Adoption Challenges of Code Randomization

Per Larsen
perl@uci.edu
University of California, Irvine

Michael Franz
franz@uci.edu
University of California, Irvine

ABSTRACT

Languages in the C family are distinguished by their efficiency, maturity, and their lack of guardrails compared to other mainstream language in use today. Their efficiency properties kept these languages relevant as new ones appeared. Their lack of memory safety and the resulting vulnerabilities is an ongoing challenge.

Code randomization, a moving target defense technique, is one among many competing answers to this challenge. Many techniques have been proposed and evaluated extensively in academic conferences but adoption in the field is lagging. The goal of this paper is to highlight why adoption is so hard and what can be done about it. Code randomization techniques offer much flexibility in their design and implementation. We encourage research that investigates the complex trade-offs between security and many equally important concerns that must be made for enhanced code randomization defenses to make their way into production.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

code randomization, exploits, mitigations, moving target defense

ACM Reference Format:
Per Larsen and Michael Franz. 2020. Adoption Challenges of Code Randomization. In *7th ACM Workshop on Moving Target Defense (MTD'20), November 9, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3411496.3421226

1 INTRODUCTION

Because there are very few safeguards when C/C++ code is running in production [24], defenders must eliminate as many bugs as possible during implementation and testing. Dynamic testing tools such as fuzzers [14] and sanitizers [23] play a key role here. Pre-release testing is far from perfect, however, so defenders must also react promptly to residual bugs that make it into the final software release and put users at risk. Once a vulnerability is known to the defenders, it can be patched. However, adversaries can discover vulnerabilities and use them to deliver malicious payloads before a patch is available. Thus the primary role of mitigations is to buy defenders time to develop a patch and buy users time to install it.

Mitigations must slow down adversaries without significantly interfering with normal program operation; a mitigation that violates this property is a mitigation that will not be deployed. This makes development of mitigations a matter of making the right trade-offs (more on that later). As a result, mitigations end up narrowly targeting specific exploitation techniques. This is perfectly fine since multiple mitigations are used together with the combined result of closing off entire avenues of exploitation. Data execution prevention [15], for instance, largely obsoleted code injection techniques on hosts where it was deployed. Faced with this challenge, adversaries discovered that exploitation did not inherently require code injection. This is just one example of how the deployment of a new technique prompts a counter-reaction from people with opposing interests. Figure 1 shows a brief and incomplete timeline of exploits and mitigations over the last twenty years. This is the frame within which we consider efforts to adopt enhanced code randomization.

We see in Figure 1 that stack canaries were among the first moving target defenses deployed. Address space layout defense was introduced a few years later as part of PaX - a patch for the Linux Kernel [18]. Since 2001, these two code randomization mitigations were deployed more widely and extended to privileged code such as OS kernels. Given the age of ASLR, its weaknesses are well understood and many papers propose improvements that substantially improve its security properties. Yet, these improvements have not seen wide uptake among operating system developers. Instead, practitioners adopted a different mitigation technique: control-flow integrity, CFI [1, 5]. CFI is not a moving target defense, rather it makes it harder to hijack the control flow by restricting the possible target addresses for each indirect branch.

2 CODE RANDOMIZATION AND CONTROL-FLOW INTEGRITY

What characterizes a good mitigation? As usual, that depends on who we ask. Academic reviewers value novelty and the security of a proposed mitigation; the goal is to bring new ideas to light after all. Evaluations need only span a handful of programs and it is not necessarily disqualifying if some programs are incompatible with the proposed technique. Practitioners, as expected, take a more pragmatic view which is perhaps best summarized as "done is better than perfect". Security-wise, a good mitigation "should eliminate a common vulnerability class or break a key exploitation technique or primitive used by modern exploits" [17]. However, performance, compatibility, maintainability, and integration are all important concerns when deciding which mitigation technique to deploy.

To concretize the discussion, let us look closer at the CFI, techniques that are deployed to counter return-oriented programming

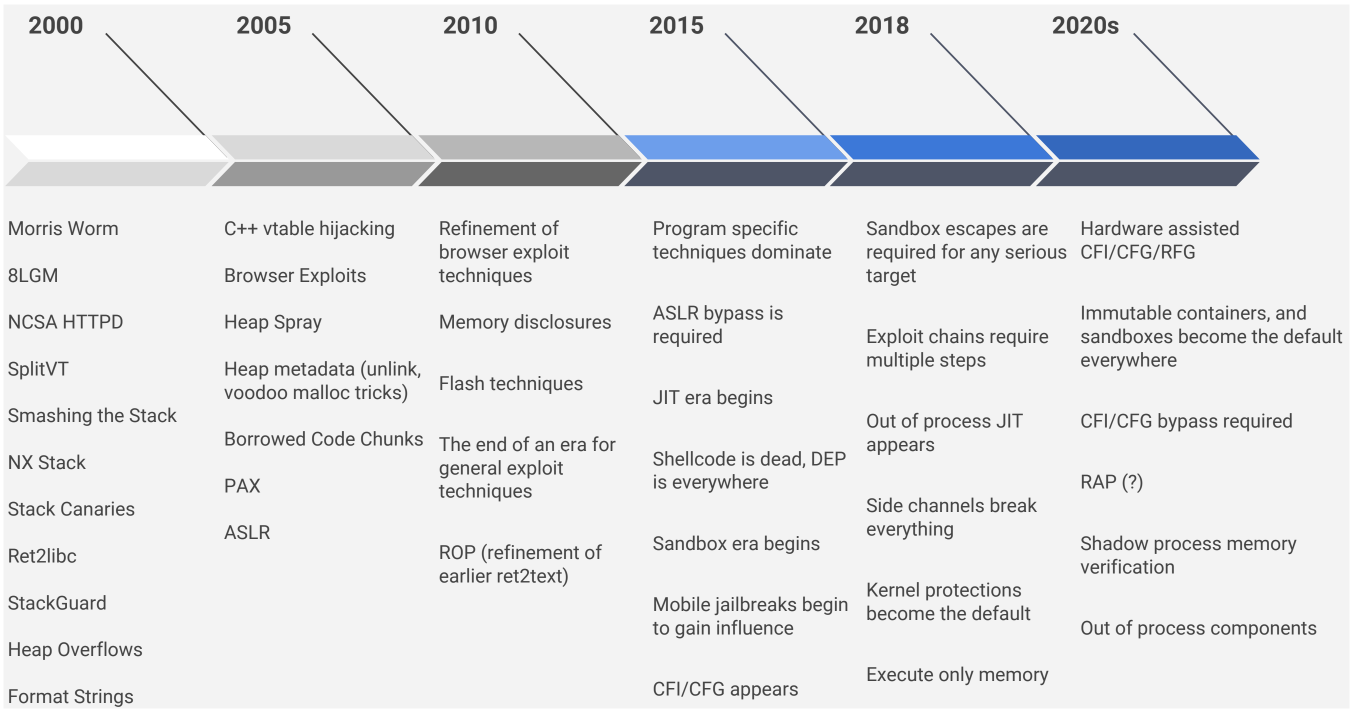
2000
2005
2010
2015
2018
2020s
Morris Worm
8LGM
NCSA HTTPD
SplitVT
Smashing the Stack
NX Stack
Stack Canaries
Ret2libc
StackGuard
Heap Overflows
Format Strings
C++ vtable hijacking
Browser Exploits
Heap Spray
Heap metadata (unlink, voodoo malloc tricks)
Borrowed Code Chunks
PAX
ASLR
Refinement of browser exploit techniques
Memory disclosures
Flash techniques
The end of an era for general exploit techniques
ROP (refinement of earlier ret2text)
Program specific techniques dominate
ASLR bypass is required
JIT era begins
Shellcode is dead, DEP is everywhere
Sandbox era begins
Mobile jailbreaks begin to gain influence
CFI/CFG appears
Sandbox escapes are required for any serious target
Exploit chains require multiple steps
Out of process JIT appears
Side channels break everything
Kernel protections become the default
Execute only memory
Hardware assisted CFI/CFG/RFG
Immutable containers, and sandboxes become the default everywhere
CFI/CFG bypass required
RAP (?)
Shadow process memory verification
Out of process components

Figure 1: An incomplete overview of exploits and mitigations from the late nineties until the present. ©Chris Rohlf (http://struct.github.io/); reproduced with permission.

and code-reuse techniques more broadly. The goal of the comparison is to see how each of these technologies address concerns that decide whether a mitigation is adopted or not. Specifically, we will look at Microsoft's Control-Flow Guard, CFG [16], and ARM Pointer Authentication Codes, PAC [19], as these CFI implementations are deployed to millions of Windows and iOS devices respectively. On the code randomization front, we will consider Address Space Layout Randomization, ASLR and Address Space Layout Permutation, ASLP. We include ASLP because it is a straightforward improvement over ASLR and the authors were involved with the design and evaluation of an ASLP prototype focused on deployability which helps guide the discussion.

Let's briefly recap the characteristics of these mitigations.

ASLR Together with data execution prevention and stack canaries, ASLR is (or should) be the baseline for all systems running potentially vulnerable code. ASLR requires that the compiler emits code that is either position independent (Linux) or can be relocated before it is executed (Windows). This let's the kernel or dynamic linker place the code section at a randomly selected offset in virtual memory. Operating systems only keep one copy of (the code in) a shared library in memory even if loaded by multiple processes. To preserve this property, ASLR requires changes to both the operating system, the loader, linker, and compiler. Moreover, memory sharing requires that the random base address is aligned to a memory page which, along with other requirements, meaningfully decreases the available entropy on 32-bit systems [22]. This is a perfect example of the trade-offs that deployed mitigations make. Hosts with 64-bit address spaces provide plenty of randomness, so exploit chains include a stage that leaks a code pointer to reveal the effects of ASLR on the target process [21].

CFI Where ASLR presents an adversary with a moving target, CFI places restrictions on the targets of indirect branches. This can thwart exploits because they tend to introduce abnormal control flows (jumps into the middle of a basic block or calls into the middle of a function) that do not occur during normal program execution.

Just like code randomization mitigations, CFI must trade security for speed, compatibility, and so on. Some CFI implementations approximate the legitimate set of branch targets via static program analysis. This approach turned out to be hard to scale and complex to implement, so the CFI implementations in use today trade some security for increased compatibility and decreased complexity. Microsoft's Control Flow Guard, for instance, simply enforces that an indirect function call targets a function that is "address-taken" meaning that it is potentially called through a function pointer. This policy is less strict than one derived from a static analysis but is simpler to apply even to applications that are written in a mix of different languages (C/C++ and assembly) and privileged code such as operating system kernels. When code is compiled, the compiler emits a function table that identifies valid indirect call targets as well as code that validates such targets before executing an indirect call. All code emitted by just-in-time compilers is treated as valid by default for compatibility but JIT compilers can provide a list of valid targets via operating system APIs and thus opt into protection with modest effort. CFG further trades security for performance by only protecting indirect calls; backward edges (returns) will be protected on processors that support hardware shadow stacks. Notice that like ASLR, CFG required changes to the operating system itself as well as the compiler.

Pointer authentication codes, PAC [19], introduced in the ARM v8.3 specification [2] protects code running on Apple's iOS devices. PAC adds new instructions that sign and authenticate pointers. Signed pointers contain a hash-based message authentication code, HMAC, which is stored in the upper bits of 64-bit pointers. The signing key is stored in a dedicated hardware register to prevent accidental leakage. Like CFG, this approach avoids the need for static analysis. This is particularly attractive since Objective-C code, which is common on iOS, uses dynamic dispatch more pervasively than C/C++ and is thus harder to statically analyze than C/C++ code. PAC delivers excellent security and compatibility with existing code at the cost of requiring changes to the instruction set architecture as well as the OS and compiler.

ASLP Address space layout permutation was first explored by Kil et al. [13] as a straightforward extension to ASLR. The idea is to shuffle the function order inside each `.text` section in addition to randomizing the starting address of the `.text` section. This means that a single leaked code pointer no longer reveals the entire layout of the module it belongs to; this makes ASLP more resilient to the techniques used to bypass ASLR. Kil et al. implemented ASLP by modifying the Linux kernel similar to ASLR and reported minimal performance overheads from doing so. It is not known whether there was an attempt to upstream ASLP into the Linux kernel or not.

The authors of this paper were involved in an effort that implemented the ASLP technique without modification of the operating system kernel or compiler. The prototype is called selfrando [8] because it makes libraries and executables randomize themselves when loaded. The security and performance properties are similar to the implementation of Kil et al.; the main difference is that binaries can be built and distributed without changes to the host operating system.

3 LEARNING FROM THE DEPLOYMENT OF CONTROL-FLOW INTEGRITY

Adding either of the aforementioned CFI solutions or ASLP to a system makes it harder to construct a code reuse exploit and none of the mitigations introduce noteworthy overheads. At the same time, all of these solutions have known weaknesses that, under the right conditions, make them bypassable. ASLP can be bypassed with an arbitrary read primitive [24]. CFG can be bypassed by adhering to the control-flow policy [6, 9, 11] – which is much more permissible than many other CFI variants. For example, CFG ensures that an indirect call targets an address-taken function but does not require that the corresponding control flow-edge exists in the source code or that the target function has the expected type. PAC is a more restrictive CFI policy because a code pointer can only be used in a branch if it was generated by the program. This leaves adversaries with two options that adhere to the control-flow policy: i) swap two legitimate pointers or ii) coerce the program into generating new code pointers to be used in a code reuse exploit.

Does this mean that PAC is better than CFG? The answer is of course no! The two mitigations simply make different trade-offs. By virtue of being a pure software mitigation, CFG could be rolled out in November of 2014 whereas PAC was introduced much later in September of 2018 and was only available on devices with the very latest hardware. CFG protection of forward branches will eventually be enforced at the hardware level and return branches will similarly be protected by a hardware shadow stack. **Lesson 1: the performance concern is so great that requiring hardware changes is acceptable.** This is nothing new, data execution prevention similarly required hardware changes albeit not to the same extent as CFI.

What does all of this have to do with moving target defenses? The fact that practitioners by and large chose CFI over improvements to code randomization raises two questions:

- Why adopt CFI ahead of code randomization?
- Where can we make the case for code randomization?

4 WHY ADOPT CFI AHEAD OF CODE RANDOMIZATION?

While we were not party to the discussions that lead to CFG and PAC being deployed, we can make educated guesses by examining 1) the trade-offs between the two and 2) the compatibility challenges faced by code randomization.

Security: CFI ensures that a particular control flow policy will be followed (modulo design or implementation errors [4, 7, 26]) irrespective of the application. Code randomization can be as effective as CFI if there is no information leakage. That is a big "if", however. There are many sources of information leakage including side channels and direct reading of memory by exploiting memory corruption. If the adversary can run code locally on the target host (by achieving co-residency in the cloud or causing the target to download malicious JavaScript), microarchitectural side channels are of particular concern. At the very least, we must assume that the code to be protected is prone to memory corruption; otherwise, there is no reason to deploy mitigations. This implies that for code randomization to match CFI, it must be deployed alongside strong countermeasures to information leakage. It is currently unclear how to pervasively counter information leakage and, if possible, what the associated cost and complexity will be.

Memory: The reason that ASLR ensures that the randomized base address is aligned to a page boundary is to make sure that only one copy of the code in shared libraries is stored in memory. This means that the page granularity is the smallest granularity of randomization that can be applied to a shared library if we want them to have a different layout in each process. Note that ASLP randomizes at the level of individual functions which are typically much smaller than a 4 kilobyte code page. We must, therefore, decide whether to randomize a shared library at page granularity in each process or randomize the library at a finer granularity and share a single randomized copy across all processes. As Nürnberger et al. [3] show, the cost of breaking memory sharing makes code randomization impractical.

Interoperability with Other Defenses: CFI is fully interoperable with a range of widely-deployed defenses. Let us consider a couple of important ones. Both iOS and Android use checksums to detect if binaries have been modified on disk; similar solutions are used to protect system files on traditional computer systems such as Windows and MacOS. When CFI and ASLR is applied at compile time, both remain transparent to checksumming mechanisms. In other words, code randomization must be applied at load or runtime to avoid changing the on-disk representation. This is a challenge because runtime policies such as SELinux and AppArmor may similarly prevent the code from being changed after it has been loaded. This means that code randomization should ideally happen exactly at one time and place: at load time before integrity defenses such as SELinux are applied. This is of course exactly how ASLR works and any code randomization technique that chooses another time is likely to be marred by compatibility issues on a range of systems. **Lesson 2: code randomization defenses are ideally integrated into the operating system.** Integration into the operating system has practical benefits too. On Windows, some libraries are pre-loaded into most process types. These need to be randomized too; when ASLR was not applied uniformly, exploits could simply target code that was left unprotected. On Linux, all libraries can be randomized at load time without customizing the operating system but the standard C library must be patched in order to work correctly after code randomization.

Startup Performance: When a modern operating system boots on low-end hardware, the time to load each library matters. CFI defenses require very little initialization if any and code can simply be memory mapped into the running process. Code randomization, on the other hand, must read all code from disk, place each function or page at a random offset, and apply relocations so it correctly references other code and data at its new location. In this aspect, code randomization is at a disadvantage to CFI. Of course, there are likely ways to reduce this startup overhead but any such solution will add complexity and reduce security so CFI ends up looking more attractive than code randomization in this aspect.

Just-in-time Compiled Code: It is fairly easy to extend code randomization to dynamically generated code. In fact, it can be done without modifying the just-in-time compiler itself [12]. Except for legacy code, better performance can be obtained by modifying the JIT compiler to randomize the code it emits. For CFI, the JIT engine needs direct modification as well. In fact, Microsoft's CFG offers special APIs that JIT engines can use to safely call statically compiled code protected by CFG.

Debugging and Telemetry This is perhaps the area where the interests of academics and practitioners diverge the most. Being able to debug an application before it is released and collect error reports when an application crashes in the field is how practitioners support modern software and keep users happy. CFI does not interfere with either of these processes but code randomization does. For ASLR, debugging is handled by simply turning it off. For finer-grained types of code randomization, debuggers and stack unwinders must be made randomization aware which yet again adds to the complexity of deploying this type of mitigation.

When the developers of mainstream operating systems decided to adopt a new mitigation, we think it is clear why they chose CFI. It has well-understood security properties in the presence of information leaks and is arguably simpler to implement in a way that does not interfere with other defenses, operating system optimizations, and development workflows.

5 WHERE CAN CODE RANDOMIZATION SHINE?

Again, we can only offer speculation and educated guesses based on our own past experience. One can argue that now that many mainstream operating systems have deployed some form of CFI, they should consider something like ASLP next. While we believe that CFI and ASLP are complementary in their security properties, the added security may not be enough to justify the added complexity. The main reason, in our opinion, is that there are many sources of information leaks that threaten to undermine code randomization. Most recently, we have seen a steady stream of microarchitectural side-channel vulnerabilities that exploit design flaws at the hardware-level. These threats are particularly worrisome on systems where untrusted code is co-resident with trusted code operating on sensitive data (browsers, mobile apps, etc.) or when code from different owners is co-resident in public cloud infrastrucure.

Embedded devices are in the complementary set of the systems we enumerated above. The embedded space is characterized by heterogeneity at the hardware and software level and a focus on low per-unit costs. Taken together, this means that widespread hardware support for CFI is likely a long way out. Absent such hardware support, ASLP is highly performance competitive with CFI variants that protect both forward (calls, jumps) and backward (returns) indirect control flows. Moreover, many processors are 32-bit only; this makes ASLP more attractive than ASLR since the former does not require a large address space to offer enough entropy to withstand brute force attacks. The threat of information leakage is also lower than on traditional computers. Embedded hardware often runs a single firmware image compiled from trusted code. Running such code on processors that perform much less speculation than power-hungry, high-performance chips reduce the threat of information leakage via microarchitectural side channels.

It may also be interesting and worthwhile to investigate to integrate the idea of moving target defenses such as code and data randomization in new operating systems. Google's Fuchsia operating system, for instance, makes some interesting design choices such as not including the file system in the kernel. One could imagine a randomization-aware file system that provides the following two properties by design: 1) it protects against unauthorized modification of binaries 2) it randomizes binaries when loaded off disk (maybe once per boot for shared libraries and kernel extensions, and per run for everything else).

Finally, high-assurance techniques known as multi-variant execution environments can benefit from code randomization [25].

6 SUMMARY AND RECOMMENDATIONS

Code randomization is currently at a disadvantage to non-moving target solutions such as CFI in the mitigation space. One major reason is that its security properties are difficult to quantify because of the memory secrecy assumption, i.e., that the memory contents of victim processes are unknown to adversaries. Not only can memory corruption undermine this assumption, so can most side channel attacks. Second, and perhaps just as important, code randomization techniques that work at a finer granularity than ASLR seem simple at first but end up requiring complex workarounds to avoid interference with existing defenses, optimizations, and tools.

Therefore, successful moving target defense mitigations need to offer clear complementary security properties on platforms that already implement CFI. Key questions for this line of work will be how hardware support and changes to the operating system can improve the aforementioned security/complexity trade-off. Alternatively, they can target deployment on embedded, emerging, and other platforms where CFI is less attractive and where the threat from information leakage can reasonably be said to be low based on an analysis of the software as well as the underlying hardware. Another way to complement CFI is to target data-only attacks instead of code-reuse attacks. This has received much less attention than code randomization. ARM's forthcoming memory tagging extensions [10, 20] is a great example of how a moving target defense mitigation can provide valuable additional security but requires changes throughout hardware, operating system, and memory allocators.

ACKNOWLEDGMENTS

To Chris Rohlf, for permitting the authors reuse his excellent overview of exploits and mitigations. To Andrei Homescu and Hamed Okhravi, for providing valuable feedback on draft versions of this paper.

This material is based upon work partially supported by the United States Office of Naval Research under contract N00014-17-1-2782.

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 Office of Naval Research or its Contracting Agents, or any other agency of the U.S. Government.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. 340–353.
[2] ARM Ltd. 2017. ARM Architecture Reference Manual ARMv8. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf.
[3] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *USENIX Security Symposium*.
[4] Andrea Biondo, Mauro Conti, and Daniele Lain. 2018. Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets. In *Symposium on Network and Distributed System Security (NDSS)*.
[5] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1, Article 16 (April 2017).
[6] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-flow Integrity. In *USENIX Security Symposium*.
[7] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *ACM Conference on Computer and Communications Security (CCS)*.
[8] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. 2016. Selfrando: Securing The Tor Browser Against De-anonymization Exploits. In *Privacy Enhancing Technologies Symposium (PETS)*.
[9] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security Symposium*.
[10] Vincenzo Frascino. 2019. ARM v8.5 Memory Tagging Extension. Linux Plumbers Conference, https://www.linuxplumbersconf.org/event/4/contributions/571/attachments/399/642/MTE_LPC.pdf.
[11] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*.
[12] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. librando: Transparent Code Randomization for Just-in-Time Compilers. In *ACM Conference on Computer and Communications Security (CCS)*.
[13] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*.
[14] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019). Early Access. https://ieeexplore.ieee.org/document/8863940.
[15] Microsoft. 2006. Data Execution Prevention (DEP). http://support.microsoft.com/kb/875352/EN-US.
[16] Microsoft Corporation 2018. Microsoft Control-Flow Guard. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.
[17] Matt Miller. 2015. https://msrc-blog.microsoft.com/2015/09/08/what-makes-a-good-microsoft-defense-bounty-submission/. Accessed September 12th, 2020.
[18] PaX Team. 2001. *Homepage of The PaX Team.* http://pax.grsecurity.net.
[19] Qualcomm Technologies, Inc. 2017. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.
[20] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. https://arxiv.org/pdf/1802.09517.pdf.
[21] Fermin J. Serna. 2012. The info leak era on software exploitation. https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2012/BH_US_12_Serna_Leak_Era_Slides.pdf. BlackHat USA.
[22] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-Space Randomization. In *ACM Conference on Computer and Communications Security (CCS)*.
[23] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *IEEE Symposium on Security and Privacy (S&P)*.
[24] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*.
[25] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and Efficient Application Monitoring and Replication.. In *USENIX Annual Technical Conference*.
[26] Tielei Wang and Hao Xu. 2019. Attacking iPhone XS Max. https://www.blackhat.com/us-19/briefings/schedule/#attacking-iphone-xs-max-14444. BlackHat USA.