Bulletin of Networking, Computing, Systems, and Software – www.bncss.org, ISSN 2186-5140
Volume 7, Number 1, pages 59–65, January 2018

High-performance and Secure Just-in-time Compiler Protection (preliminary version)

Tomoyuki Nakayama and Masanori Misono
Graduate School of Information Science and Technology
The University of Tokyo
Tokyo, Japan
{nakayama,misono}@os.ecc.u-tokyo.ac.jp

Takahiro Shinagawa
Information Technology Center
The University of Tokyo
Tokyo, Japan
shina@ecc.u-tokyo.ac.jp

***Abstract*—Modern browsers have just-in-time (JIT) compilers that compile JavaScript programs into native binary code on the fly. Since recent JIT compilers put JavaScript data into non-executable memory regions, simply putting shell code as JavaScript data does not work. To overcome this protection, recent attacks exploit constants in JavaScript programs that are complied into instructions placed in code regions, and use them as small pieces of code, called gadgets, to chain them by return-oriented programming (ROP). To counter this attack, recent browsers introduce constant blinding that encrypts JavaScript constants with a secret key and decrypts them at run time, preventing attackers from inserting arbitrary gadgets. Unfortunately, current browsers (including Firefox, Google Chrome, and Microsoft Edge) only blind constants larger than two bytes for the performance reason, allowing attackers to emit one and two byte gadgets that are known to be sufficient to mount ROP attacks. This paper proposes a high-performance and secure constant blinding technique for JIT compilers. In this technique, we decide whether to blind a constant based on the value of the constant. If a constant includes a value that can be interpreted as a control flow instruction (e.g. `ret` and `jmp`), we blind that constant even if it is two bytes or less. Otherwise, we do not blind the constant because it cannot be used as a gadget. This technique effectively reduces the overhead of constant blinding by reducing the number of constants that must be blinded, while improving the security by eliminating the possibility that even small constants are exploited as gadgets. We implemented this technique in ChakraCore, the JIT Engine of Microsoft Edge, running on x64 systems and measured the performance of the JIT engine. Experimental results confirmed that our technique improved performance by maximum of 2.85% compared with blinding all constants.**

***Index Terms*—Code-reuse attacks, JIT compiler, return-oriented programming, constant blinding**

I. Introduction

Web browsers are found on PCs, tablets, smartphones, smart TVs, gaming consoles, and so forth. Almost everybody surfs the Internet every day. Thanks to the prevalence of the Internet, web browser developers are struggling to pursue the performance or processing speed. Modern browsers such as Google Chrome, Microsoft Edge, Mozilla Firefox, and Apple Safari have their own just-in-time (JIT) compliers. JavaScript programs are compiled to intermediate representations in advance. Then they are re-compiled to native code at runtime by JIT compilers. This mechanism contributes to high speed code execution.

These familiar browsers, however, are often targeted by software exploitations. For example, attackers declare numerous constants in JavaScript programs. Then, after JIT compilation, the attacker can use JITed code as shellcode, if there exists some vulnerabilities in JIT compilers about manipulating the instruction pointer. Constant blinding is one of the defenses against such attacks. It blinds constants in JavaScript programs by XORing with random value.

Besides constant blinding, there are many research on protecting JITed code layouts. INSeRT [41] and librando [26] randomize the JITed code by randomly inserting either illegal instructions (INSeRT) or NOP instructions (librando) into the code. Libmask [27] transforms constants into global variables and marks the memory pages for these global variables as read only.

Current constant blinding, however, targets all constants larger than two bytes. Further, due to performance impacts, constants less than two bytes are ignored. Still, recent study showed that you can succeed in exploiting by using only one-byte and two-byte constants.

In this paper, we propose secure and high-performance constant blinding. It only blinds constants that contain values such that they are translated into control flow changing instructions (`ret`, for example) by JIT compilation.

We implemented the proposal in ChakraCore, the open-source software of Microsoft Edge's JIT compiler Chakra, and ran several benchmarks. The results showed that the performance was improved by the maximum of 2.85%and indicated the effectiveness of our proposal.

II. Background

Memory disclosure attacks have been main attacks against computer systems. They have the ability to execute arbitrary code on remote target systems after hijacking the control flow. Memory disclosures which leak code pointers enable attackers to bypass defense schemes like ASLR and exploit binaries using ROP [37], JOP [15], and so on [7] [9] [10] [15]. These attacks are called code-reuse attacks (CRAs). CRAs divert programs' control flow to gadgets located at predetermined memory address and chain together to construct malicious payloads [15] [21] [24] [14] [37].

To thwart these attacks, many kinds of solution have been proposed so far. They are classified into two methods: control flow integrity (CFI) and fine-grained code randomization. The former prevents attackers from redirecting the program execution flow [1] [34] [42] [43]. CFI, however, have been targeted and exploited by many kinds of attacks [**?**] [24] [21] [14]. The latter diversifies the code and therefore makes attackers thwart the reuse of instruction sequences. There are some levels of diversification granularities; function [6] [28], basic block [22] [40], and instruction [25] [33]. Unfortunately, in spite of much efforts above, a lot of studies, say, direct memory disclosure attacks [38] [39], indirect leakage of address from the stack and heap [20], and side-channel attacks [5] [11] [32] [36] have succeeded in bypassing code randomization defenses.

A. JIT Exploitations

(Direct) JIT-ROP [38] leverages scripting environment such as JavaScript (or ActionScript) and dynamically searches for gadgets in code areas via memory disclosure vulnerabilities. Attackers use such vulnerabilities to follow code pointers and collect as many code pages as possible. Attackers finally search for desired gadgets and API function calls in these pages and defeat fine-grained diversifications. Execute-only memory is used as leakage-resilient defense. A lot of research has proposed to use a combination of execute-only memory and randomized memory segments to defend randomized code layouts from direct leakage, say, JIT-ROP attacks [29] [12] [13] [18] [19] [20] [23].

Oxymoron [4] is the first protection against JIT-ROP. It uses x86 memory segmentation to hide references between code pages and impedes the recursive gadget harvesting. Both XnR [3] and HideM [23]prevent code pages from being read by emulating the execute-only memory. XnR marks code pages with "Not Present"; whenever an instruction fetch or data access is attempted on a code page, it brings a page fault. Then the OS verifies the source of the page fault and marks the page as present, readable and executable, or terminates execution. HideM uses the split-TLB to direct instruction fetches and data reads to different physical memory locations in order to transparently prevent code from being read by memory dereferencing operations. Thus HideM allows instruction fetches of code and prevents data accesses.

Nevertheless, some research demonstrated that Oxymoron, XnR, and HideM are bypassed by indirect JIT-ROP [20] [16]. Indirect JIT-ROP simply harvests code pointers from readable data pages and constructs ROP payloads. Readactor [18] is the first system that defend both types of JIT-ROP attacks. Readactor implements code pointer hiding [30], fine-grained randomization, and execute-only memory with a thin hypervisor. Although execute-only memory and fine-grained randomization prevent only direct JIT-ROP, code pointer hiding mitigates indirect JIT-ROP by hiding code pointer destinations and replacing code pointers in readable memory with trampoline layers in execute-only memory.

JIT Spraying [8] [35] (and subspecies of that [2]) is different from aforementioned attacks. Attackers prepare JavaScript programs containing numerous constant values which can be erroneously executed as attacker-controlled instruction sequences by JIT compilation. The attackers use some vulnerabilities such as manipulating the EIP to redirect code flow to the native code. This fact means that the attacker can execute hidden shellcode.

III. Constant Blinding

To tackle with JIT exploitations, Chakra, the JIT engine of Microsoft Edge, deploys defense mechanism called constant blinding. It masks constants in JavaScript programs by XORing with randomly generated value during JIT compilation. For example, considering that RND_KEY = 0x2511663F, following instruction

```
mov eax, 0x3C909090
```

will be transformed into two instruction sequences by constant blinding. Although constant blinding brings additional overhead (increases instructions), It makes JIT engine safe because the attacker can no longer use the constants he prepared.

```
mov eax, 0x1981F6AF
xor eax, 0x2511663F
```

Existing constant blinding, however, has two problems. First, constants under two bytes are ignored, because of performance overheads. This design is unfavorable because certain research have proposed to exploit JIT compilers using only one or two-byte constants [2]. Second, constant blinding is applied to whatever constants larger than two bytes. This problem causes JIT engines high performance overhead.

IV. Design and Implementation

Therefore we aim to propose secure and high-performance constant blinding. We concentrate on values which change the control flow of the program. Our proposed constant blinding only blinds constants which contain those values, regardless of their sizes.

For example, "`RET`" transfers program control flow to a return address located on the top of the stack. Actually, return-oriented programming (ROP), one of the famous exploit techniques [37], uses instruction sequences which end with RET instruction (called gadgets).

We decided the constants to be targeted in our proposal. They are described in TABLE IV [17].

V. Experiments and Results

We use the version 2.0.0.0 of ChakraCore with Clang 5.0.0 and Cmake 3.9.0. As for comparison, we use four different source code: no-modified ("Default"), constant blinding to all constants ("All-Blind"), disable constant blinding ("No-Blind"), and the proposal ("Propose"). For each, we build

Instructions	Opcode
RET	C3, CB, C2, CA
JMP	E8, FF, 9A
CALL	EB, E9, EA
SYSCALL	0F 05
INT n, INTO, INT 3	CD, CE, CC

TABLE I
THE LIST OF SPECIFIC CONSTANTS WHICH OUR PROPOSAL TARGETS

Benchmark	Propose/No-Blind	Propose/Default	Propose/All-Blind
JetStream	99.21%	101.24%	102.85%
Octane	99.61%	100.70%	100.60%
Kraken	100.15%	98.88%	99.04%
SunSpider	99.07%	100.13%	98.37%

TABLE II
COMPARISON OF RELATIVE VALUES FOR EACH FOUR BENCHMARKS

ChakraCore and run four benchmarks: JetStream, Octane, Kraken, and SunSpider, all of them are JavaScript benchmarks for web browsers.

The evaluation was performed on a PC with Intel Core i7-3630QM processor, 16.0GB RAM running Ubuntu 16.04 (x64). Also we disabled Intel TurboBoost Technology, Intel HyperThreading Technology, and Intel SpeedStep Technology.

Fig.1, Fig.2, Fig.5, and Fig.6 represents the results of running JetStream, Octane, Kraken, and SunSpider for each four kinds of source code. According to them, for almost all programs (x axis), the values of ”Propose” is between ”No-Blind”, the best performance case, and ”All-Blind”, the worst performance case.

Fig.3, Fig.4, Fig.7, and Fig.8 represents the improvements of ”Propose” over ”Default”. The key point of this comparison is that both of them are under the same security level. According to them, the benchmark results indicate that ”Propose” is superior to ”All-Blind”. Considering qualitatively, our proposal has benefit of better performance because the target constants to be constant-blinded is less. However, we cannot observe this benefit significantly than we expected. This is because the influence of CPU is stronger than that of our proposal. So we are now conducting on evaluations for ARM CPU, most of IoT devices or moblile devices run under ARM CPUs and therefore this try will be meaningful.

TABLE.II describes performance improvements of ”Propose” among four benchmarks. Be careful that both JetStream and Octane measure program processing speed per unit of time, so higher value is better. On the contrary, both Kraken and SunSpider measure execution time, so lower value is better. Based on these benchmark characteristics, you can find that all values go better in the column of ”Propose/All-Blind”. These results indicate the effectiveness of our proposal.

Fig. 1. The Results of running JetStream benchmark

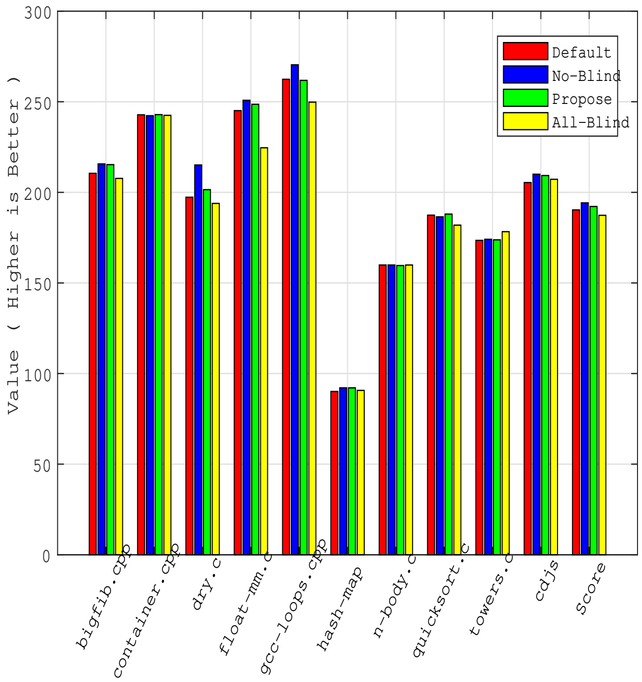


Fig. 2. The Results of running Octane benchmark

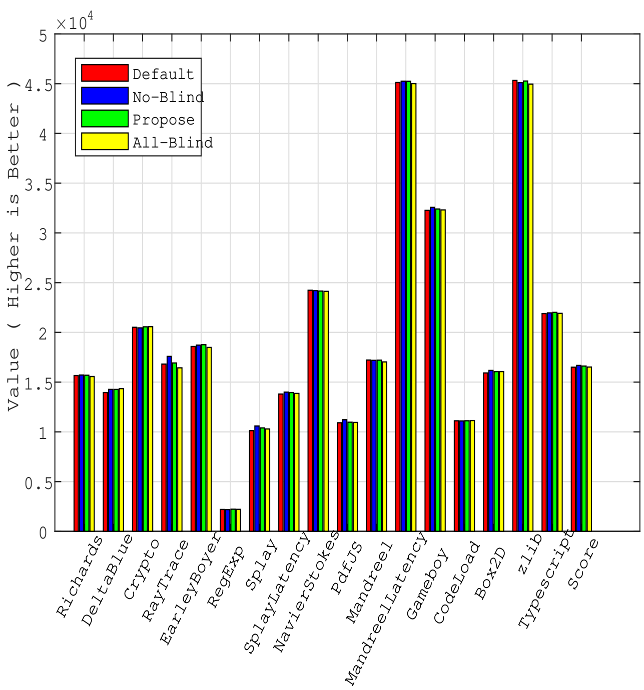


Fig. 3. The improvements of "Propose" over "All-Blind" in JetStream benchmark

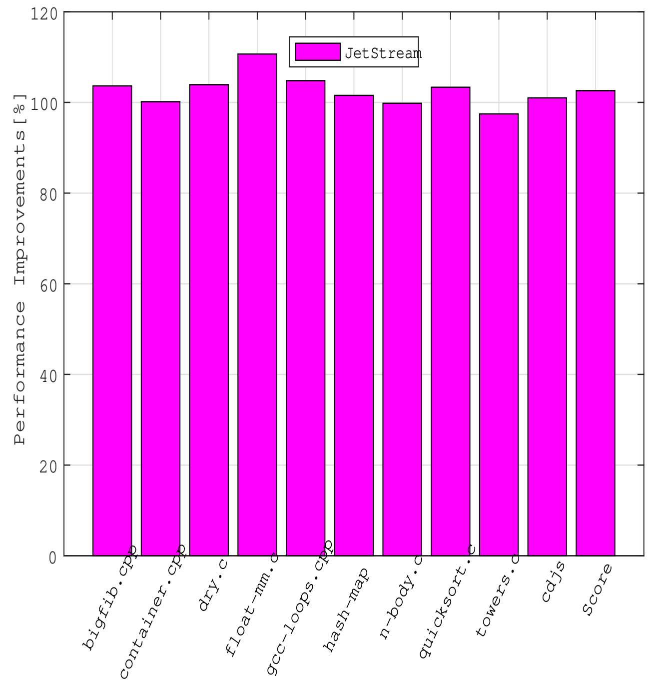


Fig. 4. The improvements of "Propose" over "All-Blind" in Octane benchmark

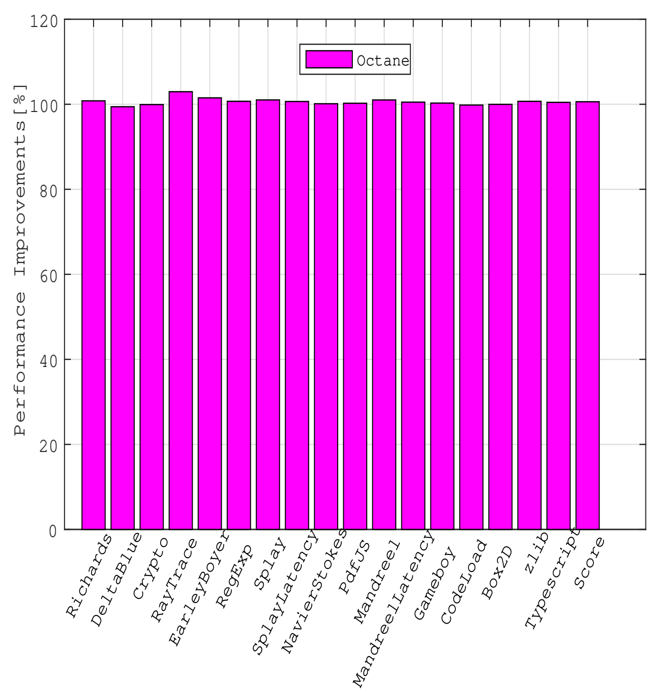


Fig. 5. The Results of running Kraken benchmark

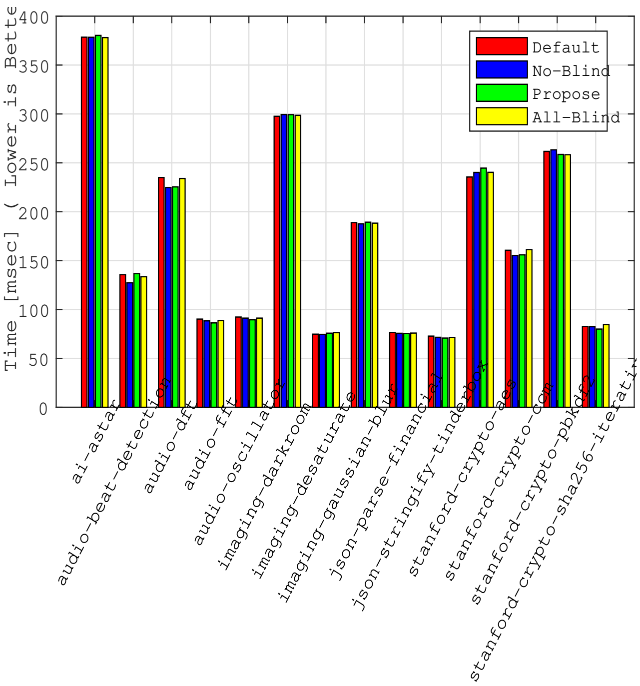


Fig. 6. The Results of running SunSpider benchmark

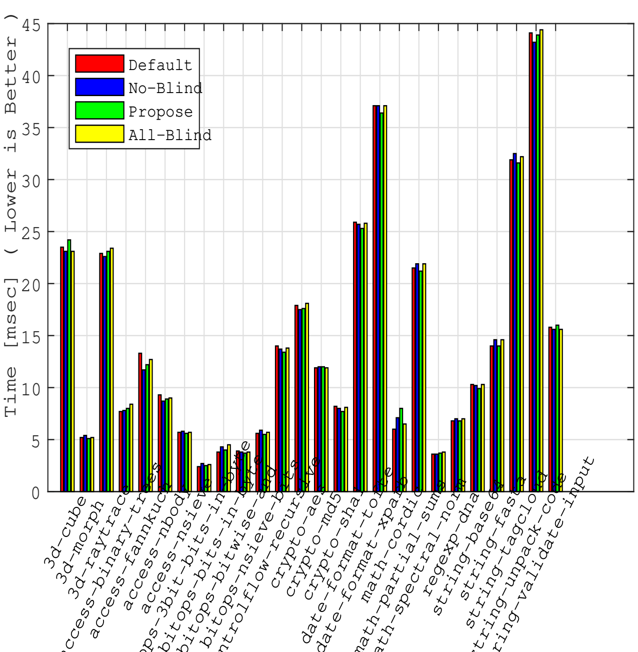


Fig. 7. The improvements of "Propose" over "All-Blind" in Kraken benchmark

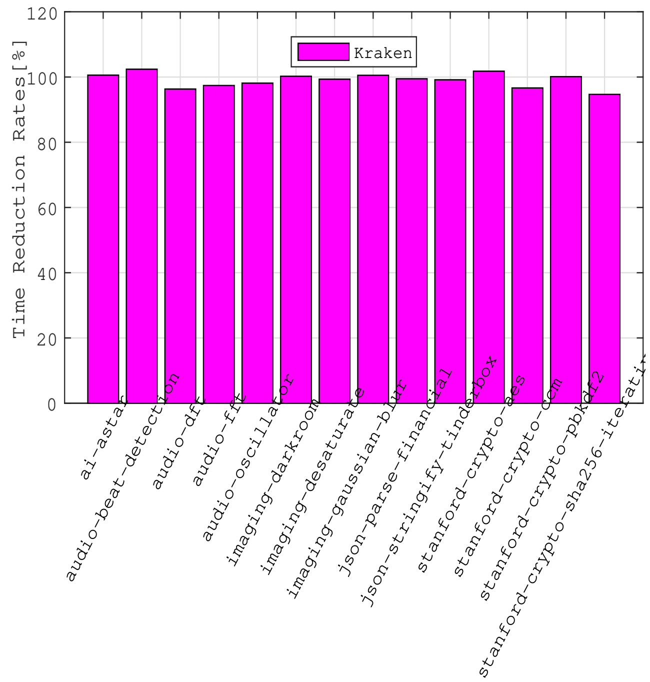
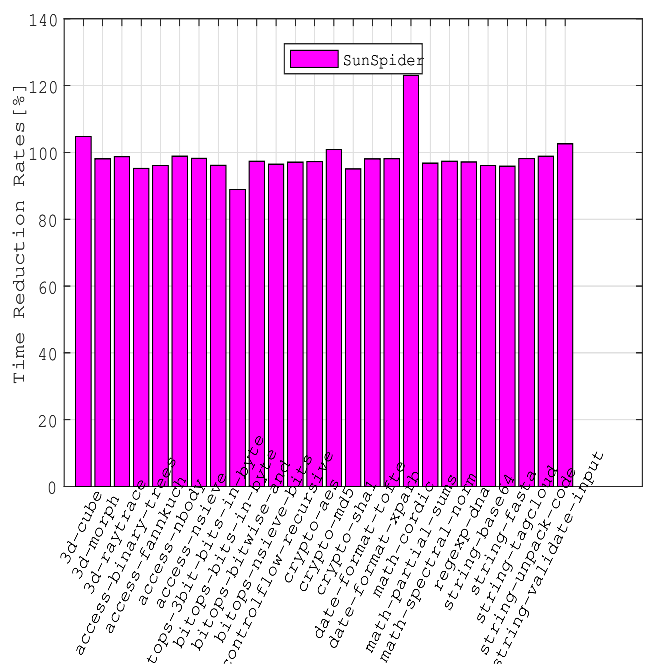


Fig. 8. The improvements of "Propose" over "All-Blind" in SunSpider benchmark



VI. Conclusions and Future Works

As the complete eradication of memory disclosure vulnerabilities remains a challenging task, defenses against their exploits is necessary. In particular, web browsers are the most targeted applications due to their popularity and property of marking JITed code with RWX. To defend against such attacks, modern browsers deploy a mitigating method called constant blinding. It prevents JIT Spraying and JIT-ROP, and related attacks by masking integer constants by XORing with a random value to generate the obfuscated constant. However, constant blinding is applied only constants larger than two bytes and therefore bypassed by using only one-byte or two-byte constants.

In this paper, we introduced a modified constant blinding scheme. It utilizes constant blinding to only constants which contain some control flow changing words, regardless of their sizes. Then we demonstrated the performance evaluation on ChakraCore. The results showed that our proposal performs 2.85% performance improvements at most.

You still have rooms to further improve secure constant blinding. It is known that constant blinding is incomplete in that constants in some JavaScript writing manners survive constant blinding [31].

We will work on evaluating our proposal from various points of view to prove its effectiveness objectively in real environment. Also, as there are a lot of cyber attacks targeting IoT devices, we will introduce our proposal to ARM architecture and survey the difference of benchmark results with those of Intel CPU.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[2] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *Proceedings of the 13rd Conference on Network and Distributed System Security Symposium (NDSS)*, 2015.

[3] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik 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 (CCS)*, pages 1342–1353. ACM, 2014.

[4] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, volume 14. USENIX Association, 2014.

[5] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R Gross. Cain: Silently Breaking ASLR in the Cloud. In *Proceedings of the 9th USENIX Conference on Offensive Technologies(WOOT)*, 2015.

[6] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security)*, pages 271–286. USENIX Association, 2005.

[7] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, and Dan Boneh. Hacking Blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, pages 227–242. IEEE, 2014.

[8] Dionysus Blazakis. Interpreter Exploitation. In *Proceedings of the 4th USENIX Conference on Offensive Technologies(WOOT)*. USENIX Association, 2010.

[9] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 30–40. ACM, 2011.

[10] Erik Bosman and Herbert Bos. Framing Signals-A Return to Portable Shellcode. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, pages 243–258. IEEE, 2014.

[11] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, pages 987–1004. IEEE, 2016.

[12] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the 14th Conference on Network and Distributed System Security Symposium (NDSS)*, 2016.

[13] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor. Exoshim: Preventing Memory Disclosure Using Execute-Only Kernel Code. In *Proceedings of the 11th International Conference on Cyber Warfare and Security (ICCWS)*, pages 56–66, 2016.

[14] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, pages 385–399. USENIX Association, 2014.

[15] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 559–572. ACM, 2010.

[16] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 952–963. ACM, 2015.

[17] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual. *Volume 3B: System programming Guide*, 2011.

[18] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP)*, pages 763–780. IEEE, 2015.

[19] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael 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 (CCS)*, pages 243–255. ACM, 2015.

[20] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 13rd Conference on Network and Distributed System Security Symposium (NDSS)*, 2015.

[21] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*. USENIX Association, 2014.

[22] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge Me If You Can: Secure and Efficient Ad-hoc Instruction-Level Randomization for x86 and ARM. In *Proceedings of the 8th ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 299–310. ACM, 2013.

[23] Jason Gionta, William Enck, and Peng Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 325–336. ACM, 2015.

[24] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP)*, pages 575–589. IEEE, 2014.

[25] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*, pages 571–585. IEEE, 2012.

[26] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. librando: Transparent Code Randomization for Just-In-Time Compilers. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pages 993–1004. ACM, 2013.

[27] Abhinav Jangda, Mohit Mishra, and Benoit Baudry. libmask: Protecting Browser JIT Engines from the Devil in the Constants. In *Proceedings of the 14th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2016.
[28] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 339–348. IEEE, 2006.
[29] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.
[30] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 280–291. ACM, 2015.
[31] G. Maisuradze, M. Backes, and C. Rossow. Dachshund: Digging for and Securing Against (Non-) Blinded Constants in JIT Code. In *Proceedings of the 15th Conference on Network and Distributed System Security Symposium (NDSS)*, 2017.
[32] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking Holes in Information Hiding. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*, pages 121–138. USENIX Association, 2016.
[33] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*, pages 601–615. IEEE, 2012.
[34] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, pages 447–462. USENIX Association, 2013.
[35] Chris Rohlf and Yan Ivnitskiy. Attacking Clientside JIT Compilers. *Black Hat USA*, 2011.
[36] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, pages 54–65. ACM, 2014.
[37] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pages 552–561. ACM, 2007.
[38] 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 *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)*, pages 574–588. IEEE, 2013.
[39] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the 2nd European Workshop on System Security*, pages 1–8. ACM, 2009.
[40] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, pages 157–168. ACM, 2012.
[41] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. INSeRT: Protect Dynamic Code Generation Against Spraying. In *Proceedings of the 1st International Conference on Information Science and Technology (ICIST)*, pages 323–328. IEEE, 2011.
[42] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)*, pages 559–573. IEEE, 2013.
[43] Mingwei Zhang and R Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22th USENIX Security Symposium (USENIX Security)*, pages 337–352. USENIX Association, 2013.