it is one of the most popular web servers and has been an attractive attack target [7], [21], [35], [9], [20]. We chose Nginx in version `1.4.0`, as this version contains a stack buffer overflow vulnerability that was also exploited by Hacking Blind [7]. This way, we can later show the effectiveness of our solution in the presence of a real vulnerability. We use Nginx web server to evaluate the correctness, effectiveness, and performance of RuntimeASLR. Further, we applied RuntimeASLR to SPEC CPU2006 benchmarks to evaluate the performance of the pointer tracking component. All experiments were conducted on a Dell OptiPlex 390 equipped with an Intel® Core™ i3 2120 running at 3.30 GHz and 8 GiB of RAM. This particular processor has two physical cores each of which supports *Simultaneous Multithreading*, which presents itself as four cores to the operating system. For operating system, we used an unmodified fresh install of the latest Ubuntu long-term support release `14.04.2`.

A. Correctness

We first evaluated the correctness of RuntimeASLR by analyzing its accuracy in pointer identification.

1) Theoretical Analysis: In this section, we present an upper bound for false positive pointer detection. As already described in subsection III-A, the probability for a random integer to be mistaken as a pointer is $p = b \cdot 2^{-64 \cdot n}$ for b bytes being mapped into a process' address space and n-runs are performed in multi-run pointer verification. Of course, this only applies to a single pointer identification. As every executed instruction is inspected for potential pointers, in the worst case, every inspected instruction might introduce a new random integer that could be mistaken for a pointer. So in the worst case, the entire mapped memory only consists of instructions that handle 64 bit integers. Even though it is technically not possible due to encoding, a maximum of b instructions can exist in b bytes of memory. Therefore, the probability of identifying at least one integer coincidentally as a pointer is

$$1 - (1-p)^b = b \cdot p - \binom{b}{2} \cdot p^2 + \binom{b}{3} \cdot p^3 - \cdots - p^b$$

according to the binomial probability. Assuming $b < 2^{47}$ and n $>= 2$, then $\binom{b}{i} \cdot p^i$ must be significantly larger than $\binom{b}{i+1} \cdot p^{i+1}$, where $1 < i < b-1$ and $p^b > 0$. Therefore, this probability can be safely simplified to:

$$b \cdot p = b^2 \cdot 2^{-64 \cdot n}$$

To fill in some numbers, we can assume that 100 MB of the address space are used, and 2 runs are performed in multi-run pointer verification. This yields a false positive rate of 2^{-76}, which is negligible.

2) Memory Analysis: Although the protected programs do work successfully, we want to thoroughly check the correctness of RUNTIMEASLR in practice. To this end, we evaluate the false positive and false negative rate of pointers and check if the address space is re-randomized correctly.

a) False pointer checking: The goal of false pointer checking is to scan the entire address space for 8-byte pointers and make sure that our tracking did find all (no false negatives) and not too many (no false positives) pointers. Checking is done *right before* re-randomization, when it is crucial that all pointers have been tracked correctly. To be fair, multi-run pointer verification was used during the analysis to check false positive pointers. For false negative pointers, we check each 8-byte value in the readable memory to see if it points to mapped memory. We ran the Nginx web server and hooked `fork()` for the memory analysis. This revealed a total of 7952 pointers were tracked with no false positives, i.e. all tracked pointers indeed point into valid mapped memory. The analysis further revealed four "false negatives", i.e. some readable 8-byte values that point into valid memory had not been tracked. However, we found that those *"pointers"* were located in freed heap objects and hence were no longer valid. Since Nginx did not clear the data when freeing heap objects, the out-of-scope pointers remained in memory.

b) Memory space consistence checking: It is of paramount importance that our forcefully imposed re-randomization at run-time is indeed correct. To verify this, we compare the address space *right after* re-randomization to legitimate address space before forking, but at the same address. This verification process is two-fold. First, the program is run with ASLR enabled so that it is executed at a random address. At the moment a `fork()` syscall is issued, we take a memory snapshot of the entire address space, record base addresses of mappings, and abort the program without re-randomization. Second, we run the program again, which results in a different address, but this time let the re-randomization happen after `fork()` is called. However, the re-randomization of our Pintool is instructed to use the recorded addresses of each mapped block of memory instead of true randomness. This way, the address space of our re-randomization should look exactly like the address space of a properly randomized ASLR process. After that, we compare the remapped memory with the previously dumped memory. Our results show that all mappings are exactly the same, which indicates that all pointers are correctly patched. Please note that in the first ASLR run, the program is run with Pin enabled because our comparison (re-randomization) will have Pin enabled as well. Otherwise comparisons would differ in the mapped memory of Pin.

c) Some Interesting Policies: Unlike previous pointer tracking approaches (e.g., [46], [11]) that only empirically cover a handful of taint policies (e.g., assignments, addition/-subtraction, and bitwise `AND`), we have automatically discovered a total of **342** taint policies that operate pointers. Table II lists some that we think are interesting for the reader.

#	Instruction	Input	Output
1	`rdtsc`	N/A	rax=0 and rdx=0
2	`cpuid`	rax,rbx,rcx,rdx=1	rax,rbx,rcx,rdx=0
3	`and rcx, rax`	rcx=1 and rax=0	rcx?=1
4	`neg rcx`	rcx=1	rcx=0
5	`add rcx, [rax+0x28]`	rcx=0 and mem=0	rcx?=1
6	`mov eax, edi`	rax=1 and rdi=1	rax=0
7	`rol rax, 0x11`	rax=1	rax=0
8	`lea rdx, [rip+0x21e3cc]`	rdx=0	rdx=1
9	`shr rax, 0x2`	rax=1	rax=0
10	`leave`	rsp=1 and rbp=1	rsp=1 and rbp=1
11	`movdqu xmm8, [rsi]`	xmm8=10 and mem=00	xmm8=00
12	`pslldq xmm2, 0x5`	xmm2=01 and mem=00	xmm2=00

TABLE II: Selected interesting taint policies that are hard to identify based on heuristics. Policies are stored in compact manner. Some properties are omitted. 1: pointer; 0:non-pointer; mem: the memory read or written; ?: ambiguous policy.

B. Security

1) Address Space Analysis: RUNTIMEASLR performs module-level re-randomization on the child process. RUNTIMEASLR provides `mremap` with cryptographically secure randomness obtained from `/dev/random`. In this evaluation, we set the entropy of randomness to be 28 bits – which is the default entropy of Linux' default ASLR. The generated random base address is of the form `0x7f???????000`, where `?` bits are randomized. The security provided by RUNTIMEASLR is the re-randomization of child processes' address space, which consists of two parts: (1) remapping all modules of protected program to randomized addresses; (2) all other modules (e.g., of Pin and Pintool) are unmapped. To evaluate the security, we run Nginx web server with RUNTIMEASLR. We configured the number of worker processes to be 4, which is the default number suggested by Nginx and is exactly the amount of physical CPU cores of the test machine. We then verify the re-randomized memory mappings of each worker process by checking `/proc/worker-pid/maps`. We empirically confirmed that the question-marked bits in `0x7f???????000` are indeed continuously randomized. The relative addresses between

```
=================
Assuming ASLR
Stack reading
Testing 18 - Found 18
Testing 71 - Found 71
Testing 86 - Found 86
Testing e4 - Found e4
Testing 34 - Found 34
Testing 7f - Found 7f
Testing 0 - Found 0
Testing 0Connect timeout
Connect timeout
Testing ff
Not found... damn - trying again
Testing 0 - Found 0
Stack has 0x7f34e4867118
.text ptr 7f34e4867118
Pad 0 Ret 7f34e4867118
PLT AT 7f34e4867118
```

(a) Nginx w/o RUNTIMEASLR

```
=================
Assuming ASLR
Stack reading
Testing 78 - Found 78
Testing 22 - Found 22
Testing ff
Not found... damn - trying again
Testing ff
Not found... damn - trying again
Testing ff
Not found... damn - trying again
Testing ff
Not found... damn - trying again
Testing ff
Not found... damn - trying again
Testing ff
Not found... damn - trying again
Testing ff
Not found... damn - trying again
```

(b) Nginx w/ RUNTIMEASLR

Fig. 6: ASLR bypassing (clone-probing) attack against Nginx web server with stack reading of Hacking Blind.

different modules are also randomized, which further improves the effective entropy [23] of default Linux ASLR. We also confirmed that all other mappings (of Pin and Pintool) are not shown in `/proc/worker-pid/maps`, and hence have been completely unmapped. Based on these results, we conclude that the expected security is indeed achieved by RUNTIMEASLR.

2) Real Exploit Testing: To further verify the effectiveness of RUNTIMEASLR, we also test if RUNTIMEASLR can defeat real clone-probing attack. Fortunately, the Hacking Blind attack (`BROP`) [7] is publicly available[3]. We downloaded the provided exploit and ran it without any change against `Nginx 1.4.0`. As per `BROP` website, the exploit consists of several steps, including ASLR bypassing with stack reading, locating the required gadgets, and building the ROP shellcode. The foremost step is to efficiently bypass ASLR. The approach proposed in `BROP` is so-called stack reading. As already mentioned in section I, it overwrites the return address on stack byte-by-byte with a single guessed byte. As the newly forked worker process has the same address space as the crashed one, the probing can be repeatedly performed for each byte.

To verify the effectiveness of RUNTIMEASLR, we ran Nginx with and without RUNTIMEASLR. The results are shown in Figure 6. In the case of Nginx without RUNTIMEASLR, stack reading can successfully bypass ASLR within two minutes. However, if RUNTIMEASLR is enabled, we had to manually terminate the exploit after 10 hours, as it could not even guess 3 bytes. The reason why stack reading does not work under RUNTIMEASLR is apparent: the correctly guessed bytes keep changing in later trials due to re-randomization.

C. Performance

In order to test the performance of our approach, we conducted two independent experiments to be able to measure the performances of the instrumented parent process (with heavy-weight pointer tracking) and uninstrumented child (i.e., worker) processes.

1) Performance with Web Benchmarks: To measure the child process performance, we used the `Nginx 1.4.0` web server and measured different performance aspects. The performance slowdown that might potentially be introduced by our RUNTIMEASLR is measured in comparison to an unmodified system as baseline. Nginx used four worker child processes – one pinned to each core. The client to measure network performance was running on a second machine, which was connected to the Nginx web server over a `1000BASE-T` (Gigabit Ethernet) switch using a $10m$ long `CAT 5e` cable. We measured two dimensions, *connection concurrency* performance and *throughput* performance to check if either of them is affected by RUNTIMEASLR.

Connection concurrency measures the round-trip time in relation to the number n of simultaneously connected clients, i.e., the time it takes from the beginning of a request until the complete response has been received from the server. The more clients that are requesting resources from the server at the same time, the longer it takes the server to send back the entire response. To measure the connection concurrency, we used the *Apache Benchmark* (`ab 2.3`) tool. `ab` lets each of the n simultaneous connections request 10,000 documents from the server as fast as possible.
Throughput in turn measures the rate of successfully received bytes per time unit. For this purpose, we let Nginx deliver a 1 GiB[4] file 100 times and averaged the download speed.

As a test bed, we used HTML documents obtained from real web sites. We mirrored the top 10 Alexa [2] websites into our Nginx root document root directory using `wget`. Performance results showed that the response times only loosely correlate to the size of the delivered content (see Figure 7). Since our performance slowdown evaluation is interested in the RUNTIMEASLR performance in relation to the baseline, the document size and content did not affect the results as the quotient of $\frac{\text{RUNTIMEASLR}}{\text{baseline}}$ was unaffected. For the remainder of the concurrency tests, we therefore picked a random top 10 web site to test with: Number 6, Amazon.com (442.1 kB of HTML, CSS and JavaScript).

Fig. 7: Response time for single real-world documents retrieved from our Nginx webserver using 30 concurrent connections. Content mirrored from Alexa Top 10.

Concurrency Results: We tested the concurrency performance of the RUNTIMEASLR-enabled Nginx with 10, 20, 30, 40 and 50 concurrent clients and there was no measurable performance overhead. Figure 8 shows the median of 10,000 response times with the number of concurrent connections $n \in 10, 20, 30, 40, 50$. The worst measured performance difference was for 10 simultaneous connections and is a

[3] http://www.scs.stanford.edu/brop/

[4] We use the notation of a *Gibibyte* (1 GiB $= 1024 \times 1024 \times 1024$ bytes) instead of a *Gigabyte* (1 GB $= 1{,}000^3$ bytes)

slowdown of 0.51%, which is well within the jitter margin of the experiment.

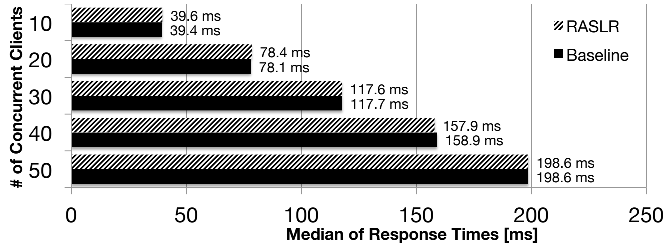


Fig. 8: Nginx response time with different concurrencies.

The overall distribution of response times is depicted in Figure 9, which shows that the majority of response times is $\approx$ 200 ms for 50 simultaneous connections and $\approx$ 40 ms for 10 simultaneous connections. The performance difference between RUNTIMEASLR and the baseline is not measurable.

Throughput Results: We also measured the average and peak throughput for a large single file download over HTTP using only one connection at the same time. The throughput result for 100 downloads of a 1 GiB file is shown in Table III. The measured difference is only 0.5% slowdown on average and a mere 0.1% slowdown for the peak throughput.

	Throughput	
	Average	**Peak**
Baseline	870.5 MBit/s	898.1 MBit/s
RUNTIMEASLR	866.4 MBit/s	897.2 MBit/s

TABLE III: Throughput performance of HTTP file download with RUNTIMEASLR enabled and without (baseline)

2) Micro Benchmark of Fork: In traditional UNIX systems and Linux, the `fork()` syscall is tuned to be very fast. We measured the performance overhead of our instrumented `fork` to give the reader an insight of introduced delays. Table IV shows the micro-benchmarks for the instrumented `fork()` with RUNTIMEASLR. Without RUNTIMEASLR, forking is extremely efficient, taking only 0.1 ms. RUNTIMEASLR performs Pin detaching, address space remapping, Pin unmapping, and context switching for `fork()`. While unmapping and context switching is efficient, the detaching of Pin seems to be a bottleneck. Although percentage of performance overhead is significant, the absolute overhead is less than 150 ms. More importantly, most daemon processes, e.g., Nginx web server and Apache web server, only fork worker processes once and delegate work instead of creating a new process for each work item, so the performance overhead of `fork()` with RUNTIMEASLR only affects the starting time of these servers.

	Detach	Remap	Unmap & Context switch	Total fork
Fork w/ RUNTIMEASLR	109.7 ms	27.0 ms	0.8 ms	137.5 ms

TABLE IV: Micro benchmark for fork() with RUNTIMEASLR. The original fork() takes 0.1 ms. Worker processes are pre-forked once at startup in many daemon servers.

Benchmark	Original (seconds)	RUNTIMEASLR (seconds)	Overhead (times)
gcc	1.13	12,783	11,312
mcf	2.64	24,708	9,359
hmmer	2.28	19,004	8,335
libquantum	0.06	1,491	24,850
xalancbmk	0.08	1,932	24,150
soplex	0.02	217	10,850
lbm	2.28	2,468	1,028
sphinx3	1.61	12,661	7,863

TABLE V: Pointer tracking on SPEC CPU2006 benchmarks. Pointer tracking is completely detached in child process, and thus does not affect the performance of child (worker) process.

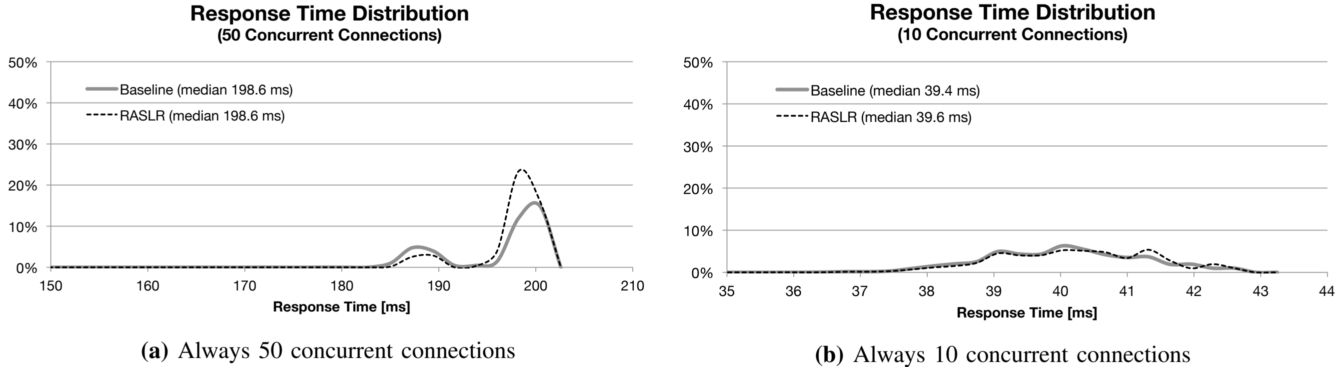
3) Performance of Pointer Tracking: RUNTIMEASLR is mainly used to prevent clone-probing attacks targeting daemon servers. As shown in VII-C3, RUNTIMEASLR imposes no overhead to the web service itself. Here, we also want to evaluate the performance of the pointer tracking component. We first measure the time for Nginx web server to start. On our machine, it finishs starting within `35` seconds. We then apply RUNTIMEASLR to SPEC CPU2006 benchmarks which contains some relatively complicated programs, e.g., gcc. Since these benchmark programs do not adopt daemon-worker scheme, pointer tracking will be performed for all executed code. In this evaluation, all the benchmark programs are compiled with options `-pie -fPIC -O2`. Table V shows the results. The significantly reduced performance is not surprising, as RUNTIMEASLR performs runtime taint analysis to accurately identify all pointers. However, pointer tracking is only performed in the parent process at startup. The child process, the actual worker process, is not affected by pointer tracking (see VII-C1). For the long-running daemon servers, the starting overhead introduced by "one-time" pointer tracking before `fork` is acceptable, but the performance of its provided service is more critical, and in our case not affected. One may also wonder about the performance of taint policy generation, although it is less concerned than pointer tracking, since it is performed offline. The running time for taint policy generation aginst Nginx is about 30 seconds.

VIII. DISCUSSION

In this section, we discuss some potential problems with RUNTIMEASLR, which might appear in special cases.

A. Soundness of Taint Policy Generation

The taint policy generation mechanism of RUNTIMEASLR is not sound in general, since the instruction abstraction may generate ambiguous policies. In Section VI-A2, we performed an in-depth analysis to understand how ambiguous policies are introduced in the tested programs. From that analysis, we learned that normal code emitted by compilers usually process pointers in an universal manner; however, special programs (e.g., dynamic loader and glibc) may process pointers specially, resulting in ambiguous policies (about 1% out of all policies). Although the simple range-checking approach (see Section VI-A2) is sufficient to handle the ambiguous cases listed in Table I, manual analysis may still be required to confirm the correctness of taint policies when more ambiguous cases are reported by RUNTIMEASLR. If the ambiguous policies cannot



(a) Always 50 concurrent connections

(b) Always 10 concurrent connections

Fig. 9: Nginx response time distribution with/without our pointer tracking `Pin` instrumentation.

be handled by range-checking, one may need to manually define the special taint policies. Alternatively, an automated approach to handle ambiguous policies is to relax the instruction abstraction. For example, including the effective width (or the highest order of bit) of the operands can remove ambiguous case 5 in Table I. A tradeoff needs to be made between the degree of abstraction and pointer tracking performance – relaxing the abstraction will introduce further performance overhead in pointer tracking.

B. Completeness of Taint Policy

The pointer tracking of RUNTIMEASLR consists of the offline taint policy generation and runtime taint tracking. In our experiments, because the sample programs used for generating taint policy include the target program, and the inputs for two runs for policy generation and taint tracking are exactly the same, the policy set is usually complete for pointer tracking in practice – we did not meet any missed policy in Nginx. However, we cannot guarantee the 100% completeness of policy set, as the two runs of program for policy generation and taint tracking are independent. For example, if the system administrator changes the Nginx configurations at runtime (i.e., after policy generation), new code paths not covered during policy generation may be introduced, thus may result in false negative policy. Assuming such cases exist, we can either manually add the reported new policies – as the total number of x86_64 instructions is limited[5] – or adopt an automatic approach: if we find an instruction generates, updates, or removes a pointer, but is not covered in existing policy set, we can append it to the policy set at runtime.

C. Applicability for General Programs

A program that does not adopt the daemon-worker scheme is not vulnerable to clone-probing, so it is not necessary to use RUNTIMEASLR. As specified in threat model in subsection II-A, RUNTIMEASLR is dedicated to server programs that *pre-fork* worker processes processing users' actual requests and perform light-weight tasks (e.g., worker process management) in daemon process, e.g., web servers. RUNTIMEASLR performs runtime taint analysis to accurately identify all pointers in parent (daemon) processes, so the performance of parent process is dramatically reduced. Current implementation of RUNTIMEASLR is not suitable for server programs that perform heavy-weight tasks in parent process for performance reasons. Regarding performance in pointer tracking, RUNTIMEASLR can be improved in two ways: (1) static code instrumentation using a compiler or binary rewriting can help improve the pointer tracking performance. It is worth noting that statically instrumented code is hard to be completely detached in worker processes, which is actually the main reason we chose to employ dynamic instrumentation; (2) as our primary goal is to make the pointer tracking accurate, our current implementation still has room for improvements to tweak its performance. Improving the performance of pointer tracking is a long-studied topic [9], [26], [28], we can employ these techniques to improve the performance of pointer tracking.

[5]http://ref.x86asm.net/coder64.html

D. Pointer Obfuscation

Pointer tracking in RUNTIMEASLR can generally handle pointer obfuscations, as encryption and decryption are symmetric, i.e., the encrypted pointers will be recognized when they get decrypted. However, we encounter a problem when pointer patching happens on encrypted or otherwise obfuscated pointers. If we do not know how the pointers are encrypted, it is impossible to patch the encrypted pointers in memory; therefore, we assume all pointer obfuscations applied in the protected program must be known to RUNTIMEASLR. To better handle this, we provide a detection to help users identify pointer obfuscations. Note that, in the case of Nginx, we only found one such case – `glibc`'s mangled pointers.

IX. RELATED WORK

A. Pointer Tracking

The heart of runtime re-randomization is accurately tracking all pointers. Pointer tracking has been well-studied for object bounds checking for memory safety, and pointer protection. Existing works either employ type-based analysis or heuristics-based analysis to identify pointers. Type-based analysis [27], [40], [39], [30] statically infers the type information of an object (e.g., a pointer). It is efficient and easy to use; however, it suffers from a high number of false negatives. Pointers with non-pointer type (e.g., base addresses) and the ones prepared by the OS are not covered. Memcheck [46], Clause et al [11], and Raksha [15] empirically specify the pointer propagation rules to track pointers. Although they specified very detailed rules, it is

still difficult to cover all cases, due to the complexity of C/C++ programs. We propose an automatic mechanism to accurately identify all pointers (subsection VII-A). Some interesting cases that are not discussed in heuristics-based analysis papers are shown in Table II.

B. Re-randomization

Morula [32] is one of most related works, as it also targets the ASLR limitation with `fork`. Android adopts the Zygote model that starts app by forking, so that every app shares the same address space layout. Morula addresses this problem by simply maintaining a pool of pre-forked but re-randomized Zygote processes (template), so that when an app is about to start, one Zygote process is picked, and it starts execution from its entry point. Unfortunately, in our case of daemon processes, its child processes are forked on the fly, which starts execution from the point after `fork()` rather than the entry point. The semantic-preserving requirement of re-randomization makes RUNTIMEASLR fundamentally more difficult than Morula. We cannot simply `fork-exec` the child process; rather, have to perform runtime re-randomization. Similarly, ASR [22] performs re-randomization at load-time, so not all semantics are preserved. Isomeron [16] and Dynamic Software Diversity [12] dynamically choose targets for indirect branches at runtime to reduce the probability of predicting the correct runtime address of ROP gadgets. In their proposals, clone-probing attacks are still effective, as child processes still share the same memory layout. TASR [6] re-randomizes code sections after a pair of socket read/write. It requires compiler support and cannot protect data pointers.

C. Code and Data Diversification

Code diversification [42], [31] makes multiple copies of the code, which preserves the semantics. An assumption of Code diversification is that the attacker cannot get the same copy of the code. This assumption does not hold when there are memory *overread* vulnerabilities in the program [47]. Data layout randomization [5], [10], [34] mitigates buffer overrun by making the offset between the target data (e.g., return address) and overflow point unpredictable. However, clone-probing attacks can still work under code or data diversification, since the child processes still share their parent's layout.

D. Fine-grained ASLR and Control-flow Integrity

Fine-grained ASLR [22], [29], [4], [48], [24], [18] and Control-flow Integrity [50], [51], [1], [17], [41], [37], [38] aim to make the exploit more difficult after ASLR is bypassed. Complementarily, RUNTIMEASLR aims to defend against the first step– bypassing ASLR. Therefore, RUNTIMEASLR is orthogonal to these techniques.

X. CONCLUSION

A fundamental limitation with ASLR is that the forked child processes always share the same address space layout as their parent process. This has resulted in clone-probing attacks. We propose RUNTIMEASLR, the first approach that prevents clone-probing attacks without altering the intended semantics of child forking. RUNTIMEASLR consistently re-randomizes the address space of every child after `fork()` at *runtime* while keeping the parents state. The heart of RUNTIMEASLR is an automatic, systematic, and holistic pointer tracking mechanism, which we believe is a useful tool for future researches on pointer identification and protection. Our evaluation results on Nginx web server show that RUNTIMEASLR can correctly identify all pointers and effectively prevent clone-probing attacks. More importantly, RUNTIMEASLR imposes no performance overhead to the provided service (after pre-forking).

ACKNOWLEDGMENT

We thank the anonymous reviewers and David Wagner for their valuable feedback, as well as our operations staff for their proofreading efforts. This research was supported in part by the BMBF-funded Center for IT Security, Privacy and Accountability (CISPA). Kangjie Lu and Wenke Lee were supported in part by the NSF award CNS-1017265, CNS-0831300, CNS-1149051 and DGE-1500084, by the ONR under grant N000140911042 and N000141512162, by the DHS under contract N66001-12-C-0133, by the United States Air Force under contract FA8650-10-C-7025, by the DARPA Transparent Computing program under contract DARPA-15-15-TC-FP-006. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, ONR, DHS, United States Air Force or DARPA.

REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM Conference on Computer and Communication Security*, 2005.

[2] Alexa Internet, Inc., "Top 500 Global Sites," http://www.alexa.com/topsites.

[3] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 21st ACM conference on Computer and communications security*, 2014.

[4] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *23rd USENIX Security Symposium*, Aug. 2014.

[5] S. Bhatkar and R. Sekar, "Data space randomization," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08, 2008.

[6] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM Computer and Communications Security (CCS'15)*, Oct 2015.

[7] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2014.

[8] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev, "Address space randomization for mobile devices," in *Proceedings of the Fourth ACM Conference on Wireless Network Security*, ser. WiSec '11, 2011.

[9] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'11, 2011.

[10] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries," in *Proceedings of the 2015 Network and Distributed System Security Symposium*, ser. NDSS '15, 2015.

[11] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, 2007.

[12] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *22nd Annual Network & Distributed System Security Symposium*, 2015.

[13] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *36th IEEE Symposium on Security and Privacy*, 2015.

[14] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz, "It's a TRAP: Table randomization and protection against function reuse attacks," in *Proceedings of the 22nd ACM conference on Computer and communications security*, ser. CCS '15, 2015.

[15] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, 2007.

[16] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *22nd Annual Network & Distributed System Security Symposium*, 2015.

[17] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium*, 2014.

[18] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *8th ACM SIGSAC symposium on Information, computer and communications security (ACM ASIACCS 2013)*. ACM, 2013, pp. 299–310.

[19] I. Dillig, T. Dillig, and A. Aiken, "Reasoning about the unknown in static analysis," *Commun. ACM*, 2010.

[20] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14, 2014.

[21] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[22] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[23] W. Herlands, T. Hobson, and P. J. Donovan, "Effective entropy: Security-centric metric for memory randomization techniques," in *Proceedings of the 7th USENIX Conference on Cyber Security Experimentation and Test*, ser. CSET'14, 2014.

[24] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[25] Intel, *Intel 64 and IA-32 Architectures Software Developers Manual – Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*. Intel Corporation, 2025.

[26] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: Efficient parallelization of dynamic data flow tracking," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, 2013.

[27] R. Johnson and D. Wagner, "Finding user/kernel pointer bugs with type inference," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, ser. SSYM'04, 2004.

[28] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12, 2012.

[29] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proceedings of the 22Nd Annual Computer Security Applications Conference*, 2006.

[30] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[31] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[32] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From Zygote to Morula: Fortifying weakened aslr on android," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14, 2014.

[33] L. Li, J. E. Just, and R. Sekar, "Address-space randomization for windows systems," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, ser. ACSAC '06, 2006.

[34] Z. Lin, R. D. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," in *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '09, 2009.

[35] Long Le, "Exploiting nginx chunked overflow bug, the undisclosed attack vector," http://ropshell.com/slides/Nginx_chunked_overflow_the_undisclosed_attack_vector.pdf.

[36] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM conference on Computer and communications security*, ser. CCS '15, 2015.

[37] A. J. Mashtizadeh, A. Bittau, D. Mazieres, , and D. Boneh, "Cryptographically enforced control flow integrity," 2014, arXiv preprint arXiv:1408.1451.

[38] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *22nd Annual Network & Distributed System Security Symposium*, 2015.

[39] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[40] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, May 2005.

[41] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

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

[43] PaX Team, http://pax.grsecurity.net/.

[44] ——, "PaX Address Space Layout Randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt.

[45] J. Seibert, H. Okkhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 54–65.

[46] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, 2005.

[47] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE Symposium on Security and Privacy*, 2013.

[48] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary Stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[49] H. Xu and S. Chapin, "Address-space layout randomization using code islands," in *Journal of Computer Security*. IOS Press, 2009.

[50] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[51] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium*, 2013.