Code Randomization: Haven't We Solved This Problem Yet?

Stephen Crane
sjc@immunant.com

Andrei Homescu
ah@immunant.com

Per Larsen
perl@immunant.com

Abstract—**Two decades since the idea of using software diversity for security was put forward, ASLR is the only technique to see widespread deployment. This is puzzling since academic security researchers have published scores of papers claiming to advance the state of the art in the area of code randomization. Unfortunately, these improved diversity techniques are generally less deployable than integrity-based techniques, such as control-flow integrity, due to their limited compatibility with existing optimization, development, and distribution practices.**

This paper contributes yet another diversity technique called pagerando. Rather than trading off practicality for security, we first and foremost aim for deployability and interoperability. Most code randomization techniques interfere with memory sharing and deduplication optimization across processes and virtual machines; ours does not. We randomize at the granularity of individual code pages but never rewrite page contents. This also avoids incompatibilities with code integrity mechanisms that only allow signed code to be mapped into memory and prevent any subsequent changes. On Android, pagerando fully adheres to the default SELinux policies. All practical mitigations must interoperate with unprotected legacy code; our implementation transparently interoperates with unmodified applications and libraries. To support our claims of practicality, we demonstrate that our technique can be integrated into and protect all shared libraries shipped with stock Android 6.0. We also consider hardening of non-shared libraries and executables and other concerns that must be addressed to put software diversity defenses on par with integrity-based mitigations such as CFI.

I. Motivation

Anyone who develops a new exploit mitigation face non-trivial trade-offs. Improving security often degrades (some aspect of) performance and vice versa. To complicate matters, mitigations have complex and often synergistic interactions. ASLR and CFI [1], for instance, both rely on DEP to prevent code injection. Finally, academic research tends to focus on normal program operation. Our experience, however, from interviewing and interacting with security engineers is that "auxiliary" processes such as distribution, updating, and debugging may prevent adoption of new and improved defenses.

CFI passed through the proverbial eye of the needle and is now a fully supported feature in mainstream C/C++ compilers such as MSVC, GCC, and LLVM. To keep performance overheads reasonable, only forward edges are checked but Intel has announced that hardware support for CFI is forthcoming. This will allow checking of backwards (return) edges as well.

With more and more code being protected by some variant of CFI, should we keep working on code-randomization defenses? We believe so, for the following reasons:

1) CFI and code randomization target complementary steps in the exploit kill chain. CFI prevents the control flow hijacking step which diverts execution to an attacker-controlled location. Code randomization, in contrast, makes it hard for an attacker to pick the "right" code locations that, when executed, achieve the exploit goals.
2) CFI and code randomization have disjoint weaknesses. Code randomization assumes that memory contents are kept secret from adversaries but various types of information leakage can undermine that assumption. CFI is secret-less but permits control flows not strictly needed for correct program execution, which leaves harmful control flows available to adversaries [2], [3], [4], [5].
3) Hardware capabilities may favor one solution over the other. Future high-end x86 processors will likely make CFI and even memory safety solutions run with tolerable overheads. Mobile processors, on the other hand, favor randomization-based solutions. Modern, 64-bit ARM processors support non-readable (execute-only) code pages which helps prevent information leakage [6], [7].

Goals and Contributions The remainder of this paper describes our efforts to design a code randomization defense with the same level of practicality as CFI and the other standard mitigations in widespread use today. Our technique distinguishes itself by being the first that meets the operational requirements for protection of shared libraries in the Android operating system while offering higher entropy and leakage resilience than existing defenses on that platform. In particular, our solution—**page**rando—is first to randomize code in shared libraries at the level of individual memory pages while still interoperating transparently with other types of code. The following sections explain the design of **page**rando (§ II), evaluate our techniques (§ III), and discuss a few remaining issues (§ IV).

II. Pagerando Design

The goal of **page**rando is to increase the randomization granularity over ASLR [8] while being equally efficient and deployable. ASLR, as implemented in Linux, simply maps the program binary into memory at a randomly chosen base address. While it is easy to fault ASLR for only randomizing the base address, this approach has a number of practical benefits. Most importantly, ASLR'ed code can be distributed and patched just like any binary. Similarly, since the on-disk binary is never modified, ASLR is transparent w.r.t. code signing and checksumming techniques as well. Finally, ASLR maps position independent code pages directly into virtual memory without modification so only one copy of the `.text` section of a shared library is kept in physical memory.

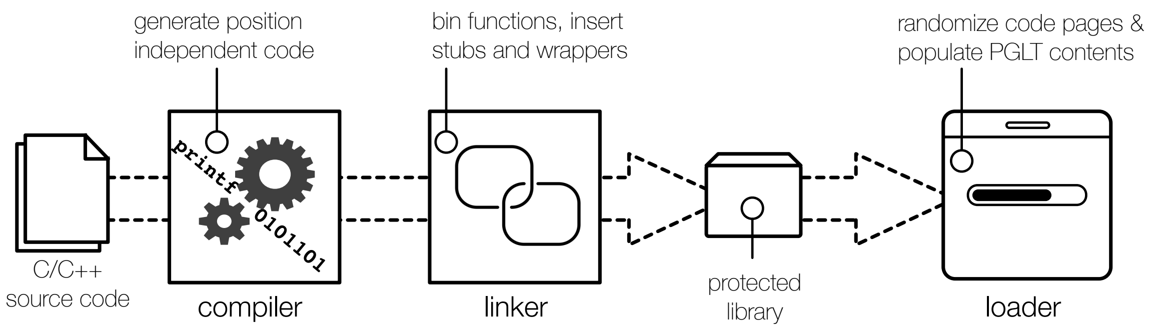
generate position independent code
bin functions, insert stubs and wrappers
randomize code pages & populate PGLT contents
printf
0101101
C/C++ source code
compiler
linker
protected library
loader

Fig. 1. Build process when using **page**rando.

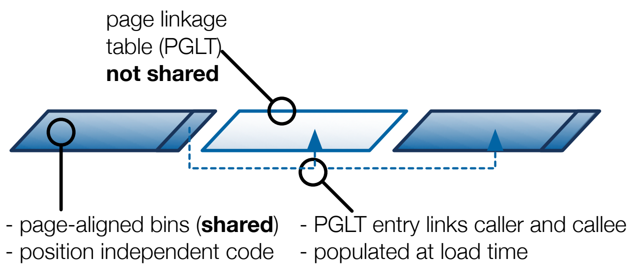
page linkage table (PGLT)
not shared
- page-aligned bins (**shared**)
- position independent code
- PGLT entry links caller and callee
- populated at load time

Fig. 2. Combining shared position independent code pages with private PGLT data pages to enable inter-page calls.

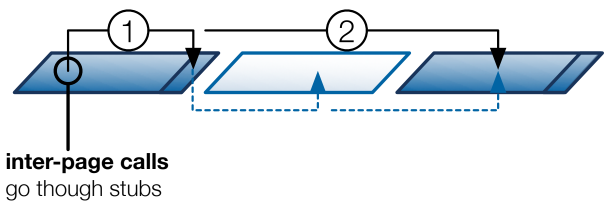

Fig. 3. Performing inter-page calls using the PGLT.

The Oxymoron approach by Backes and Nürnberger [9] shows that it is possible to randomize the layout of individual code pages while still allowing memory sharing across processes. **page**rando randomizes at the same granularity as Oxymoron, but sorts functions into bins rather than splitting functions at page boundaries, since binning avoids the need to insert extra jumps into functions that cross page boundaries. Unlike Oxymoron, **page**rando supports dynamic linking, does not rely on x86 segmentation, and is interoperable with external libraries and JIT'ed code. Like ASLR, binaries are prepared for randomization by the compiler and linker, shipped to the host system, and randomized at load time as shown in Figure 1. Since we target Android in our implementation of **page**rando, we modified the LLVM compiler and `gold` linker used in AOSP 6.0.1 to prepare libraries for randomization. We also modified the Android dynamic loader to map each compatible code page at a random virtual memory address.

Page-level randomization reduces the value of information leaks. Because ASLR does not randomize relative offsets inside a loaded module, any leaked pointer reveals the entire module's code layout. With **page**rando, the leakage from any code pointer is limited to a single page of code. Since key gadgets such as those that pivot the stack or drive the main loop in a COOP-style attack [10], [11] are rare, a single 4KiB code page is unlikely to contain all required gadgets [12], [13].

Figure 2 shows how code is placed in immutable, page-aligned bins so that each physical memory page can still be shared among all processes that load the same library. After the loader maps each bin at a random virtual addresses, it must update all references inside and outside the bins to point to the correct target addresses. It can't modify the contents of the bins themselves because this would prevent sharing the same physical memory pages between multiple processes. We preserve the immutability of bins by adding an extra level of indirection through a *page linkage table* or PGLT. The PGLT is used to locate all objects, e.g., functions, external to the bins referencing them. Each reference inside a bin accesses a fixed entry in the PGLT and the loader can freely modify PGLT entries since they are stored in data memory private to each process. The loader populates the PGLT with the correct addresses via dynamic relocations that our modified linker adds for each PGLT entry.

A call that references a callee in another bin uses the PGLT to determine the correct target address (Figure 3). The caller loads the destination address from a constant index in the PGLT and jumps to it. Note that intra-bin calls proceed as normal and do not use the PGLT.

In traditional, position-independent binaries, instructions access static data (such as variables inside `.data` or `.bss`) using PC-relative references instead of absolute references that require load-time relocation. PC-relative references to data are valid if the compiler and linker can assume a fixed linear memory layout for an ELF binary, where any location is at a fixed known offset from any other location in the module. Randomizing the locations of the bins invalidates this assumption, since each bin is loaded at a random address independently of the rest of the binary (this is also why we need to update inter-bin branches). However, the Global Offset Table (GOT) is not part of any bin, so the offset from the GOT to other data objects remains fixed and known at build-time. We rewrite all data accesses to be GOT-relative (i.e. a fixed offset from the start of the GOT) and use the PGLT to locate the GOT. We reserve the first entry in the PGLT for the GOT address, and rewrite each data access to read this entry and add it to the GOT-relative offset of the destination (known at build-time) to determine the absolute address of the data reference.

To access the PGLT during execution, randomly located code needs to be able to locate it. One way to make it easy to locate is to always place the table at a fixed location in memory. Alternatively, we could place the table at a random location and store a pointer to it at a fixed location. However, both approaches would put it within easy reach of adversaries. Instead, we place the PGLT at a random address, store this address inside the `r9` register and use that register to access the PGLT. To prevent our code from overwriting the pointer, we modified the compiler to reserve this register and never use it for other purposes in PGLT-compatible code. We initialize this register whenever code from another module calls a function inside a bin as shown in Figure 4. We assume that the contents of `r9` is preserved across the lifetime of the function and its callees (the ARM ABI specifies that `r9` is a callee-saved register, so external functions will preserve it across calls). To make sure the register is correctly initialized before use, we identify all entry points into the protected library (all code addresses that can be called from the outside) and build a table of entry wrappers, one per entry point. Each entry wrapper saves the current value of `r9`, sets `r9` to the correct value, and then transfers control to the actual entry point. This ensures compatibility with non-**page**rando binaries, since the executable and other libraries call **page**rando code as normal.

In some cases, entry wrappers need to perform some extra steps to pass parameters correctly to the callee. Some parameters are passed on the stack and the callee expects them to be on top of the stack at the call site, but saving `r9` to the stack in the entry wrappers breaks this expectation. The compiler solves this problem for us, as we emit each entry wrapper as compiler bytecode—LLVM IR in this case—and each wrapper calls the callee using a LLVM function call with a full list of the parameters. The compiler then takes care of correctly passing the parameters to the callee using the required registers and stack slots. We encountered a related but slightly more complicated corner case for variadic functions (functions with a variable number of type of arguments). In this case, the function internally uses the `va_start` compiler intrinsic to initialize a `va_list` structure containing a list of the parameters. To correctly support the intrinsic, we lift it and the structure out of the function and into the wrapper, and pass in the `va_list` structure as a new hidden parameter to the callee, instead of the variadic argument.

III. Evaluation

Our implementation of **page**rando targets 32-bit ARMv7 shared libraries for the Android Open Source Project (AOSP) version 6.0.1 (Marshmallow). We evaluate our implementation on a HiKey ARM development board (LeMaker version) with a HiSilicon Kirin 620 8-core ARM Cortex-A53 processor and

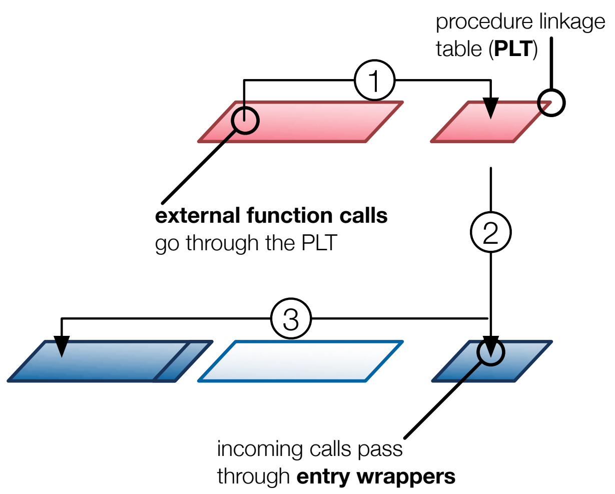
procedure linkage table (**PLT**)
1
external function calls go through the PLT
2
3
incoming calls pass through **entry wrappers**

Fig. 4. When an external library calls a function inside a bin, the call is routed through an entry wrapper which points `r9` to the PGLT before transferring control to the callee.

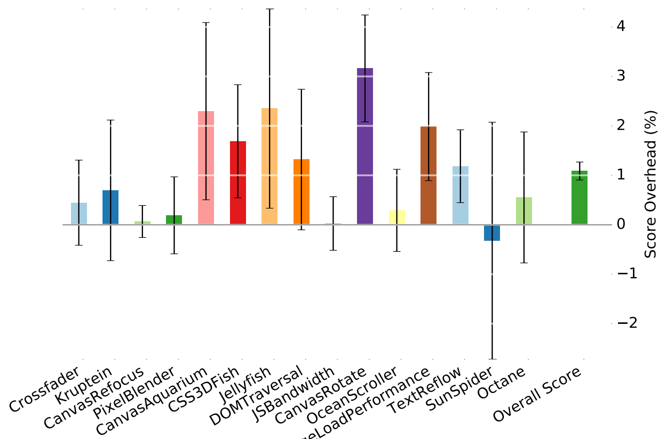

Fig. 5. **page**rando score overhead on Vellamo 3.2 browser benchmarks with Chrome. Error bars represent 95% confidence interval.

2GB of RAM. The board requires a 64-bit Android kernel, but 64-bit Android is by default a multiboot 32/64-bit system with 32-bit libraries in `/system/lib/` and 64-bit libraries in `/system/lib64/`. We built the 64-bit portion of the OS as normal, without **page**rando, and applied **page**rando protections to *all* 32-bit system libraries. We installed all benchmarks with the `armeabi-v7a` ABI, which means that the benchmark apps will only use the 32-bit **page**rando protected runtime and system libraries.

We found that dynamic frequency scaling had a significant impact on benchmark variance, so we configured the CPU to run at a constant 1.2 GHz. Running this board at a fixed clock speed required adding a heatsink and fan to avoid overheating the CPU.

A. Vellamo Benchmark Performance

We evaluate the overall performance and memory impact of **page**rando using the Vellamo Mobile Benchmark 3.2. Vellamo consists of an extensive browser benchmark suite (including

Sunspider and Octane), a CPU performance suite, and multiprocessing benchmarks. As mentioned, we installed the 32-bit version of the Vellamo app so it will only use **page**rando protected system libraries. We also installed the 32-bit version of the Chrome browser for Android (v51.0.2704.81) for use in the browser portion of Vellamo. We ran Vellamo 10 times and report the average results.

We found that the impact of **page**rando largely depends on the shared library usage of the benchmark. We believe the browser chapter of Vellamo is the most representative of real-world usage of shared libraries. The metal chapter consists of CPU-bound workloads without many library calls, and the multi chapter is similar but with a large outlier, Process-Communication, which measures inter-process communication latency in a tight loop, similar to the Bionic benchmarks we discuss below.

Overall, **page**rando lowers the total score of the Vellamo browser benchmarks by 1.09%, metal by 0.62%, and multi by 6.5%. We show the entire browser chapter results in Figure 5.

We also measured the memory overhead for Chrome while running the Vellamo benchmarks using the `dumpsys` system tool. This tool reports a metric called the Proportional Set Size (PSS) of an app, which is the app's total unshared RAM usage plus a portion of the shared RAM usage corresponding to the number of processes that RAM is shared with. We found that our current implementation of **page**rando adds 19% to the average shared library memory PSS of Chrome. We believe there is substantial room for optimization of this overhead. We describe how and where we can optimize overheads in § IV-A.

B. Bionic Micro-benchmarks Performance

We also evaluated the worst-case performance impact of **page**rando by repeatedly calling small library functions in a tight loop. The Android implementation of `libc`, Bionic, includes a small micro-benchmark which repeatedly calls 166 different `libc` functions. Since **page**rando requires instrumentation at every library entry point to set up the PGLT register and the library functions themselves are small, we naturally observe a higher percentage performance overhead for the Bionic benchmarks. Overall, we found that **page**rando incurs a $52.33 \pm 0.19\%$ geometric mean performance slowdown over all micro-benchmarks (averaged over 5 runs).

C. Security

The entropy of **page**rando is equivalent to other solutions randomizing at the page level [9]: A shared library having p bins and being loaded into an n bit address space, allows $\frac{n!}{(n-p)!}$ different layouts. The probability of guessing the full layout is inversely proportional to the number of layouts, i.e. $\frac{(n-p)!}{n!}$. The C standard library, `libc.so`, contains roughly 384KiB code which fits into 96 4KiB bins. If we conservatively assume that a 32-bit system has 2^{16} base addresses to chose from [14], the probability of guessing where code pages reside in memory is $\frac{(2^{16}-96)!}{2^{16}!} \sim 2^{-1535}$ which is far less than the corresponding probability with ASLR (2^{-16}) for all shared libraries containing more than 4KiB of code.

Like other types of fine-grained code randomization, **page**rando is also more resilient to layout disclosure than ASLR but remains theoretically vulnerable to attacks that can disclose arbitrarily many code pointers [15]. We discuss ways to increase leakage resilience of **page**rando without making it any less practical or deployable in the following section.

IV. Discussion

This section outlines a few outstanding issues that, when addressed, strengthen the security of code randomization techniques and put them on par with other exploit mitigations.

A. Optimizations

Our prototype implementation of **page**rando has ample room for optimization of both inter-bin control-flow transfers and library entry point wrappers. We implemented inter-bin control flow in the linker by adding a code snippet (stub) for each call or jump between bins. Thus, each inter-bin control flow requires an extra direct call or jump to the stub that would not be required if the compiler inserted all instrumentation for bin interwork. Enabling link-time optimization would allow the compiler to optimally insert this instrumentation.

Our current implementation of entry wrappers uses code relocations to conveniently load the absolute address of the PGLT, which disallows sharing of these wrappers between processes. Entry wrappers should instead load the address of the PGLT from a PC-relative data word, stored next to the entry wrapper code in memory. This allows the dynamic loader to only relocate a single location and share all entry wrapper code between processes. This will significantly lower the per-process memory overhead when using **page**rando.

Even further optimization is possible by coarsening the randomization granularity to a small multiple of the page size.

B. Mixed-Granularity Randomization

Different randomization granularities are ideal for different classes of code. Page-level randomization that **page**rando implements is ideal for shared libraries which are loaded in multiple processes and thus must be shared. However, executables and libraries that are only loaded in a single process do not benefit from memory sharing. Load-time randomization of the executable code at a finer granularity, e.g. at the level of individual functions [16], increases security against leakage attacks while requiring less instrumentation than **page**rando. The **self**rando function-level randomizer (which was recently integrated into the Tor Browser [17]) is an example of a practical and deployable load-time option for native code not in shared libraries [12].

C. Debugging

One of the key challenges with code randomization is that it interferes with debugging. With ASLR, debuggers simply adjust for a single base address offset. In the case of **page**rando, we need to teach the debugger that modules have a base address for each bin (ELF segment).

Mature debuggers like `gdb` already support a plethora of ABIs. Moreover, the ABI for Texas Instruments C6000 embedded processors uses a *data segment base table*, DBST, which is substantially similar to our PGLT. Therefore, `gdb` can be extended to support the **page**rando ABI with modest effort.

D. Error Reporting

Automatic reporting of crashes is critical for diagnosing and removing bugs in deployed software. The reported instruction pointer indicates the crash location and return addresses can be used to determine the calling context. With code randomization, the instruction pointer and return addresses at the time of the crash will diverge even if crashes occur at the same location in the source code. To make sure that error reports can be symbolicated (translated to source code locations) and correlated, we must normalize the code addresses in crash reports. Doing so is possible if we store the seed value used to initialize the random number generator or store a mapping that translate randomized addresses back to canonical ones.

The normalization mapping must remain hidden to adversaries. One way to keep it hidden is to unmap the memory pages storing the mapping during normal operation and only allow read access to the mapping while normalizing addresses. Another option is to securely store the randomization seed which can then be used together with the program binary to compute the normalization map lazily. Either option is likely fast enough in practice since normalization of code addresses is only necessary for error reports, back traces, and exception handling—none of which happens on the fast path.

E. Leakage Resilience

Leaking code pointers or other information correlated with the code layout is the most effective way to bypass code randomization defenses [18]. ASLR is particularly susceptible to leakage but finer-grained variants are vulnerable too—particularly when adversaries can access arbitrary memory locations [15], [19] or launch an exploit repeatedly [20], [21]. There are several ways make code randomization defenses leakage resilient. One line of work seeks to prevent leakage in the first place by preventing reads to code pages [22], [23] and by hiding code pointers that indirectly reveal code locations [7], [11], [24], [25]. An alternative line of defense is to rewrite the code continuously [26], [27] to invalidate the leaked information or use destructive reads to prevent execution of what was read [28], [29]. Runtime re-randomization strategies require accurate tracking and updating of all pointers in memory which is not possible in practice [30]. Destructive reads are interesting as well but potentially less secure than leakage resilience strategies [31]. Out of these approaches, preventing leakage resilience in the first place appears to be the most practical and efficient strategy on systems with hardware support for execute-only memory—which includes modern ARMv8 Android devices.

V. Related Work

Backes and Nürnberger showed that randomizing shared libraries below the level of memory pages substantially increases system-wide memory consumption [9]. Their solution—Oxymoron—uses an indirection table similar to our PGLT but uses the vestiges of x86 segmentation rather a dedicated register to address the indirection table. Since **page**rando targets ARM processors, we reserve a register to point to the PGLT. Unlike Oxymoron, our solution supports dynamic linking and interoperability with legacy code.

Bojinov et al. [32] describes the difficulties of making traditional ASLR compatible with the restrictions of the Android operating system. Since then, Google has added additional restrictions such as SELinux [33] security policies.

When child processes are created by forking off an existing process, the child inherits the memory layout of the parent process. In Android, apps are launched by forking the Zygote process which pre-loads shared libraries and resources. Lu et al. [34] prototyped techniques to randomize on fork. Their prototype uses dynamic binary instrumentation which interferes with Android's SELinux policies.

Sun et al. [35] also studied randomization of code in shared libraries on Android. Unlike **page**rando, their solution, Blender, does not require modification to the Android compiler or linker. Moreover, Blender randomizes processes that are forked off the Zygote process like Lu et al. However, Blender is limited to base-address randomization and is therefore no more resilient to information leakage than ASLR.

Previous work on integrity and randomization-based exploit mitigations is extensive. The surveys by Szekeres et al. [36], Skowrya et al. [37], Larsen et al. [38], and Burow et al. [39] provide comprehensive overviews of the academic literature.

VI. Conclusion

This paper demonstrates that practical and efficient code randomization solutions can be built and integrated into modern *NIX operating systems. We do so by using our tool to compile and protect all shared libraries on stock Android 6.0.

Acknowledgment

We thank the anonymous reviewers as well as Vishwath Mohan and Nick Kralevich for their feedback and insights.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-10-C-0237 and by Silent Circle LLC.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency, its Contracting Agents, the Air Force Research Laboratory, or any other agency of the U.S. Government.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information System Security*, vol. 13, 2009.

[2] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[3] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security Symposium*, 2015.

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

[5] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[6] *ARM Compiler Software Development Guide v5.04*, 2013, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471k/chr1368698593511.html.

[7] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[8] PaX Team, *Homepage of The PaX Team*, 2001, http://pax.grsecurity.net.

[9] M. Backes and S. Nürnberger, "Oxymoron - making fine-grained memory randomization practical by allowing code sharing," in *USENIX Security Symposium*, 2014.

[10] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[11] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[12] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A. Sadeghi, "Selfrando: Securing the tor browser against de-anonymization exploits," in *Privacy Enhancing Technologies Symposium (PETS)*, 2016.

[13] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size does matter in turing-complete return-oriented programming," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[14] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *ACM Conference on Computer and Communications Security (CCS)*, 2004.

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

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

[17] J. Kopstein, "Tor is teaming up with researchers to protect users from FBI hacking," http://motherboard.vice.com/read/tor-is-teaming-up-with-researchers-to-protect-users-from-fbi-hacking, 2016.

[18] F. J. Serna, "The info leak era on software exploitation," in *BlackHat USA*, 2012.

[19] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling client-side crash-resistance to overcome diversification and information hiding," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[20] J. Siebert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[21] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[22] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[23] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[24] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[25] 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 *ACM Conference on Computer and Communications Security (CCS)*, 2015.

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

[27] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[28] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[29] J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2016.

[30] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "Beyond the PDP-11: Architectural support for a memory-safe C abstract machine," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[31] K. Z. Snow, R. Rogowski, F. Monrose, J. Werner, H. Koo, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[32] H. Bojinov, D. Boneh, R. Cannings, and I. Malchev, "Address space randomization for mobile devices," in *ACM Conference on Wireless Network Security (WiSec)*, 2011.

[33] B. McCarty, *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., 2004.

[34] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make ASLR win the clone wars: Runtime re-randomization," in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[35] M. Sun, J. C. S. Lui, and Y. Zhou, "Blender: Self-randomizing addresss space layout for android apps," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2016.

[36] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[37] R. Skowyra, K. Casteel, H. Okhravi, and N. Zeldovich, "Systematic analysis of defenses against return-oriented programming," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013.

[38] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[39] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz, "Control-flow integrity: Precision, security, and performance," *Computing Research Repository (CoRR)*, vol. abs/1602.04056, 2016. [Online]. Available: http://arxiv.org/abs/1602.04056