

# Attacker-defined Abstractions

Programming Benign System Functionality  
For Subversive Purposes



Erik Bosman

VU Amsterdam, 2024

---

Attacker-defined Abstractions  
Programming Benign System  
Functionality For Subversive Purposes

---

Ph.D. Thesis

Erik Bosman

*VU University Amsterdam, 2024*



Copyright © 2024 by Erik Bosman.

ISBN 978-94-6473-612-0

DOI <http://doi.org/10.5463/thesis.810>

Cover design by Erik Bosman.  
Printed by Ipskamp Printing.

VRIJE UNIVERSITEIT

**ATTACKER-DEFINED ABSTRACTIONS  
PROGRAMMING BENIGN SYSTEM FUNCTIONALITY FOR  
SUBVERSIVE PURPOSES**

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor of Philosophy aan  
de Vrije Universiteit Amsterdam,  
op gezag van de rector magnificus  
prof.dr. J.J.G. Geurts,  
in het openbaar te verdedigen  
ten overstaan van de promotiecommissie  
van de Faculteit der Bètawetenschappen  
op dinsdag 19 November om 15.45 uur  
in een bijeenkomst van de universiteit,  
De Boelelaan 1105

door

Erik Johannes Cornelius Bosman

geboren te Amsterdam

promotor: prof.dr.ir. H.J. Bos

copromotor: dr. C. Giuffrida

promotiecomissie: prof.dr. S. Bratus  
dr.ir. A. Continella  
prof.dr. M. van Dijk  
dr. M. Lindorfer  
dr. G. Portokalidis

## Acknowledgments

In order to defend against security breaches, it is important to identify all the ways computer systems can be compromised. Looking back, my PhD has been in service of finding new ways to subvert these systems, in order to get a better picture of the overall problem. But it didn't start that way.

When I started my PhD, with Herbert Bos as my supervisor, I thought I would be building systems to protect against attacks. Just as I had done with my master's thesis, also under Herbert's supervision. But the more you learn about the systems you try to protect, the greater the chance that you find new problems.

When told him about a fun solution to a Capture the Flag contest, using what turned into Sigreturn Oriented Programming, I did not think this could be useful as a publication until he convinced me otherwise.

I want to thank Herbert for his support, and for encouraging me to explore my own ideas, and for recognizing when some of those ideas could have merit. Without this, I might have never considered the other attacks that ended up defining my thesis.

I also want to thank Herbert for creating a nice environment to work in, with nice colleagues. Since much of doing a PhD involves becoming very specialized in a specific topic, it can at times feel pretty lonely. But I think you have created an environment with VUsec with lots of nice people with a systems background to discuss your work with on a technical level. I don't think I could have finished my PhD an environment where this wasn't the case, or where I wasn't free to pursue my research interests.

Perhaps the most dreaded fear associated with doing a PhD is the feeling your work has been scooped. In which case there's no-one better to talk to than Cristiano Giuffrida. It seems every conversation with him will turn into a brainstorm, leaving you with a feeling all is not lost. I also want to thank Cristiano for his contributions to in particular Dedup est Machina, making it overall a much stronger paper.

I also want to thank Kaveh Razavi, without whom Dedup est Machina would definitely not have been submitted as quickly as it was. Without him I would probably have given up on Security and Privacy deadline, but with your drive we made it work. That we made it at all was close to a miracle. We had to rethink the whole paper shortly before the submission deadline, when it turned out that one of our planned attacks didn't work due to an implementation detail in Windows.

Similarly, I want to thank Ben Gras, who along with Kaveh quickly made the aforementioned scrapped idea from Dedup est Machina work in a Linux virtualization environment. In the end it turned into something so much better with Flip Feng-shui, turning one paper idea into two.

I like to thank Bart Preneel for contributing some much needed cryptography expertise to the Flip Feng-shui paper.

I owe an enormous amount of gratitude to Asia Slowinska, who patiently helped me overcome my writer's block and finish my thesis, after it being stalled for years. Thanks to you I was able to write the introduction I intended to write. I would not have been able to finish my thesis without your help.

I am grateful to the members of my promotion committee: Sergey Bratus, Andrea Continella, Marten van Dijk, Martina Lindorfer, and Georgios Portokalidis for taking the effort of reading my thesis, and for making valuable suggestions for improvements. I am delighted that Sergey could be part of my committee since his concept of a *Weird Machine* has guided much of my thinking on the subject.

I will fondly remember the many conversations, and reading group discussions I've had with many of my colleagues, among others: Remco, Andrei B., Angelos, Xi, Istvan, Victor v.V, Enes, Dennis, Erik vd K, Lucian, Pietro, Alyssa, Sebastian, Andrei T, Stephan, Koustubha, Hany, Manolis, Koen, Emanuele, Radhesh, Taddeus, Andrea, Victor D, Elia, Floris, Brian, Marius, Sanjay, Elias, Enrico, Manuel, Math  , Bala, Klaus, Raphael, Saideh, Johannes, Alvise, Jakob, Dave, Robin, and Sander.

I want to thank my friends, especially Amran, Heleen, Jan, and Jildou, who helped keep me sane during difficult times, both before and during covid lockdown, and Marten, Wilmer, Jelmer, Sjoerd and Wilco, who I've known for a long time, and who are probably to blame for my long obsession with computers.

And last, I want to thank my parents who have always supported me, and have always believed in me way more than I believed in myself.

Erik Bosman

Amsterdam, The Netherlands, October 2024

## Contents

<b>Acknowledgments</b>	<b>v</b>
<b>Contents</b>	<b>vii</b>
<b>List of Figures</b>	<b>xi</b>
<b>List of Tables</b>	<b>xiii</b>
<b>Publications</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Framing Signals—A Return to Portable Shellcode</b>	<b>9</b>
2.1 Introduction . . . . .	11
2.2 Related work . . . . .	13
2.3 On the construction of weird machines (or: why exploitation is getting harder) . . . . .	15
2.4 Signal delivery on UNIX systems . . . . .	18
2.4.1 Delivering the signal . . . . .	18
2.4.2 Sigreturn and Data Execution Prevention . . . . .	18
2.5 Sigreturn Oriented Programming . . . . .	19
2.5.1 Signal delivery on Linux . . . . .	19
2.5.2 The sigreturn system call on Linux . . . . .	20
2.5.3 Sigreturn on other UNIX flavors . . . . .	20
2.5.4 Abusing sigreturn . . . . .	21
2.6 SROP for exploitation . . . . .	22
2.6.1 A simple <code>execve()</code> SROP exploit . . . . .	22



2.6.2	Finding a sigreturn gadget . . . . .	22
2.6.3	Executing system calls . . . . .	23
2.6.4	SROP exploitation on Linux x86-64 using a system call chain . . . . .	24
2.6.5	Bootstrapping to arbitrary code execution . . . . .	26
2.6.6	Exploiting the Asterisk web server . . . . .	28
2.7	SROP as a backdoor . . . . .	29
2.8	SROP to circumvent code signing . . . . .	31
2.8.1	Sigreturn on iOS . . . . .	32
2.8.2	A system call proxy . . . . .	33
2.9	The Linux system call interface makes SROP Turing complete . . . . .	33
2.10	Mitigation . . . . .	34
2.11	Conclusion . . . . .	36
<b>3</b>	<b>Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector</b> . . . . .	<b>37</b>
3.1	Introduction . . . . .	38
3.2	Background . . . . .	39
3.2.1	Memory Deduplication . . . . .	39
3.2.2	The Memory Deduplication Side Channel . . . . .	40
3.3	Attack Primitives . . . . .	41
3.4	Microsoft Edge Internals . . . . .	45
3.4.1	Object Allocation . . . . .	45
3.4.2	Native Array Representation . . . . .	46
3.4.3	Programming Memory Deduplication from JavaScript . . . . .	47
3.4.4	Dealing with Noise . . . . .	47
3.5	Implementation . . . . .	48
3.5.1	Testbed . . . . .	48
3.5.2	Leaking Code Pointers in Edge . . . . .	48
3.5.3	Leaking Heap Pointers in Edge . . . . .	50
3.5.4	Discussion . . . . .	54
3.6	Rowhammering Microsoft Edge . . . . .	55
3.6.1	Rowhammer Variations . . . . .	55
3.6.2	Finding a Cache Eviction Set on Windows . . . . .	56
3.6.3	Finding Bit Flips . . . . .	56
3.6.4	Exploiting Bit Flips . . . . .	57
3.6.5	Creating a Counterfeit JavaScript Object . . . . .	57
3.6.6	Dealing with Garbage Collection . . . . .	58
3.6.7	Putting the Pieces Together . . . . .	58
3.6.8	Discussion . . . . .	59
3.7	System-wide Exploitation . . . . .	60
3.7.1	Memory Allocation Behavior . . . . .	61
3.7.2	Controlling the Heap . . . . .	61
3.7.3	Server Fingerprinting . . . . .	63

3.7.4	Password Disclosure . . . . .	63
3.7.5	Heap Disclosure . . . . .	65
3.7.6	Dealing with Noise . . . . .	66
3.7.7	Time and Memory Requirements . . . . .	67
3.8	Mitigation . . . . .	67
3.9	Related work . . . . .	68
3.9.1	Side Channels over Shared Caches . . . . .	68
3.9.2	Side Channels over Deduplication . . . . .	69
3.9.3	Rowhammer Timeline . . . . .	70
3.10	Conclusions . . . . .	70
<b>4</b>	<b>Flip Feng Shui</b>	<b>73</b>
4.1	Introduction . . . . .	74
4.2	Flip Feng Shui . . . . .	75
4.2.1	Memory Templating . . . . .	76
4.2.2	Memory Massaging . . . . .	76
4.2.3	Exploitation . . . . .	78
4.3	Cryptanalysis of RSA with Bit Flips . . . . .	79
4.4	Implementation . . . . .	81
4.4.1	Kernel Same-page Merging . . . . .	81
4.4.2	Rowhammer inside KVM . . . . .	83
4.4.3	Memory Massaging with KSM . . . . .	85
4.4.4	Attacking Weakened RSA . . . . .	86
4.4.5	End-to-end Attacks . . . . .	87
4.5	Evaluation . . . . .	89
4.5.1	dFFS on the Rowhammer Testbed . . . . .	90
4.5.2	The SSH Public Key Attack . . . . .	90
4.5.3	The Ubuntu/Debian Update Attack . . . . .	91
4.5.4	RSA Modulus Factorization . . . . .	92
4.6	Mitigations . . . . .	93
4.6.1	Defending against dFFS . . . . .	94
4.6.2	Mitigating FFS at the Software Layer . . . . .	97
4.7	Related Work . . . . .	98
4.7.1	Rowhammer Exploitation . . . . .	98
4.7.2	Memory Massaging . . . . .	99
4.7.3	Breaking Weakened Cryptosystems . . . . .	99
4.8	Conclusions . . . . .	100
4.9	Cryptanalysis of Diffie-Hellman with Bit Flips . . . . .	101
<b>5</b>	<b>Conclusion</b>	<b>103</b>
	<b>Contributions to Papers</b>	<b>109</b>

<b>References</b>	<b>111</b>
<b>Summary</b>	<b>123</b>

## List of Figures

2.1	Excerpt from <code>s5/signal.s</code> , UNIX V6 interrupt routine . . . . .	19
2.2	The signal frame as it looks like in Linux 86-64 . . . . .	21
2.3	Vsyscall is mapped in a process' address space . . . . .	25
2.4	The vsyscall page contains the <code>syscall &amp; ret</code> gadget multiple times . .	26
2.5	Steps involved in the Linux x86-64 SROP exploit . . . . .	27
2.6	Our example backdoor automaton . . . . .	31
2.7	A syscall proxy controllable over a pipe/file/network socket . . . . .	32
2.8	Our Turing-complete interpreter . . . . .	35
3.1	The <i>alignment probing</i> primitive . . . . .	42
3.2	The <i>partial reuse</i> primitive . . . . .	43
3.3	The <i>birthday heap spray</i> primitive . . . . .	44
3.4	Incremental disclosure of a code pointer through JIT code. . . . .	49
3.5	Entropy of an arbitrary randomized heap pointer before and after using the timing side channel. . . . .	51
3.6	The <i>birthday heap spray</i> primitive to leak high-entropy heap ASLR with no attacker-controlled alignment or reuse. . . . .	52
3.7	Birthday heap spray's reliability and memory trade-offs. . . . .	53
3.8	Flipping a bit in an object pointer to pivot to the attacker's counterfeit object. . . . .	59
3.9	Aligned and unaligned objects allocated in a single nginx pool. . . . .	62
3.10	nginx password hash disclosure using alignment probing. . . . .	64
3.11	nginx heap disclosure using partial reuse. . . . .	66
4.1	Memory deduplication can provide an attacker control over the layout of physical memory. . . . .	77

4.2	A SO-DIMM with its memory chips. . . . .	82
4.3	DRAM's internal organization. . . . .	83
4.4	Required time and memory for templating. . . . .	90
4.5	Number of usable 1-to-0 bit flips usable in the SSH <code>authorized_keys</code> file for various modulus sizes. . . . .	91
4.6	CDF of successful automatic SSH attacks. . . . .	92
4.7	Compute power and factorization timeout trade-off for 2048-bit RSA keys. . . . .	94
4.8	CDF of success rate with increasing templates. . . . .	95
4.9	Probability mass function of successful factorizations with one flip. . . . .	96

**List of Tables**

2.1 `Sigreturn` and `syscall` gadgets and their location . . . . . 23

3.1 Time and memory requirements to leak pointers in the current im-  
plementation. . . . . 54

3.2 Time and memory requirements for our deduplication-based attacks  
against `nginx`. . . . . 67

3.3 Full deduplication rate versus deduplication rate of zero pages alone  
under different settings in Microsoft Edge. . . . . 68

4.1 Examples of domains that are one bit flip away from `ubuntu.com` that  
we purchased. . . . . 93

4.2 Memory savings with different dedup strategies. . . . . 97



## Publications

This dissertation includes a number of research papers, as appeared in the following conference proceedings <sup>1</sup>:

Erik Bosman, and Herbert Bos Framing Signals—A Return to Portable Shellcode<sup>2</sup>. In *Proceedings of the 35th IEEE Symposium on Security and Privacy, (S&P '14)*, pages 243–258. May 18-21, 2014, San Jose, California, USA.

Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector<sup>3</sup>. In *Proceedings of the 37th IEEE Symposium on Security and Privacy, (S&P '16)*, pages 987–1004. May 23-25, 2016, San Jose, California, USA.

Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos Flip Feng Shui: Hammering a Needle in the Software Stack<sup>4</sup>. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16)*, pages 1–18. August 10-12, 2016, Austin, TX, USA.

Related publications not included in the dissertation are listed in the following:

Erik Bosman, Asia Slowinska, and Herbert Bos Minemu: The World’s Fastest Taint Tracker In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID '11)*, pages 1–20, September 20-21, 2011, Menlo Park, CA, USA.

---

<sup>1</sup> Some minor editorial changes have been made to the text and some of the figures to improve readability.

<sup>2</sup> Appears in Chapter 2.

<sup>3</sup> Appears in Chapter 3.

<sup>4</sup> Appears in Chapter 4.



Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida ASLR on the Line: Practical Cache Attacks on the MMU In *Proceedings of the 24th Annual Network and Distributed System Security Symposium, (NDSS '17)*, pages 875–889, February 26 - March 1, 2017, San Diego, California, USA

Enes Göktas, Benjamin Kollenda, Philipp Kopp, Erik Bosman, Georgios Portokalidis, Thorsten Holz, Herbert Bos, and Cristiano Giuffrida Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P '18)*, pages 227–242, April 24–26, 2018, London, United Kingdom

Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos On the Effectiveness of Same-Domain Memory Deduplication In *Proceedings of the 15th European Workshop on Systems Security (Eurosec '22)*, pages 29–35, April 5–8, 2022, Rennes, France

## 1

**Introduction****Abstractions are a necessary evil**

Perhaps above all, the success of computing has been due to abstraction.

A modern computer consists of billions of transistors, wires, and capacitors, yet most programmers writing software need not concern themselves with any of them. Between the programming environment in which a typical application is written, and the silicon doing the actual computation sit layers and layers of abstractions. From arithmetic primitives, (micro) operations, context switches, device drivers, all the way to reliable network communication protocols, scripting languages, and complete game engines, each abstraction provides a simplified interface to perform a more complex task. The result is a system built up from components that can be individually tested, repurposed, and replaced.

By agreeing on the interfaces that abstractions provide, hardware and software engineers can specialize. Knowing the nitty-gritty details of a component's inner workings is not required just to interoperate with it. For instance, an application programmer does not need to be aware of the intricacies of a CPU cache as long as they know how to read and write to a variable. Likewise, hardware engineers and operating system programmers are normally unaware of the specific programs that will be running on the systems they create. Without building and relying on abstractions, nobody would get any real work done!

But all this comes with a major downside. Division of labor means that systems are built without any one person being able to oversee the whole stack. A large part of modern hardware and software engineering is about conveying what functionality and guarantees a component provides to engineers implementing other parts of a system, and this is not always straightforward.

There are many ways in which mistakes can be introduced on the border where

abstractions touch. It is relatively easy to document a component's syntactic details, such as function names and their arguments dictated by the API, and to provide a sufficient functional description for a programmer to start using such an abstraction. But getting something to work is easier than *verifying* it works all the time. Programmers make implicit assumptions and test only on specific systems.

For example, programmers often overestimate what guarantees the C programming language offers and write code that causes undefined behavior. This may work on one system, when compiled with one compiler, but malfunction when compiled with a different compiler.

On top of that, it is often infeasible to stay completely ignorant about implementation details of a given abstraction. For instance, a network filesystem behaves differently from a filesystem on a local disk, and CPU cache sizes have a big influence on how long it takes to run a computational task of a certain size. No longer is it sufficient to rely on the functional description of the API and one needs to be aware of implementation details of a system. These abstractions—where the implementation seeps through—have been coined leaky abstractions [122].

Sometimes the requirements change for an existing piece of code. An application may suddenly become faulty when it migrates from a locally controlled environment to a remote one. The change may also affect basic assumptions about security boundaries and the attack surface. For instance, the potential for compromise is higher when an application is deployed on a multi-tenant system.

Requirements may also change for implementers of the abstractions themselves. Operating systems used to run only code installed by the end user or system administrator. Nowadays, they also need to function correctly in the presence of potentially malicious actors. For example in the form of malicious websites, mobile apps, or malformed documents.

Just like any other sufficiently complex construction, abstraction layers contain flaws. Because computer systems are in essence a mathematical model, they are also brittle like a math proof. A mistake in one layer can break the guarantees of all the parts of the system that depend on it.

As if all these individual problems were not enough, in a real environment, problems across abstraction layers can interact in unforeseen ways. So, on one hand, we absolutely need abstractions to hide complexity, but on the other hand, those same abstractions also mask problems.

## Attackers create their own abstractions

A well-designed computer system is imagined by its makers as a neat stack of inter-operating components. These components work together through interfaces which provide abstractions, allowing the system as a whole to perform its intended goal. An attacker – on the other hand – has a different goal in mind. Their target is not the system as it is designed, but the system as it is actually implemented. In the

end, even the most abstract concepts will in some way have to turn into a pile of transistors, wires, and capacitors holding the right electrical charges and doing all the actual work. It is this **actual** system that an attacker needs to subvert.

Just like the designers of a system, the attacker must try to make sense of their target, and may co-opt many of the same abstractions as a lens through which they can understand its inner workings. Yet, they are not restricted to existing characterizations. Attackers have the advantage that they only need to focus on parts that further their objectives and have the choice to find the weakest links in the system. In search of flaws, they often become experts in very specialized parts of the system, knowing it better even than the people responsible for maintaining it. In the process, they come to understand system behaviors that may seem superfluous to its creators, but which turn out to be essential for exploitation.

Finding a flaw that breaks security guarantees does not automatically lead to a compromised system. An attacker may not even have direct control over the part of the system which contains a flaw. It is their job to nudge the system into a state where circumstances are just right for the flaw to be exploited.

In doing so, it is useful for the attacker to have a playground, a part of the system that is particularly exposed to manipulation by the attacker, but which may not necessarily contain a flaw itself. Any program that processes large amounts of untrusted inputs under the control of an attacker gives the attacker the ability to manipulate state in the system. For starters, attackers could cause the program to place a lot of untrusted data into its own memory. Then, if those inputs encode complex data structures, an attacker may control *where* the program stores its data. By anticipating how the program allocates memory buffers based on specially crafted inputs, they can manipulate its memory allocator into placing specific pieces of data close to each other. This is commonly referred to as *memory massaging*, or *Heap Feng-shui* [121]. Furthermore, programs may exert other application-specific behaviors advantageous to the attacker. Browsers are particularly powerful in this regard because they let an attacker even run interactive programs.

To help with creating order in the chaos that is a complex running system, an attacker can make use of *exploit primitives*, small steps that make it easier to use the system to further their goals. By repurposing existing mechanisms and using peculiarities of the implementation of the system, attackers create new building blocks which can interact with each other. Those exploit primitives are abstractions in their own right, just not as the creators of the system intended.

It is not a coincidence that the same parts of the system that perform useful tasks for the operation of the system are often also of benefit to the attacker as exploit primitives. Many of the small steps that an exploit takes – like reading, writing, and allocating memory, and performing system calls – are quite similar to the small steps that make up a legitimate program. It is by chaining those small steps that an attacker can achieve end-to-end exploitation [120; 15; 74].

## On the study of well-behaved Weird Machines

Any interaction an attacker might have with a computer system is viewed as input by the designers of that system. This input is read, and processed by code running on the system, which is controlled by its owner. For an attacker, on the other hand, interaction with a system defines how they can manipulate its state. What is input for the designer, becomes the code controlling the system in the view of the attacker, with the attacker as its programmer.

By writing this code, the attacker programs a *weird machine*. As explained by Sergey Bratus [22]: [Weird machine refers to] “*finding and programming an execution model (a machine, such as a virtual automaton) within the target via crafted inputs*”.

Subsequently, a formal description of weird machines in the context of exploitation has been given by Thomas Dullien [43]. He considers three stages: namely setup, instantiation, and execution of the weird machine. All of them are controlled by the attacker’s input. During the setup phase, the attacker guides the system into a state in which the flaw can be exploited. Then, by exploiting that flaw, the system enters a new state which does not conform to the specification, also known as a weird state. At this point the weird machine is instantiated. Now during its execution, it transitions between weird states, fulfilling the attacker’s goals.

It is natural to think of the moment when the state changes to a weird one, as its starting point. Due to broken assumptions, the program can easily behave in unexpected ways, leading to a state explosion. But lots of state manipulation happens in the setup phase. Many of the intuitions behind weird machines already apply before the system is put in a weird state.

Even before the state explosion, there are still a lot of states to work with and program like a weird machine. Exploits can often be viewed as multiple stages leading to increasing amounts of control at every step. Often, the crucial parts of the exploit happen before the moment of corruption, when the control over the system is still limited. Thus, in this thesis, we adopt a broader definition, where we view the setup phase already as a running weird machine, and the transition to a weird state as just another point in its execution.

Before the transition to a weird state, the attacker’s and the designer’s view of the program state can be viewed as orthogonal to each other. The designer thinks of the execution state in terms of the higher level abstraction which implements the specification, and which is not (yet) violated. The attacker, on the other hand, is already trying to manipulate the running program in a way that advances their goals.

The programming of weird machines in itself can be studied in absence of a flaw even if it only becomes useful once a flaw is identified. Attackers often gain extensive knowledge about complex code bases like web browsers just to be prepared for a potential bug to show up. This knowledge is transferable between different bugs.

But what if there is no bug that transitions the machine into a weird state? Does it still make sense to speak of a weird machine? In the absence of suitable bugs,

attackers can exploit interactions between layers and achieve their goals by combining primitives across them. A common thread in this thesis is the study of weird machines in one abstraction layer which can lead to exploitation when a flaw is triggered in another one.

## Contributions

This thesis explores three ways in which unforeseen interactions between abstraction layers can lead to security vulnerabilities, or make them worse.

### Framing Signals—A return to Portable Shellcode

Chapter 2 explores how the choice of implementation of an obscure part of the operating system can influence the ease with which a vulnerable application program can be exploited.

We show it is possible to write code-reuse exploits for UNIX/Linux programs which use only code that the operating system has made available in every process. In a technique we call *Sigreturn Oriented Programming (SROP)*, we abuse the stub that allows programs to resume normal execution after a signal handler has completed. We describe how to construct a weird machine by means of fake returns from signals.

It turns out that this gives attackers a lot of control over a program's execution state, simplifying writing exploits greatly. In some cases, it even allows for exploits that work regardless of what specific version of a vulnerable program is being exploited.

This serves as an example of how the unforeseen interaction between two pieces of code in the system, each developed independently, can impact the security of a system.

### Dedup est Machina

Chapter 3 examines the security implications of memory deduplication and how it can be combined with Rowhammer [69], a hardware flaw, to create exploits for programs that themselves need not have exploitable vulnerabilities. We show how memory deduplication can be viewed as a programmable weird machine capable of leaking secrets.

Memory deduplication is a performance optimization commonly found in virtualization environments, as well as on desktop operating systems. The memory subsystem regularly checks for chunks (pages) of virtual memory with identical contents. If it finds any, it points them to the same chunk of physical memory, freeing up space. This mechanism is seemingly transparent to the owners of those chunks.

Here, the implicit assumption that programs rely on for confidentiality is that there exist security boundaries that prevent the contents of private memory from

being leaked to other parts of the system. This assumption is broken by a new implementation of an old interface (in this case the virtual memory subsystem).

Writing to a deduplicated page will trigger a copy-on-write action, causing a noticeable delay which can be used as 1 bit side-channel revealing that this page was also present somewhere else in the system.

One contribution of our work is to show how deduplication can be abused in a more powerful way than via a simple 1 bit oracle.

In general, attackers often have some sort of control over the contents of a victim program's memory as programs often have to process untrusted input. In normal circumstances, this should not be a problem and would not have an effect on the security of a system. But because of the interaction with memory deduplication, it *does* become a security concern. The opportunity to place known data in a victim's memory creates a playground for the attacker and enhances their ability to leak specific secrets.

By interacting with an application, an attacker can exert some control over the very pages they are trying to leak. If a victim can be tricked into completely surrounding secrets with known or predictable data in memory, the content of a memory page ends up depending only on this sensitive data. Worse still, by controlling the alignment of secret data, or by partially overwriting (freed but still present) secrets, information with higher entropy can be leaked incrementally. In another scenario, we trick a victim web browser into creating many pages with different secrets (where we only need to know one) while also guessing many secrets at the same time. This increases the chances that one of our pages matches one of the secret pages, similar to the *birthday paradox*.

Setting up guess pages and manipulating a victim program into creating pages that are suitable to leak secrets can be viewed as programming a weird machine. Namely one that compares any two pages and leaves a measurable trace. It is worth noting that it is a well-behaved part of the system that performs the computation that is useful to us.

The final contribution is that we show how to exploit a browser by serving a web page without making use of any flaws in said browser. Instead, we use memory deduplication as an OS side channel and combine it with a hardware flaw (Rowhammer) to create an arbitrary read/write exploit primitive. The browser serves as a playground to manipulate state, in order to exploit flaws in different layers.

## Flip Feng Shui

Chapter 4 continues to explore the interaction between Rowhammer and deduplication. Here, deduplication is used as a means to get more control over an otherwise random memory corruption. We combine deduplication and Rowhammer to reliably take over one virtual machine from another one running on the same host.

While Rowhammer is a very powerful exploitation primitive on paper, allowing attackers to flip bits in otherwise bug-free code, actual exploitation can be quite

challenging for the following reason: both the memory pages and the location within these pages which can be modified are unpredictable. At the same time, once a flip is found, it can usually be reliably repeated.

First, we have to find pages in the victim's memory where bit flips would bring the victim into an exploitable state. We target their public key material. By flipping bits in the public key, we can construct a corresponding private key which grants an attacker unauthorized access to the victim's virtual machine.

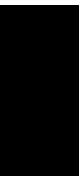
Now that we have identified virtual pages in which some bit flips will lead to compromise, we need to find physical pages on the host machine where these specific bits can be reliably modified. We do this by performing Rowhammer on our *own* memory.

In the following step, we trick the hypervisor to place the victim's vulnerable memory page onto the exact physical location vulnerable to flipping the right bit. For this, we again exploit memory deduplication, now, by replicating the victim's page in our own address space and waiting for a deduplication. This tricks the hypervisor to point the victim's page to the physical location where we can flip bits.

Finally, we do the Rowhammer step again and corrupt the right bit in the right page so that we can exploit the victim's system.

Compared to the previous chapter, we craft a different exploit primitive from the same (well-behaved) weird machine. Instead of creating a memory leak primitive, we use deduplication to perform an action, similar to Heap Feng-shui, to get the right data into the right place. Combining this with Rowhammer gives us a write primitive.





## Framing Signals—A Return to Portable Shellcode

### Abstract

Signal handling has been an integral part of UNIX systems since the earliest implementation in the 1970s. Nowadays, we find signals in all common flavors of UNIX systems, including BSD, Linux, Solaris, Android, and Mac OS. While each flavor handles signals in slightly different ways, the implementations are very similar. In this chapter, we show that signal handling can be used as an attack method in exploits and backdoors. The problem has been a part of UNIX from the beginning, and now that advanced security measures like ASLR, DEP and stack cookies have made simple exploitation much harder, our technique is among the lowest hanging fruit available to an attacker.

Specifically, we describe Sigreturn Oriented Programming (SROP), a novel technique for exploits and backdoors in UNIX-like systems. Like return-oriented programming (ROP), sigreturn oriented programming constructs what is known as a ‘weird machine’ that can be programmed by attackers to change the behavior of a process. To program the machine, attackers set up fake signal frames and initiate returns from signals that the kernel never really delivered. This is possible, because UNIX stores signal frames on the process’ stack.

Sigreturn oriented programming is interesting for attackers, OS developers and academics. For attackers, the technique is very versatile, with pre-conditions that are different from those of existing exploitation techniques like ROP. Moreover, unlike ROP, sigreturn oriented programming programs are portable. For OS developers, the technique presents a problem that has been present in one of the two main operating system families from its inception, while the fixes (which we also present) are non-trivial. From a more academic viewpoint, it is also interesting because we show that sigreturn oriented programming is Turing complete.

We demonstrate the usefulness of the technique in three applications. First, we describe the exploitation of a vulnerable web server on different Linux distributions.

Second, we build a very stealthy proof-of-concept backdoor. Third, we use SROP to bypass Apple's code signing and security vetting process by building an app that can execute arbitrary system calls. Finally, we discuss mitigation techniques.

## 2.1 Introduction

Signal handling has been an integral part of UNIX (and UNIX-like) systems ever since the very first implementation by Dennis Ritchie in the early 1970s. Signals are an extremely powerful mechanism to deliver asynchronous notifications directly to a process or thread. They are used to kill processes, to tell them that timers have expired, or to notify them about exceptional behavior. The UNIX design has spawned a plethora of UNIX-like “children” of which GNU Linux, several flavors of BSD, Android, iOS/Mac OS X, and Solaris are perhaps the best known ones in active use today. While each flavor handles signals in slightly different ways, the different implementations are all very similar.

We show that the implementation can be used as an attack method in exploits and backdoors, much like return oriented programming (ROP [114])—although the technique is different. The problem has existed, to the best of our knowledge undiscovered, for 40 years already. Moreover, now that advanced security measures like ASLR, DEP and stack cookies are making simple exploitation much harder, it probably ranks among the lowest hanging fruit available to an attacker.

In the tradition of ‘weird machines’ [42], we describe a technique for executing attacker-provided code in otherwise benign binaries. However, rather than executing shellcode directly, returning into the C library, or piecing together a program using ROP, we construct our weird machine by means of fake returns from signals. The technique, which we refer to as *sigreturn oriented programming*, is generic and we can use it both for exploits, backdoors, and system call proxies. Moreover, we prove that sigreturn oriented programming is Turing complete.

The key idea behind sigreturn oriented programming is that an attacker can abuse the way in which most UNIX systems return from a signal handler.

When the kernel delivers a signal, it suspends the process’ normal execution and changes the user space CPU context such that the appropriate signal handler is called with the right arguments. When this signal handler returns, the original user space CPU context is restored. Specifically, a program returns from the handler using `sigreturn`, a ‘hidden system call’ on most UNIX-like systems, that reads a signal frame (`struct sigframe`) from the stack, put there by the kernel upon signal delivery. The frame contains all information needed for a safe return: the values of the registers, stack pointer, flags, etc.

The problem is that anyone who controls the stack is able to set up such a signal frame. By calling `sigreturn`, attackers may determine the next state for the program, and, as we shall see, chain together `sigreturn` and other system calls and to execute arbitrary code.

Like return-oriented programming (ROP), sigreturn oriented programming (SROP) is a generic technique and we will show how we used it in exploits, backdoors and system call proxies, and across a wide variety of operating systems. Compared to ROP, it has different preconditions that in certain cases are simpler to satisfy. For instance, SROP needs only a single gadget for the exploit and for several Linux, An-

droid, and BSD distributions that gadget is always present, and better still, always located at a fixed location. In that case, attackers need not even know in advance the exact version of the executable and all the libraries, which greatly simplifies the attack when detailed reconnaissance is not possible.

For attackers, what is especially attractive about SROP compared to ROP, is its re-usability. Unlike ROP code, SROP programs are not very dependent on the content of the executable. Like traditional shellcode, this makes it possible to reuse the same SROP code across a host of applications. Moreover, the technique works on widely different instruction set architectures and operating systems. For example, we have implemented SROP successfully on 32 bit and 64 bit x86, as well as on ARM processors. Likewise, we successfully tested the technique on Linux (different distributions), Android, FreeBSD, OpenBSD, and Apple's Mac OS X/iOS. The program that we use to demonstrate Turing completeness of sigreturn oriented programming works on 32 bit and 64 bit Linux.

### Contributions

In this chapter we will describe a new type of weird machine and explain how to program it using sigreturn oriented programming. Specifically, we introduce:

1. a new generic exploitation technique, known as sigreturn oriented programming (SROP), that in some cases requires no prior knowledge about the victim application and yields reusable 'shellcode';
2. a novel and stealthy backdoor technique based on SROP;
3. a system call proxy to bypass Apple's iOS security model;
4. a proof that SROP is Turing complete;
5. possible mitigation techniques.

### Applications

We demonstrate the practicality of our exploitation technique using a vulnerability found recently in the Asterisk web server. We show that using SROP, we can construct a single exploit which works for different versions of the same program on different distributions of Linux, including Debian Wheezy (released in May 2013), Ubuntu LTS 12.04 (the latest Long Term Support version of Ubuntu, released in 2012 and supported for five years), and Centos 6.3 (released in 2012 and supported for 10 years).

We then demonstrate the wider practical applicability of the technique by means of a stealthy backdoor. The backdoor is hard to find even with state of the art forensics tool. For example, even a complete memory dump of the process will not reveal any backdoor instructions, as all logic is hidden in the data in the form of an SROP program. Finally, we show how we can bypass Apple's well-known security model that consists of elaborate code vetting and prevents malicious apps from making it

into the App Store. In our case, there is no need to add any malicious instructions to the application whatsoever, so it will be extremely hard to detect. However, provided with the right inputs, it will function as a system call proxy that can be programmed using sigreturn oriented programming.

### Outline

The remainder of this chapter is organized as follows. In Section 2.2, we place SROP in the context of related work. In Section 2.3, we discuss weird machines and how to program them, as well as the role of return-oriented programming. Signal handling is the topic of Section 2.4. Section 2.5 discusses the SROP technique in detail. We discuss our exploitation technique in Section 2.6, a stealthy backdoor in Section 2.7, and our iOS system call proxy in Section 2.8. Turing completeness will be discussed in Section 2.9. In Section 2.10 we discuss possible mitigation techniques. Finally, we conclude in Section 2.11.

## 2.2 Related work

Our work fits in the general category of weird machines. The term weird machine was originally coined by Sergey Bratus and was quickly picked up by other researchers [42; 127]. It is a generic term to describe systems in which unwanted or unexpected powerful computations are found to be possible. For instance, researchers have shown that it is possible to use weird machines constructed from arcane parts, such as ELF symbol relocation logic in the dynamic loader [117] and even the x86 virtual memory subsystem [64].

For this chapter, Return-Oriented Programming (ROP) is most relevant [114]. ROP is an exploitation technique that allows attackers to execute code in the presence of security measures like non-executable stacks and code signing. As a precondition to a successful ROP exploit, the attacker needs to gain control over the stack and obtain valid code pointers. The attacker then manipulates the return address to jump to a sequence of instructions that ends with a return. As the attacker controls the stack, she can keep jumping to code gadgets and thus string together a program.

A variant of the same technique, known as jump oriented programming, uses jumps instead of returns [15]. Finding the appropriate gadgets and stringing them together is hard, but researchers have shown that it is possible to construct ROP compilers [110; 81] to make this easier. Similarly, modern protection mechanisms like address space randomization [100; 14; 115] make it hard to find the addresses of the code snippets. On the other hand, given an initial code pointer, it is often possible to extract the remaining code pointers necessary for a successful exploit [119]. Also, in Linux and Windows, executables with fixed load addresses, and libraries incompatible with ASLR, may lead to parts of the address space that are constant [108].

Since ROP and JOP have quickly become popular with attackers, many research groups have tried to mitigate the issues, for instance by trying to remove useful gadgets [76; 94], permuting the order of functions [14; 67], or dynamic binary instrumentation [33; 26]. Also, researchers have proposed to use in-place code randomization with low overhead [99] and by monitoring branch histories [98]. Even though none of these solutions have stopped attackers from using ROP, they keep raising the bar. Many of the mitigations do not work for SROP. For instance, the gadgets used by SROP are essential for the functioning of the binary and cannot be removed. We believe that SROP may be among the most convenient methods of attack even against programs that apply all common security measures.

At a superficial level, sigreturn oriented programming shares some of the characteristics of ROP. We will see that sigreturn oriented programming also requires setting up the appropriate values on the stack and ‘returning’ to a value determined by the attacker. But the technique is different. Specifically, SROP uses fake signal frames and does not really depend on finding gadgets and stringing them together. For this reason also, SROP code has better re-usability.

The use of signals themselves in exploitation has been thoroughly documented by Michael Zalewski in: "Delivering signals for fun and profit" [134]. The author shows that there are many pitfalls to consider when programming signal handlers. The pre-emptive nature of signals can lead to the use of data structures while they are in an inconsistent state, corrupting them in the process, or to make an application do unexpected things while in a state with elevated permissions. Our work is different, in that we use fake signal frames as partial instructions in a weird machine.

A user space call which is somewhat similar to the `sigreturn` system call is `longjmp`. In "Bypassing stackguard and stackshield" [24], overwriting a pointer to a user context subsequently used by `longjmp` is mentioned as an exploitation vector. It is noted however, that the existence of such a pointer in an exploitation context is extremely rare.

Microsoft Windows does not implement POSIX signals. Its fault handling mechanism is designed to integrate well with C++’s exception handling. Through a mechanism called Structured Exception Handling (SEH), it allows functions to unwind the stack and do some processing until the exception is caught in an earlier stack frame. As exception handler pointers are put on the stack, this has been a steadfast overwrite target for stack buffer overflows on Windows [78] and several mitigation techniques have been proposed and implemented [89]. However, since SEH on Windows does not return to the state before the handler was executed, they cannot provide a mechanism for SROP-like exploits.

An independent effort to attack iOS devices that also builds on benign apps becoming evil is provided by Jekyll [128]. Like our work, the authors deliberately introduce vulnerabilities in the app, so that they can easily change the control flow later, by exploiting it by means of a ROP exploit. Likewise, we show that it is now *easy* to circumvent all of iOS’s defensive mechanisms, including DEP [88], ASLR [100; 14], sandboxing, app review and code signing. Unlike Jekyll which needs

to carry its own gadgets, the gadget for sigreturn oriented programming is already present. Thus, even detection techniques that specifically scan for Jekyll-like functionality are no longer effective. Finally, we can use sigreturn oriented programming in a post-exploitation scenario. For instance, we can use the system call proxy for a jailbreak—allowing attackers to jailbreak via a root process that was not written by them.

In the next section, we will revisit the current trend among attackers to move away from regular shellcode injection toward attacks based on code reuse. We will argue that because of this, exploitation is getting harder and that SROP may be a useful new weapon in the hands of an attacker—removing some of the obstacles that an attacker encounters in modern systems.

## 2.3 On the construction of weird machines (or: why exploitation is getting harder)

Sigreturn oriented programming is not unlike other exploitation techniques where attackers execute foreign ‘code’ in an existing program. Not so long ago, doing so was relatively straightforward. Typically, attackers managed to find a buffer overflow vulnerability on the stack which allowed them to overwrite the return address. All they needed to do was make the return address point to their own machine code which would be stored in a buffer or environment variable. As soon as the function returned, the program would continue executing the attackers’ code.

### Code reuse attacks (ROP, JOP, and ret-into-libc)

With modern protection measures, such traditional exploitation of memory corruption bugs, where machine code is injected by the attacker and directly executed, has become rare. Specifically, data execution prevention (DEP [88]) features, present in practically all modern CPUs and operating systems, separate machine code and writable data. In other words, attackers can no longer execute their shellcode directly. To bypass such security measures, they must resort to different exploitation techniques. Rather than injecting regular machine code, modern attacks typically reuse logic or code from the executable itself using ret-to-libc [120] or return oriented programming [114]. In this way, attackers can construct strange automata to do their bidding.

Constructing such an automaton is not easy. Over the past few years, the security research community has especially focused on generic techniques like *return oriented programming* (ROP) [114] and *jump oriented programming* (JOP) [15]. These techniques make use of chunks of executable code present in the program itself. Chained by indirect control flow instructions, these chunks thread together a sequence of low level instructions which do what the attacker wants. An attacker that



controls an application's stack because of a buffer overflow may look in the original application's binary for small sequences of instructions that do something interesting and end with a return instruction. For instance, a sequence to add two registers, or to load or store a register, etc. These sequences are known as 'gadgets'. If the attacker manages to divert the control flow to one of the gadgets, the gadget will execute the first small part of the attacker's code and then execute a return. Since the attacker controls the stack, the address to return to is also under the attacker's control. So the attacker returns to another gadget. By chaining together gadgets, the attacker can execute arbitrary code, written for an instruction set that consists of the gadgets and a stack pointer that functions as a strange sort of program counter.

While such a procedure may sound straightforward, a working ROP exploit is often highly complex. For instance, a ROP attack may require an attacker to manipulate the state of the program to prepare it for exploitation, by (1) making it leak information about code pointers (for instance, by means of a format string attack), (2) lining up heap objects in a specific way to pave the way for a dangling pointer exploit (using advanced heap feng shui [121]), (3) gathering enough useful code gadgets from the binary to chain together the attacker's final program (which typically assumes that the exact versions of the executable and libraries are known), and (4) positioning the appropriate data on the stack, and (5) diverting control to the first gadget.

The ability of a program to go beyond its specification allows it to act as a *weird machine* [42; 127]. This machine can thereafter be manipulated, programmed by an attacker just like any other programmable machine. The attacker's input now serves as its (rather peculiar) machine code, giving her control over the program's execution, far beyond its intended use.

### What the attacker wants

Some techniques for programming weird machines are very application specific. They depend very much on the flaw(s) that make up a vulnerability in a single application and cannot be re-used for the next bug. Others, like heap feng-shui with Javascript [121], apply to a whole class of applications and considerable effort is spent on making them applicable across systems. Clearly, reusable techniques are more valuable than a trick that can be used only once.

From an attacker's perspective, we therefore distill the following objectives for the construction and use of weird machines:

1. **Re-usability:** reusable techniques are better than one-offs. Ideally, we would like to be able to reuse the same exploits for multiple programs.
2. **Simplicity:** the less manipulation is needed *a priori*, the better. If the list of manipulations is long and complex, the attack may be difficult to pull off for some programs.

3. **Generality:** a technique that can be used for exploits is good, but it is even better if it can also be used for backdoors and other purposes.
4. **Variety:** the more choice for exploiting a program, the better. If we have more than one attack vector for a program with different preconditions, the probability of finding one that fits a target program increases.

As explained above, a typical exploit in reality consists of a sequence of different techniques, where more application-specific code gymnastics bootstrap more generic exploitation methods such as ROP, and then possibly the attacker's native code. Again, the more generic the technique, the better the exploit code can be re-used for other vulnerabilities.

### ROP makes attacks harder

As we saw earlier, exploitation techniques have evolved in answer to the protection measures that are now employed on modern systems: direct shellcode execution has become rare, and ROP exploits have become popular. Unfortunately for attackers, ROP is not such an easy drop-in solution as normal shellcode used to be, because useful code-snippets and their addresses are different for every new binary for which an exploit is being written.

It is true that ROP compilers make it possible to automatically generate a useful ROP chain for most target binaries, but even with state-of-the-art ROP compilers, return-oriented programming complicates the attacker's life significantly. With a traditional stack overflow and without data execution prevention, the same exploit could very easily work across multiple binaries with the same vulnerability. Now, however, the attacker needs to know exactly which binary the victim uses to feed it to a ROP compiler, or possibly even build a ROP chain manually. It may be difficult to determine the exact version of a binary because an application has had several updates which may or may not have been installed. In the worst case, the binary is a custom build with an unknown compiler and unknown compile flags. Even if the attacker can determine the version, it still is a lot of work to do this for every binary.

The other pre-conditions for a successful ROP attack are important as well. Besides control over the stack and an initial control-flow diversion, the executable needs to leak the address of a code pointer which, depending on the application, may be hard.

In the remainder of this chapter, we will see that sigreturn oriented programming scores well on all of the above four criteria. SROP code is portable [objective 1], the attack is simple (e.g., it requires a minimal number of gadgets that in quite a few systems can be found at fixed locations) [objective 2], it has many applications [objective 3], and it enlarges the attackers' repertoire with different preconditions [objective 4].

## 2.4 Signal delivery on UNIX systems

Signals have been an integral part of UNIX systems almost since its inception [106]. Initially little more than a convenient abstraction around hardware interrupts, the mechanism was later adapted as a generic system, able to receive notifications both from the kernel and from other processes. By registering a signal handler function, a process can deal with asynchronous notifications outside of its normal control flow.

Sigreturn oriented programming requires a thorough understanding of how signals are handled on UNIX systems. In particular, we will zoom in on the way the system restores the process state upon returning from a signal handler.

### 2.4.1 Delivering the signal

When a kernel delivers a signal to a process, the normal execution flow of the process is temporarily suspended. Specifically, the kernel changes the user space CPU context such that a previously registered signal handler function is called with the right arguments. When this signal handler returns, the original user space CPU context is restored.

In true UNIX fashion, this is implemented in an elegant way that requires no bookkeeping on the kernel side. The suspended user context is simply saved on the process' stack and restored from the stack when the signal handler returns. Initially, this was done simply in the interrupt trap handler (Fig. 2.1). But the introduction of virtual memory and hardware memory protection made this impossible, as a user-space signal handler could no longer directly return to the interrupt routine. This led to the introduction of the `sigreturn` system call in 4.3BSD. `Sigreturn` takes as first argument a pointer to the user context to be restored. A piece of trampoline code is placed in the user address space which first calls the signal handler and then does a `sigreturn` system call to signal the kernel it should restore the old user space context.

Linux does something similar to the BSDs, except that it executes the signal handler directly. When the signal handler returns, it returns to an address written there by the kernel. A stub at this address is then responsible for calling `sigreturn`. Unlike on BSD (and iOS/Mac OS X), there is no argument to `sigreturn`. The kernel simply loads the user context from the stack.

Figure 2.2 shows the top of the stack of a 64 bit x86 Linux process when a signal handler exits.

### 2.4.2 Sigreturn and Data Execution Prevention

The `sigreturn` calling trampoline has to be in user space memory. It used to be custom for kernels to write the trampoline code together with the user context on the stack. A returning signal handler would just jump to this code on the stack. However, this requires an executable stack, allowing for easy shellcode injection on

```

NSIG = 0

# interrupt vector
tvect:
    mov r0,-(sp);
    # load signal handler function
    mov dvect+[NSIG*2],r0;
    br lf;
    NSIG=NSIG+1
    # repeated a total of 20 times (for 20 signals)
    ...
1:
    # push the register state on the stack
    mov r1,-(sp)
    mov r2,-(sp)
    mov r3,-(sp)
    mov r4,-(sp)
    # call the signal handler
    jsr pc,(r0)
    # restore registers
    mov (sp)+,r4
    mov (sp)+,r3
    mov (sp)+,r2
    mov (sp)+,r1
    mov (sp)+,r0
    # return from interrupt
    rtt

```

**Figure 2.1:** Excerpt from `s5/signal.s`, UNIX V6 interrupt routine (comments added for clarity). We see that even in these early versions, the suspended CPU state was stored on the stack.

the stack. Nowadays, depending on the architecture, the `sigreturn` trampoline code is either provided in an executable memory page provided by the kernel, allowing the kernel to know its location, or it resides somewhere in `libc`.

## 2.5 Sigreturn Oriented Programming

In this section, we discuss the key ideas behind `sigreturn` oriented programming. To do so, we first introduce the way Linux and other UNIX flavors return from signals in some detail and then discuss how `sigreturn` is vulnerable to abuse.

### 2.5.1 Signal delivery on Linux

At the new top of the stack, the kernel writes the code address that will become the return address for the signal handler. This code address points to a small stub which does nothing more than invoke the `sigreturn(0)` system call. This `sigreturn()` call undoes everything that was done in order to invoke the signal handler (changing the process's signal mask, switching stacks). Thus, it restores the process's sig-

nal mask, switches stacks, and restores the process's context (registers, processor flags)—making the process resume execution exactly at the point where it was interrupted by the signal.

## 2.5.2 The `sigreturn` system call on Linux

While `sigreturn` is a system call, called like any other, it is special in the sense that no user space program ever needs to be aware of its ABI. The kernel is responsible for setting up the user space stack in such a way that it is eventually called.

As mentioned earlier, the trampoline code invoking `sigreturn` used to be on the stack in the *signal frame* in the `pretcode` field, but because executing this trampoline requires an executable stack, it is no longer used on recent kernels. Interestingly, vestiges still exist on Linux i386 to help gdb identify the signal frame. In reality, however, the stub has since moved to the *virtual dynamic shared object* (*vdso*), a kernel-supplied piece of code mapped in every process' address space. On x86-64 Linux, the stub is present in `libc` itself. It must be supplied to the kernel when registering signals using the `sigaction.sa_restorer` field. In both cases, the `sigreturn` stub address is normally randomized by ASLR [100; 115].

When `sigreturn(0)` is called, the kernel will use the user space stack pointer to find the *signal frame* which it had stored there previously and load the original user context into the CPU's registers. In this way, it restores the original context for the interrupted process.

Figure 2.2 shows a signal frame that a 64 bit Linux kernel would place on the stack. We see that the frame contains register values, including the stack pointer (RSP), the instruction pointer (RIP), the segment registers (CS, FS, and GS) and the flags. The return address is at the top of the frame, followed by the user context, and the registers. The regular registers may hold arguments and we will use them in this way also.

The floating point state pointer points to the saved floating point unit state. It is only important if there is any floating point state to restore, i.e., if there were any floating point operations before the signal arrived. If the pointer is `NULL`, on the other hand, the kernel assumes that there were no such operations and ignore it.

Signal frames on other architectures look very similar, albeit that the register context stores by definition other, architecture-specific registers and values.

## 2.5.3 `Sigreturn` on other UNIX flavors

Different UNIX and UNIX-like systems implement returning from signals in slightly different ways. On BSD, Mac OSX and iOS, `sigreturn` is similar, except that it is part of the ABI and that it takes the location of the `struct sigframe` as first argument. On OpenSolaris there is no `sigreturn`; restoring the original context happens in user space and is handled completely by `libc`, which provides a wrapper around signal handlers and restores the user space context. While other operating systems

0x00	rt_sigreturn()	uc_flags
0x10	&uc	uc_stack.ss_sp
0x20	uc_stack.ss_flags	uc_stack.ss_size
0x30	r8	r9
0x40	r10	r11
0x50	r12	r13
0x60	r14	r15
0x70	rdi	rsi
0x80	rbp	rbx
0x90	rdx	rax
0xA0	rcx	rsp
0xB0	rip	eflags
0xC0	cs / gs / fs	err
0xD0	trapno	oldmask (unused)
0xE0	cr2 (segfault addr)	&fpstate
0xF0	__reserved	sigmask

**Figure 2.2:** The signal frame as it looks like in Linux 86-64

also provide interesting avenues for exploitation, we will restrict ourselves to various flavors of Linux, BSD, and iOS in the remainder of this chapter, often using Linux as an illustrative example.

### 2.5.4 Abusing sigreturn

Restoring the context of an interrupted process by loading a previously saved stack frame is convenient because it relinquishes the responsibility of the kernel to keep track of the signals it delivered. However, it also has a major drawback: the kernel does not keep track of the signals it delivered. In other words, there is no way of telling whether a `sigreturn` is legitimate.

By setting up a correct `struct sigframe`, loading the right system call number, and executing a system call instruction, an attacker can trivially fool the kernel into acting like a signal handler just finished. In that case, the kernel will load a user space context constructed by an attacker from the stack.

In the remainder of the chapter we will explore the unintended consequences of being able to do a `sigreturn` on arbitrary data.

## 2.6 SROP for exploitation

As mentioned, attackers can use sigreturn oriented programming for different purposes. First, we will outline two exploitation techniques, a simple generic technique followed by a more flexible method for 64 bit Linux processes. This second method is more complicated, but has weaker pre-conditions. In later sections, we discuss other uses of SROP.

### 2.6.1 A simple `execve()` SROP exploit

The final result of many UNIX exploits is often an attacker controlled shell. In this section we will outline an exploit which will call `execve` to start a shell with arbitrary arguments.

For any exploitation to be successful, certain pre-conditions have to be met. For instance, for direct shellcode execution, attackers must be able to load their code in executable memory and divert control to this buffer. ROP requires code pointers, gadgets, control of the stack, a control flow diversion, etc. Likewise, successful SROP exploitation using this technique is possible if the attacker satisfies the following pre-conditions:

1. The attacker should have control over the instruction pointer (for example due to a return instruction on the overwritten stack).
2. The stack pointer should be located on attacker controlled data and *NULL* bytes must be allowed (e.g., in an overflow). For BSD, Mac OS X and iOS, a function pointer overwrite with a user-controlled buffer as first argument is also a possibility. In this case there is no need for any user controlled data on the stack.
3. The attacker knows the address of a piece of data controlled by the attacker. This could be the overwritten stack, but does not necessarily have to be the same location.
4. The attacker knows the location of code calling `sigreturn`, or `syscall`, in case the attacker can control the CPU register which passes the system call number.

### 2.6.2 Finding a sigreturn gadget

As with return oriented programming, SROP needs some knowledge about the location of code in a process' address space. But unlike ROP, SROP really only needs to know the location of a single gadget, namely, the call to `sigreturn`. In our exploit we also make use of an extra gadget which does arbitrary system calls, but this additional gadget is contained in the `sigreturn` gadget. For this, we simply skip the instruction where the system call is loaded. In some cases we may be able to control the CPU register responsible for passing the system call. In that case we don't need

Operating system		Gadget	Memory map
Linux	i386	sigreturn	[vdso]
Linux < 3.11	ARM	sigreturn	[vectors] 0xffff0000
Linux < 3.3	x86-64	syscall & return	[vsyscall] 0xffffffff600000
Linux ≥ 3.3	x86-64	syscall & return	Libc
Linux	x86-64	sigreturn	Libc
FreeBSD 9.2	x86-64	sigreturn	0x7fffffff000
OpenBSD 5.4	x86-64	sigreturn	sigcode page
Mac OS X	x86-64	sigreturn	Libc
iOS	ARM	sigreturn	Libsystem
iOS	ARM	syscall & return	Libsystem

**Table 2.1:** Sigreturn and syscall gadgets and their location

a **sigreturn** gadget, and a **syscall** gadget will suffice as will be elaborated in our second exploitation technique.

As it turns out, it is surprisingly easy to find these gadgets on some architectures. On FreeBSD 9.2 on x86-64, there is a fixed memory page containing **sigreturn**. Linux on ARM, before version 3.11 (this includes all current versions of Android as well as the latest long term stable version (3.10) has a fixed **[vectors]** map with **sigreturn** gadgets. Furthermore, many versions of Linux on x86-64 have a fixed **[vsyscall]** map at with a **syscall & return** gadget (see Section 2.6.4).

If **sigreturn** is located in **libc**, and there are no known gadgets beforehand, an attacker might need to leak **libc** code addresses in order to obtain a **sigreturn** gadget. On systems which do library pre-linking, **sigreturn** gadgets may be the same system-wide, which allows local exploits to find the right gadgets. Table 2.1 shows the location of useful gadgets in various Operating Systems.

### 2.6.3 Executing system calls

As most architectures pass system call parameters to the kernel through registers (a notable exception being BSD on i386), doing a **sigreturn** allows us to do an arbitrary system call. By setting our stack frame's instruction pointer to the address of a **syscall** instruction and filling in the registers that pass the system call number and its arguments we effectively load a state in which a system call of our choice gets executed.

Since we know the location of some data controlled by the attacker, we can now do an **execve** system call to, say **/system/bin/sh** on Android, using pointers to our data for **execve**'s filename and **argv** parameters and using the fixed **sigreturn & syscall** gadgets from the **[vectors]** page.



### 2.6.4 SROP exploitation on Linux x86-64 using a system call chain

Up until now we have assumed that we knew about memory locations with attacker controlled data and that we had knowledge of a `sigreturn` gadget. For the next exploitation method we won't have to know exact locations of attacker controlled data in the address space and we will not require any knowledge about code from the application itself. This exploitation technique is targeted at x86-64 systems running a Linux kernel older than version 3.3 and using this method, we will exploit different versions of a vulnerable Asterisk server using the exact same exploit on different systems.

Our new pre-conditions are:

- The stack pointer should be located on attacker controlled data and `NULL` bytes must be allowed (e.g., in an overflow).
- The attacker should have some control over `RAX`. Specifically, `RAX` should contain the value 15.
- The attacker should have control over the instruction pointer `RIP` (for example due to a `RET` instruction on the overwritten stack).
- The attacker should know the location of a (any!) single writable page in memory. The location or content of the binary's code is not important. If the attacker manages to leak a writable address, the executable can even be position independent and the exact identity of the binary completely unknown.
- The target's system implements native `vsyscall`, which is the default on pre-3.3 kernels such as those used on Debian 7.0 (2013), Ubuntu Long Term Support (2012) and CentOS 6 (2012).
- The attacker controls data sent to a file or socket with a known file descriptor number.

In general terms, the SROP attack works by repeatedly setting up fake signal frames and returning from a "signal" to regular system calls and back. To explain how this is possible, we first describe the details of system call execution.

#### Useful gadgets on Linux x86-64: `vsyscall`

`Vsyscall` is a fast system call interface for 64 bit Linux. It was created to speed up certain time-sensitive system calls. Instead of using the `syscall` instruction, an application could simply jump to static addresses in a static page set up by the kernel (see Figure 2.3). The kernel provided user space code at these addresses which implemented `time()`, `gettimeofday()` and `getcpu()`. Any privileged data needed for these system calls to complete (such as the current time,) was provided by the kernel in data structures in the same page.

This turned out to be a security nightmare, as it provides attackers with a heap of useful gadgets for exploitation at fixed addresses, identical for every running process.

```

$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 fe:00 786514      /bin/cat
0060b000-0060c000 r--p 0000b000 fe:00 786514      /bin/cat
0060c000-0060d000 rw-p 0000c000 fe:00 786514      /bin/cat
01148000-01169000 rw-p 00000000 00:00 0         [heap]
7ff9bbc54000-7ff9bbdd1000 r-xp 00000000 fe:00 659315 /lib/x86_64-linux-gnu/libc-2.13.so
7ff9bbdd1000-7ff9bbfd1000 ---p 0017d000 fe:00 659315 /lib/x86_64-linux-gnu/libc-2.13.so
7ff9bbfd1000-7ff9bbfd5000 r--p 0017d000 fe:00 659315 /lib/x86_64-linux-gnu/libc-2.13.so
7ff9bbfd5000-7ff9bbfd6000 rw-p 00181000 fe:00 659315 /lib/x86_64-linux-gnu/libc-2.13.so
7ff9bbfd6000-7ff9bbfdb000 rw-p 00000000 00:00 0
7ff9bbfdb000-7ff9bbffb000 r-xp 00000000 fe:00 659338 /lib/x86_64-linux-gnu/ld-2.13.so
7ff9bcbef000-7ff9bcbf2000 rw-p 00000000 00:00 0
7ff9bcbf8000-7ff9bcbfa000 rw-p 00000000 00:00 0
7ff9bcbfa000-7ff9bcbfb000 r--p 0001f000 fe:00 659338 /lib/x86_64-linux-gnu/ld-2.13.so
7ff9bcbfb000-7ff9bcbfc000 rw-p 00020000 fe:00 659338 /lib/x86_64-linux-gnu/ld-2.13.so
7ff9bcbfc000-7ff9bcbfd000 rw-p 00000000 00:00 0
7fff77c2c000-7fff77c4d000 rw-p 00000000 00:00 0         [stack]
7fff77c61000-7fff77c62000 r-xp 00000000 00:00 0         [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]

```




Figure 2.3: Vsyscall is mapped in a process' address space

These gadgets include `syscall & ret`: a `syscall` (0f 05) followed by a `return` (c3). For this chapter, we are particularly interested in this `syscall & ret` gadget, because it allows one to execute a system call of choice, provided we can set the appropriate system call number in `RAX` (see Figure 2.4).

While gadgets seem to differ across *distributions*, useful gadgets appear to stay the same after kernel security *updates*. In other words, for systems running stock distribution kernels, it should be easy to determine the addresses of gadgets—even remotely! In addition, as the kernel keeps track of wall-clock time in the *same page* as the executable code, a patient attacker could just wait for the binary representation of wall-clock time to contain a (small) gadget of his/her choice and then jump to (the least significant bytes of) the wall-clock time field.

Due to its security risks, `vsyscall` was deprecated in Linux 3.1 and by default the fast user space routines now just call their normal system call counterparts. While this eliminated some harmful gadgets, due to its simplicity, it did result in the remaining gadgets, such as `syscall & ret` having *stable addresses across all distributions*. The reason is that the C file containing `vsyscall` was replaced with an assembly file that contains several `syscall & ret` gadgets. The assembly file always generates the same binary.

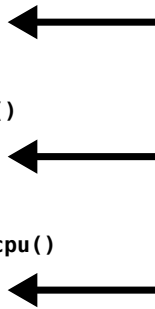
At the same time, the Linux developers added a second `vsyscall` emulation mode, which emulates vsyscalls using a trap-based mechanism. As a result, we cannot make use of the system call gadgets from this page. Note, however, that for many kernels this is not the default configuration<sup>1</sup>. Even if, there is no fixed location for system call gadgets, we may still find one in the executable itself—after all, we only need one small gadget. In that case, we lose the property of not needing reconnaissance, but the exploit still works. As a long-term release, the 3.2 kernel is

<sup>1</sup>It is certainly not default for all kernels prior to Linux 3.3. and even post-3.3 kernels allow one to boot without emulated `vsyscall`.

```

0xffffffff600000: mov $0x60,%rax ; gettimeofday()
0xffffffff600007: syscall
0xffffffff600009: retq
0xffffffff60000a: int3
...
0xffffffff600400: mov $0xc9,%rax ; time()
0xffffffff600407: syscall
0xffffffff600409: retq
0xffffffff60040a: int3
...
0xffffffff600800: mov $0x135,%rax ; getcpu()
0xffffffff600807: syscall
0xffffffff600809: retq
0xffffffff60080a: int3
...

```



**Figure 2.4:** The vsyscall page contains the `syscall` & `ret` gadget multiple times

present in many distributions today, such as Debian 7.0 (released in 2013, supported for 10 years) and Ubuntu 12.04 Long Term Support (released in 2012 and supported for 5 years)

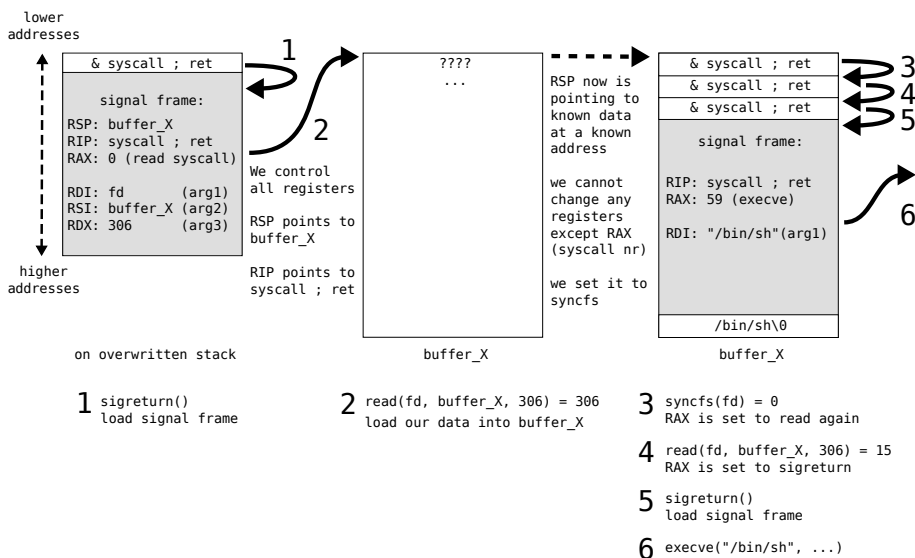
### 2.6.5 Bootstrapping to arbitrary code execution

In this section, we show how the `sigreturn` can help the attacker to execute arbitrary code. To do so, we make use of a chain of system calls and `sigreturns` that serve as an example. There may well be other sequences that an attacker can use to achieve the same effect.

The first thing we do is create a fake signal frame and trigger a `sigreturn`. We first explain what that first signal frame looks like.

In order to execute a successful `sigreturn` system call, we must ensure that the kernel does not trip over any bad values in the signal frame. The first requirement is that the code segment register is restored correctly. On x86-64, when running in 64bit mode, the code segment register should contain the value `0x33`. The second requirement is that the `fpstate` pointer is not a wild pointer. `Fpstate` points to the saved floating point unit state and if its address not valid, or the saved floating point state is not correct, our application will crash. This seems like a problem, since our assumption is that we do not yet know of any attacker controlled data on a known address. Luckily, when `fpstate` is `NULL`, Linux assumes no floating point operations had been used before the signal arrived. In this case, it clears the FPU state is and `sigreturn` succeeds.

When we execute `sigreturn`, we have complete control over the registers, as



**Figure 2.5:** Steps involved in the Linux x86-64 SROP exploit

well as over the program counter (RIP.) But we have also lost any chance of using any ASLR information present in the registers as they have been overwritten. This is why we need the address of a writable page. We will point the stack pointer in our signal frame to the bottom of this page. By filling in the necessary registers and pointing the program counter to our `syscall & ret` gadget, we can set up and subsequently execute any system call we like (see also Figure 2.5).

We will use this power to set up a `read()` system call. The `read` will read in attacker data and store it under the stack pointer. When `read` finishes, the attacker's data will serve as the return address. In our case, the attacker points it *again* to the `syscall & ret` gadget.

#### Steps 1 and 2: sigreturn and read

Summarizing, in our first two steps, we create a fake signal frame on the attack, as explained above. In the signal frame the RSP value will point to the writable page, the value for RAX will be 0 (indicating a `read` system call), and there will be appropriate values in the registers that serve as the arguments to the `read` system call. Then, the attacker diverts the program's control flow to the `syscall & ret` gadget to execute the system call and read the attacker's data on the new stack (the writable page). Because the attacker controls the return address, he can simply return to the `syscall & ret` gadget—not unlike what one would do in regular ROP.

At this point the attacker has no choice but to keep all the system call arguments the same, as the kernel does not change them during the execution of a system call. But RAX will contain the number of bytes read. This is important and the attacker uses it to select the next system call to execute. As mentioned earlier, the RAX

register indicates which system call to execute.

### Step 3: a necessary NOP

Specifically, the attacker chooses to read 306 bytes. Not only does this give quite some data which is now at a known location, but 306 also happens to be the system call number for `syncfs(int fd)`—our third step in the exploit. The `syncfs()` system call takes a file descriptor as first argument and flushes all disks belonging to this descriptor. Since our file descriptor is a socket, this will effectively be a no-op returning 0 in RAX. The attacker again makes sure to return to the `syscall & ret` gadget.

### Step 4: another read to set RAX

On x86-64, the value 0 happens to be the system call number for `read`, allowing the attacker to again read data onto the new stack—our fourth step. This time the attacker makes sure to send only 15 bytes, so that the value of RAX is now 15.

### Steps 5 and 6: a `sigreturn` to execute anything we like

As mentioned earlier, 15 is also the system call number for `sigreturn`, so we are back to where we started, but with an important difference: there is data controlled by the attacker at a known address. The following `sigreturn`, our fifth step, is again free to load an arbitrary system call into the registers, which enables the attacker to do anything he wants. For instance, he can execute an `mprotect` system call and jump to traditional shellcode, or an `execve` with the right arguments to spawn a shell.

## 2.6.6 Exploiting the Asterisk web server

We have tested our exploitation technique on a recent version of Asterisk that is vulnerable due to an unbounded stack allocation bug (CVE-2012-5976). The vulnerability has been described in depth at EIP blog [44]. Our exploit is entirely new and targets multiple binaries.

The unbounded stack allocation vulnerability occurs when a pre-authentication HTTP POST request to Asterisk's web management console allocates HTTP post data on the stack. It uses the `content-length` header sent by the client to determine how much data should be allocated. Yet it does not check whether this size is within reasonable bounds. Specifying a `content-length` of about the size reserved for the stack allows us to 'jump' with our stack pointer to the stack of a second thread in the asterisk process. Having jumped to the second thread's stack, we can start sending our post data, which will promptly overwrite the second stack.

By making sure this second thread is also initiated by us, we can overwrite this thread's stack while it is waiting on a blocking read. When we are done corrupting the stack, we send 15 bytes to the thread with the corrupted stack and when it returns to our `syscall & ret` gadget, it calls `sigreturn`, setting in motion our bootstrap method from Section 2.6.5.

We still need to fulfill our two remaining requirements: we need to control a file descriptor and we need to know a writable page. For the writable page we guessed a page in the binary's data section. By default, with non position independent execu-

bles compiled with gcc, the data section comes directly after the code section, and the code section starts at a fixed offset of 0x40000. The size of the code section does not vary much between different versions of Asterisk compared to the absolute size of the data section, therefore it is quite easy to guess a writable page in the Asterisk binary.

In order to pick the right file descriptor, we open a large number of connections to the vulnerable section and send the same data over all sockets. By picking a high value for the file descriptor we can be fairly sure that we have selected a socket that we opened.

We tested this exploit on three different vulnerable versions of the Asterisk program on different Linux distributions: Debian Wheezy (released in May 2013), Ubuntu LTS 12.04 (the latest Long Term Support version of Ubuntu, released in 2012 and supported for five years), and Centos 6.3 (released in 2012 and supported for 10 years).

The exploit worked on all Linux distributions we tried. Moreover, the re-usability of the exploit is hinted on somewhat by the fact that the code for all three versions was almost exactly the same. The only difference, was that the `syscall & ret` gadget at Centos was at a slightly different location.

## 2.7 SROP as a backdoor

Another possible use for sigreturn oriented programming is as a stealthy backdoor. By injecting signal frames into a process' address space and by either creating an extra thread in a process, or by simply replacing a process' execution with our own, it is possible to keep a presence on a system, while appearing to have left.

Developers of backdoors are keen to avoid detection. Unfortunately, injected shell code will look suspicious when a memory dump is viewed with forensics tools. Hiding all logic in data seems to be more stealthy. While in principle this can also be done using ROP, we will show that for sigreturn oriented programming it can be done in a completely generic way, which works for all processes and requires no complex ROP compiler.

For our exploitation example, we assumed that the only gadget available to us was `syscall & ret` and this was provided to us as a non-ASLR gadget by the `vsyscall` page. For our backdoor we will no longer be depending on the `vsyscall` page. We will also no longer require the architecture to be 64 bit x86 as we need not use the x86-64 specific bootstrap method of going from one system call to another which made the return values of one system call the system call number of the next.

In our backdoor scenario the attacker uses `ptrace()` to inject a weird machine into a victim process. With `ptrace()` it is trivial to find a `syscall & ret` gadget as it is possible to trap on system calls. We will also assume that we have a complete `sigreturn()` gadget at our disposal. The gadget first loads the sigreturn system call number before it does a system call. This gadget can easily be found by sending the

traced process a signal for which it has registered a handler, prompting the kernel to set up a signal frame with the `sigreturn()` gadget at the top of the stack.

Using our `syscall & ret` and `sigreturn()` gadgets and by faking some signal frames, we can create a chain of system calls. Each frame setting up the registers to do a certain system call while pointing the stack pointer at the next frame, each with a `sigreturn()` gadget at the top. Just like with ROP, it is in a sense the stack pointer which acts as an instruction pointer, only now the stack pointer always points at a complete user context state. While all we do is execute a number of system calls (possibly in a loop) it is surprisingly simple to create complex behavior.

To demonstrate this, we have created a backdoor automaton that waits for a given file to be accessed. This file could for example be an obscure file on a publicly accessible web server. When this file is read, a listener TCP socket is created and if someone connects to this socket it spawns a shell connected to this socket. If no-one connects within 5 seconds, the listener stops listening until the trigger file is accessed again.

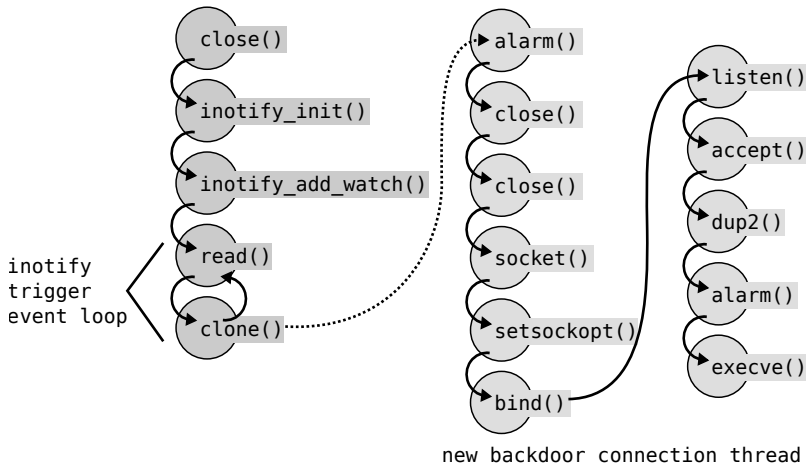
The assumption is that a remote party can easily cause the system to do a read operation on a file that otherwise is rarely read, for example a hidden file in a web document root. Only after this file is accessed, it will be possible to make a connection to the machine through a socket, something that otherwise would be very easy to spot.

To construct our backdoor, we chain together a string of signal frames executing system calls, relying on system call blocking semantics for our logic, as shown on the left-hand-side of Figure 2.6.

In particular, we make use of the `inotify` API. The `inotify` API provides a mechanism for monitoring file system events and allows one to detect accesses to individual files or monitor directories. When a directory is monitored, `inotify` will return events for the directory itself, and for files inside the directory. To determine what events have occurred, an application reads from the `inotify` file descriptor. If no events have occurred, the `read` will block (until at least one event occurs). The API allows us to wait for many events: reads, writes, closes, changes in attributes, etc. For our backdoor, we will wait for any read to the obscure file, but we can easily wait for other events. Thus, to wait for a file being accessed, the `inotify` API gives us a file descriptor with the ability to do a blocking read which returns when a file is read. This serves as our trigger.

When the `read` returns, we know that someone has accessed the file, but we cannot be entirely sure that it was our trigger or an unrelated event. To find out, we will spawn a thread that waits for a remote party (the backdoor master) to connect to our socket. If nobody connects, we assume that the file access is unrelated, close the socket and go back to monitoring file accesses. If there is a connection, we spawn a shell.

To accomplish this, the SROP system call chain of our backdoor follows the blocking read of the `inotify` file descriptor with a `clone()` system call (last system call in the leftmost column in Figure 2.6). This system call has a useful property:



**Figure 2.6:** Our example backdoor automaton. The main backdoor thread sets up an `inotify` watch list. When an attacker causes a file to be accessed, a child thread spawns, allowing the attacker to connect to a backdoor shell.

using `clone()`, it is possible to assign a different stack to the child process, pointing it to a different state in our automaton. Thus, while the parent resumes waiting for the trigger file, the child will be responsible for the backdoor connection. The actions of the child are shown in the two remaining columns of Figure 2.6. The child, sets up an `alarm()` clock and listens on a socket, blocking on `accept()`. If no-one connects, this process will be killed by the alarm and we resume monitoring the file accesses of our trigger file. If someone does connect, the alarm is reset and through a series of further system calls, a shell is spawned.

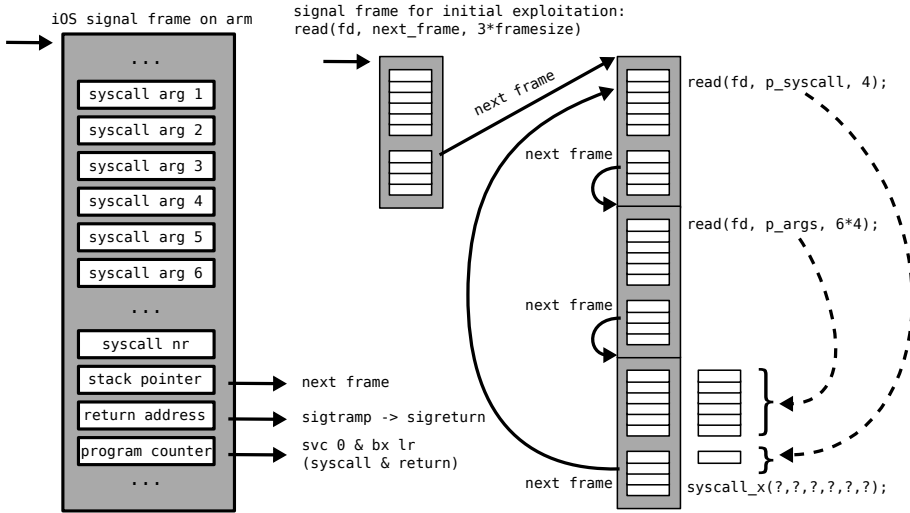
We have implemented the backdoor and tested it on several Linux distributions including the ones mentioned earlier, but also 32 bit variants. As there is no shellcode in memory, the backdoor is very hard to find for a security scanner.

## 2.8 SROP to circumvent code signing

While on Linux it is generally possible to use ROP or SROP to bootstrap more traditional shellcode, other systems like the Apple iPhone's iOS allow only signed code to run natively. Being able to run unsigned code on those systems has become a goal in itself. The jailbreaking community usually uses kernel vulnerabilities to disable the checks that verify these signatures. A well tested method to get control over the kernel is to exploit vulnerable system call interfaces. This, however, does mean that these exploits themselves have to be executed from processes running signed code.

In this section we will describe a technique for a system call proxy using SROP. The technique provides a very generic method of delivering kernel exploits from a





**Figure 2.7:** A syscall proxy controllable over a pipe/file/network socket. An initial signal frame sets up a read system call to read in a cyclic automaton which alternates between reading a system call number and its arguments from a socket, and executing said system call.

process running signed code.

## 2.8.1 Sigreturn on iOS

As a system call proxy is particularly useful for systems requiring signed code, we have implemented this on iOS. When it comes to signaling, iOS (just like Mac OS X) has a small trampoline function from which they indirectly call a signal handler. Upon returning from this signal handler, the trampoline loads the signal frame address from the stack into `r0` (the first system call argument) and then calls `sigreturn`. In iOS we can therefore immediately identify three useful gadgets from the signal handling code:

1. Sigreturn with the signal frame pointer loaded from the stack.
2. Sigreturn with the signal frame pointer in the first function argument.
3. Syscall and return gadget `svc 0 ; bx lr`

The second gadget is useful in case of a function pointer override. We will be using the first and the third gadget for our system call proxy. It is good to note that, unlike on Linux, the iOS/Mac OSX signal frame always contains pointers, so blindly executing signal frames without knowing the location of any data is hard.

### 2.8.2 A system call proxy

The goal of a system call proxy is to remotely control the system calls executed by a process. In our system call proxy automaton (Figure 2.7) we follow the same basic pattern as described in Section 2.7 of chaining system calls. To bootstrap the automaton, an initial signal frame relocates the stack and issues a `read` system call on a file descriptor controlled by the attacker, this could for example be a network socket, or a file. This `read` loads the automaton onto the new location of the stack. The automaton itself is a loop of one or more signal frames doing `read` system calls and one signal frame that will execute an arbitrary system call. The `read` system calls are responsible for filling in the system call number and their correct arguments in the last signal frame, allowing the attacker to simply supply the whole system call over a socket. System calls that use pointers to data structures as arguments could be preceded with calls to `mmap` and `read`.

## 2.9 The Linux system call interface makes SROP Turing complete

While a simple automaton chaining together system calls using `sigreturn` is enough for a backdoor to do its job, attackers may want to encode more complex logic such as obfuscation.

It is clear that if we keep the contents of the signal frames in our automaton the same during execution, that the best we can do is execute a static set of system calls, possibly in a loop. But this changes when we allow our automaton to write back into its own signal frames, changing computations to come. In fact, using an automaton which modifies function arguments and stack pointers in future stack frames, we can construct an interpreter for a Turing-complete language.

Our language has a direct mapping to “brainfuck”, a well-known Turing-complete language [92]. Conceptually, our language makes use of three registers, the program counter `PC`, the memory pointer `P` and a temporary register used for 8-bit addition `A`. These registers are modeled as file descriptors. The file they have open is `/proc/self/mem`, which on Linux is a way of reading and writing to your own address space. Adding and subtracting to and from these registers is done using `lseek`. The `PC` file descriptor points to our interpreted language program in memory. Its instructions are addresses of signal frames which implement the following operations:

1. Jump (followed by an offset used by a relative `lseek` to move the `PC`).
2. Pointer addition/subtraction (followed by an offset used by a relative `lseek` to move `P`
3. 8-bit Addition/subtraction (followed by a constant to add to the byte located at `P`

4. Conditional jump (the same as Jump, except that it only happens if the byte at P is zero).
5. Getchar of the byte at P
6. Putchar of the byte at P
7. Exit

Figure 2.8 shows the control flow diagram of the entire state machine. Structurally, it is shaped like a dispatcher, capable of executing the language's commands. Compared to the original brainfuck, our language is a little richer. Instead of increment and decrement, we offer a more generic add of 8 bit numbers.

Instruction dispatch happens by doing a read on PC, storing the value in the stack pointer of a dispatch signal frame. When the automaton then proceeds to a do sigreturn, it will jump to the signal frame belonging to the instruction it has just read.

Reading the immediates following the instructions is also done simply by reading data from PC. For pointer addition and for jumps, we use `lseek` with a `SEEK_CUR` argument to move P and PC respectively.

Conditional jumps are implemented the same way as a normal jump, with the exception that the byte value at P is first read into the high byte of the file descriptor argument for the `lseek` on PC. If the value at P is not 0, the `lseek` will be done not on PC, but on a very high, non-existing file descriptor, causing `lseek` to fail instead of seek, therefore making the jump conditional.

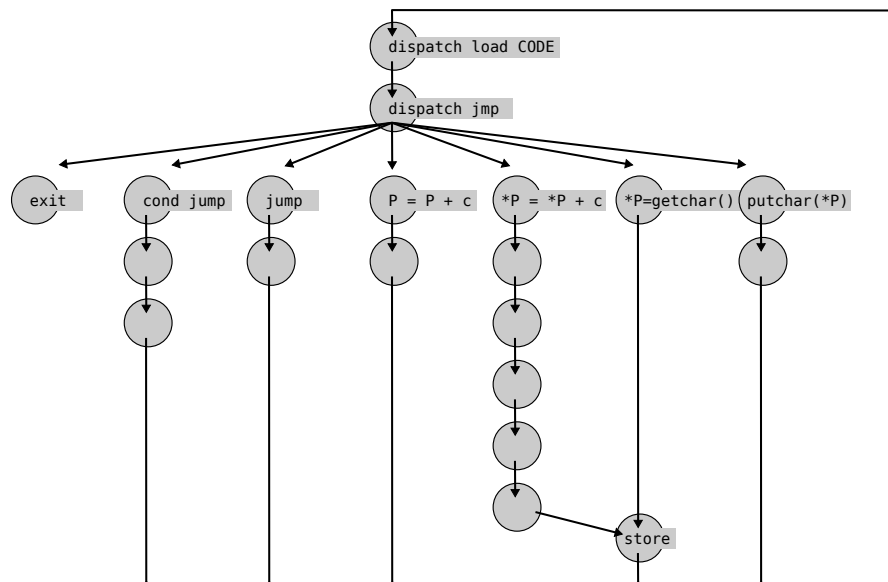
The most complex operation turns out to be our 8 bit addition of data in memory. For this we use a 512 byte buffer, each byte filled with the modulo 256 of its index. For an addition we do an absolute seek with our A register to the start of the buffer, followed by 2 relative seeks given by the current value at P and the current instruction's immediate, the result of the addition can now be read from A.

We have implemented an emulator for [92] which first translates it to our machine language and then proceeds to run it on our automaton.

## 2.10 Mitigation

When implementing exploit mitigations, one has to consider the trade-off between performance loss and security gain.

We recognize that sigreturn oriented programming by itself is not an exploitable vulnerability. Similar to ROP, it is an exploitation method that can be used in the event of a vulnerability. Often, ROP may be used instead of SROP and vice versa. Also, some of the 'universal SROP' variants we discovered in Linux on x86-64 and on ARM have been mitigated in recent Linux kernels. On x86-64, the default configuration now uses `vsyscall` emulation to eliminate useful static gadgets and on ARM, sigreturn has been removed from the static vectors page. However, this does not stop sigreturn oriented programming from being used as a generic stealthy



**Figure 2.8:** Our Turing-complete interpreter

backdoor. Also, if a `syscall & return` gadget can be found inside the binary, using a generic SROP exploit is still easier than using a binary specific ROP chain. We see SROP as a low hanging fruit for exploit writers, worthy of mitigation.

A possible approach to eliminate sigreturn oriented programming as a viable exploitation method would be to embed a kernel-supplied secret value in the sigreturn frame. Upon returning from a signal, the kernel would check this secret value against the value it had written earlier. If the value is different, the kernel can opt to let the process crash. This method is very similar to stack canaries which have been widely adopted to protect against stack buffer overflows. It also suffers from the same weakness: If an attacker can leak this secret value, he or she can use that to forge fake signal frames. This risk could possibly be remedied by making the kernel zero out the secret value during the sigreturn system call, so that the value is only present in user space while the signal handler is running. The signal canary could even be a cryptographic message authentication code on the complete signal frame, to prevent arbitrary modifications, even in the event of a memory leak.

A complimentary solution could be to keep a counter per process in kernel space which keeps track of the number of signal handlers currently executing. Upon signal delivery, the counter is increased, while a sigreturn decreases the counter. If the counter becomes negative, the process is killed. While this should work fine with single threaded processes, there could be complications with this scheme in multi-threaded programs. Keeping a counter per thread might break programs that use lightweight threads, which may switch user space contexts between threads. This may cause a signal to be delivered in one thread and return in another. On the other

hand, keeping a counter for the whole thread group could lead to an overestimation of the number of delivered signals when one of the threads does a `fork()`, unsharing the address space with the other threads. This seems to us as the lesser of two evils.

Ultimately, there's the question whether we can change the behavior of `sigreturn` at all without breaking user space applications. Kernels are supposed to have a stable ABI and for that reason it is not done to change the behavior of system calls. User space programs might break when they depend on previous behavior. However, `sigreturn` seems to be a special case. The only legitimate way of calling `sigreturn` seems to be when user space has been set up by the kernel to call it. Also, depending on the CPU features the signal frame may differ, as for example, SSE and AVX registers will be saved on platforms that support these instruction sets. So, while the location of the general purpose registers seems to be pretty stable and accessible from within a signal handler, being able to manually create a stack frame to return to should in our opinion not be considered part of the ABI.

All things considered, we strongly feel that mitigation against `sigreturn` oriented programming as an exploitation method is needed.

## 2.11 Conclusion

In this chapter, we have discussed `sigreturn` oriented programming, a novel, Turing complete technique for programming a novel type of weird machine. `Sigreturn` oriented programming is a generic technique, as we demonstrated by using it for an exploit, a backdoor, and a code-signing bypass. Moreover, it works on a wide variety of operating systems and different architectures. For several of these systems, `sigreturn` oriented programming permits exploitation without any precise knowledge about the executable. Moreover, the exploit is reusable, as it does not depend much on the victim process at all.

`Sigreturn` oriented programming represents a convenient, portable technique to program arbitrary code even in strongly protected machines. The number of gadgets needed is minimal and in many systems those gadgets are in a fixed location. As such the technique ranks among the lowest hanging fruit currently available to attackers on UNIX systems. It is important to emphasize that even if kernels are patched to eliminate these fixed-location gadgets, the usefulness for backdoors is undiminished. In summary, we believe that `sigreturn` oriented programming is a powerful addition to the attackers' arsenal.

## Acknowledgments

We thank the anonymous reviewers for their excellent feedback. This work was supported by the ERC StG project "Rosetta" and by the EU FP7 "SYSSEC" project.

## Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector

### Abstract

Memory deduplication, a well-known technique to reduce the memory footprint *across* virtual machines, is now also a default-on feature *inside* the Windows 8.1 and Windows 10 operating systems. Deduplication maps multiple identical copies of a physical page onto a single shared copy with copy-on-write semantics. As a result, a write to such a shared page triggers a page fault and is thus measurably slower than a write to a normal page. Prior work has shown that an attacker able to craft pages on the target system can use this timing difference as a simple single-bit side channel to discover that certain pages exist in the system.

In this chapter, we demonstrate that the deduplication side channel is much more powerful than previously assumed, potentially providing an attacker with a *weird machine* to read arbitrary data in the system. We first show that an attacker controlling the alignment and reuse of data in memory is able to perform byte-by-byte disclosure of sensitive data (such as randomized 64 bit pointers). Next, even without control over data alignment or reuse, we show that an attacker can still disclose high-entropy randomized pointers using a *birthday attack*. To show these primitives are practical, we present an end-to-end JavaScript-based attack against the new Microsoft Edge browser, in absence of software bugs and with all defenses turned on. Our attack combines our deduplication-based primitives with a reliable Rowhammer exploit to gain arbitrary memory read and write access in the browser.

We conclude by extending our JavaScript-based attack to cross-process system-wide exploitation (using the popular nginx web server as an example) and discussing mitigation strategies.

### 3.1 Introduction

Memory deduplication is a popular technique to reduce the memory footprint of a running system by merging memory pages with the same contents. Until recently, its primary use was in virtualization solutions, allowing providers to host more virtual machines with the same amount of physical memory [103; 118; 9]. The last five years, however, have witnessed an increasingly widespread use of memory deduplication, with Windows 8.1 (and later versions) adopting it as a default feature *inside* the operating system itself [86].

After identifying a set of identical pages across one or more processes, the deduplication system creates a single read-only copy to be shared by all the processes in the group. The processes can freely perform read operations on the shared page, but any memory write results in a (copy-on-write) page fault creating a private copy for the writing process. Such write operation takes significantly longer than a write into a non-deduplicated page. This provides an attacker able to craft pages on the system with a single-bit timing side channel to identify whether a page with given content exists in the system. When using this simple side channel for information disclosure, the memory requirements grow exponentially with the number of target bits in a page, resulting in a very slow primitive which prior work argued useful only to leak low-entropy information [11].

In this chapter, we show that memory deduplication can actually provide much stronger attack primitives than previously assumed, enabling an attacker to potentially disclose arbitrary data from memory. In particular, we show that deduplication-based primitives allow an attacker to leak even high-entropy sensitive data such as randomized 64 bit pointers and start off advanced exploitation campaigns. To substantiate our claims, we show that a JavaScript-enabled attacker can use our primitives to craft a reliable exploit based on the widespread Rowhammer hardware vulnerability [69]. We show that our exploit can allow an attacker to gain arbitrary memory read/write access and “own”<sup>1</sup> a modern Microsoft Edge browser, even when the target browser is entirely free of bugs with all its defenses are turned on.

All our primitives exploit the key intuition that, if an attacker has some degree of control over the memory layout, she can dramatically amplify the strength of the memory deduplication side channel and reduce its memory requirements. In particular, we first show how control over the alignment of data in memory allows an attacker to pad sensitive information with known content. We use this padding primitive in conjunction with memory deduplication to perform byte-by-byte disclosure of high-entropy sensitive data such as *randomized code pointers*. We then extend this attack to situations where the target sensitive information has strong alignment properties. We show that, when memory is predictably reused (e.g., when using a locality-friendly memory allocator), an attacker can still perform byte-by-byte disclosure via partial data overwrites. Finally, we show that, even when entropy-

---

<sup>1</sup>To divert, and gain control over its execution.

reducing primitives based on controlled memory alignment or reuse are not viable, an attacker who can lure the target process into creating many specially crafted and interlinked pages can still rely on a sophisticated *birthday attack* to reduce the entropy and disclose high-entropy data such as *randomized heap pointers*.

After showcasing our deduplication-based primitives in a (Microsoft Edge) browser setting, we generalize our attacks to system-wide exploitation. We show that JavaScript-enabled attackers can break out of the browser sandbox and use our primitives on any other independent process (e.g., network service) running on the same system. As an example, we use our primitives to leak *HTTP password hashes* and break *heap ASLR* for the popular *nginx* web server.

To conclude our analysis, we present a Microsoft Edge case study which suggests that limiting the deduplication system to only *zero pages* can retain significant memory saving benefits, while hindering the attacker's ability to use our primitives for exploitation purposes in practice.

Summarizing, we make the following contributions:

- We describe novel memory deduplication-based primitives to create a programming abstraction (or *weird machine* [42; 127]) that can be used by an attacker to disclose sensitive data and start off powerful attacks on a target deduplication-enabled system (Section 3.3).
- We describe an implementation of our memory deduplication-based primitives in JavaScript and evaluate their properties on the Microsoft Edge browser running on Windows 10 (Section 3.4 and Section 3.5).
- We employ our memory deduplication-based primitives to craft the first reliable Rowhammer exploit for the Microsoft Edge browser from JavaScript (Section 3.6).
- We show how our primitives can be extended to system-wide exploitation by exemplifying a JavaScript-based cross-process attack on the popular *nginx* web server running next to the browser (Section 3.7).
- We present a mitigation strategy (*zero-page deduplication*) that preserves substantial benefits of full memory deduplication (>80% in our case study) without making it programmable by an attacker (Section 3.8).

## 3.2 Background

We first discuss the basic idea behind memory deduplication and its implementation on Windows and Linux. We then describe the traditional memory deduplication side channel explored in prior work and its limitations.

### 3.2.1 Memory Deduplication

To reduce the total memory footprint of a running system, memory pages with the same contents can be shared across independent processes. A well-known example



of this optimization is the page cache in modern operating systems. The page cache stores a single cached copy of file system contents in memory and shares the copy across different processes. Memory deduplication generalizes this idea to the run-time memory footprint of running processes. Unlike the page cache, two or more pages with the same content are always deduplicated, even, in fact, if the pages are completely unrelated and their equivalence is fortuitous.

To keep a single copy of a number of identical pages, a memory deduplication system needs to perform three tasks:

1. Detect memory pages with the same content. This is usually done at regular and predetermined intervals during normal system operations [66; 86].
2. After detecting pages with the same content, keep only one physical copy and return the others to the memory allocator. For this purpose, the deduplication system updates the page-table entries (PTE) of the owning processes so that the virtual addresses (originally pointing to different pages with the same content) now point to a single shared copy. The PTEs are also marked as read-only to support copy-on-write (COW) semantics.
3. Create a private copy of the shared page whenever any process writes to it. Specifically, once one of owning processes writes to the read-only page, a (COW) page fault occurs. At this point, the memory deduplication system can create a private copy of the page and map it into the corresponding PTE of the faulting process.

On Windows (8.1 onward), memory deduplication is known as *memory combining*. The implementation merges pages that are both *private* and *pageable* [87] regardless of their permission bits. These pages exclude, for example, file-backed pages or huge pages which are non-pageable on Windows. To perform deduplication, memory combining relies on a kernel thread to scan the entire physical memory for pages with identical content. Every 15 minutes (by default), the thread calls the `MiCombineAllPhysicalMemory` function to merge all the identical memory pages found. On Linux, memory deduplication is known as *kernel same-page merging* (*KSM*). The implementation operates differently compared to Windows, combining both scanning and merging operations in periodic and incremental passes over physical memory [9].

### 3.2.2 The Memory Deduplication Side Channel

As mentioned earlier, writing to a shared page from any of the owning processes results in a page fault and a subsequent page copy. Due to these additional (expensive) operations, a write to a shared page takes significantly longer (up to one order of magnitude) compared to a write to a regular page.

This timing difference provides an attacker with a side channel to detect whether a given page exists in the system. For this purpose, she can craft a page with the exact same content, wait for some time, and then measure the time to perform a

write operation to the crafted page. If the write takes longer than a write to a non-deduplicated page (e.g., a page with random content), the attacker concludes that a page with the same content exists. Using this capability, the attacker may be able to detect a user visiting a particular web page or running a particular program. We note that, while false positives here are possible (e.g., due to a non-unique crafted page or noisy events causing regular write operations to complete in unexpectedly long time), an attacker can use redundancy (e.g., repeated attempts or multiple crafted pages) to disclose the intended information in a reliable way.

At first glance, memory deduplication seems like a very slow single-bit side channel that can only be used for fingerprinting applications [97; 123] or at most leaking a limited number of bits from a victim process [11]. In the next section, we describe how memory deduplication can be abused to provide an attacker with much stronger primitives.

### 3.3 Attack Primitives

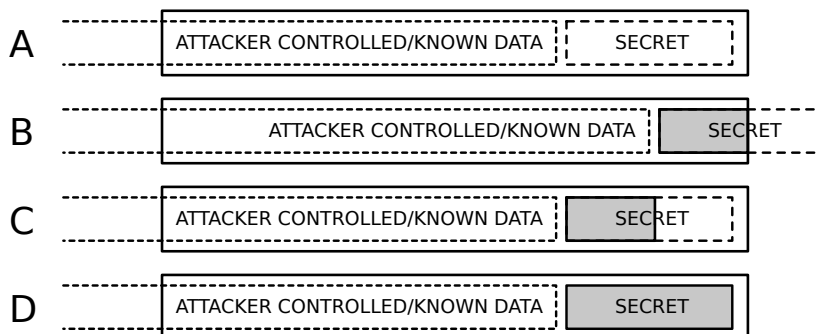
We describe efficient primitives based on the memory deduplication side channel to read *high-entropy* data from memory. Our primitives abuse a given deduplication system to build a weird machine [42; 127] that we can program by controlling the memory layout and generating pages with appropriate content. We later show that, by relying on our primitives, an attacker can program the weird machine to leak sensitive information such as randomized 64 bit pointers or even much larger secrets (e.g., 30 byte password hashes).

A naive strategy to disclose arbitrarily large secret information using the single-bit memory deduplication side channel is to brute force the space of all possible secret page instances. Brute forcing, however, requires the target page to be aligned in memory and imposes memory requirements which increase exponentially with each additional secret bit. This makes brute forcing high-entropy data not just extremely time consuming, but also unreliable due to the increasing possibility of false positives [11]. A more elegant solution is to disclose the target secret information incrementally or to rely on generative approaches. This intuition forms the basis for our memory deduplication-based primitives.

#### Primitive #1: Alignment probing

We craft a primitive that allows an attacker to perform byte-by-byte probing of secret information by controlling its *alignment*. Figure 3.1-A exemplifies a memory page with the secret data targeted by the attacker. We refer to pages that contain secret data as *secret pages* and to the pages that the attacker crafts to disclose the secret data as *probe pages*.

This primitive is applicable when attacker-controlled input can change the alignment of secret data with *weak alignment properties*. For instance, the secret may



**Figure 3.1:** The *alignment probing* primitive to leak high-entropy secrets with weak memory alignment properties.

be a password stored in memory immediately after a blob of attacker-provided input bytes. By providing fewer or more bytes, the attacker can shift the password up and down in memory.

Using this capability, the attacker pushes the second part of the secret out of the target page (Figure 3.1-B), allowing her to brute force, using deduplication, only the first part of the secret (e.g., one byte) with much lower entropy. After obtaining the first part of the secret, the attacker provides a smaller input, so that the entire secret is now in one page (Figure 3.1-C). Next, she brute forces only the remainder of the secret to fully disclose the original data (Figure 3.1-D). With a larger secret, the attacker can simply change the alignment multiple times to incrementally disclose the data.

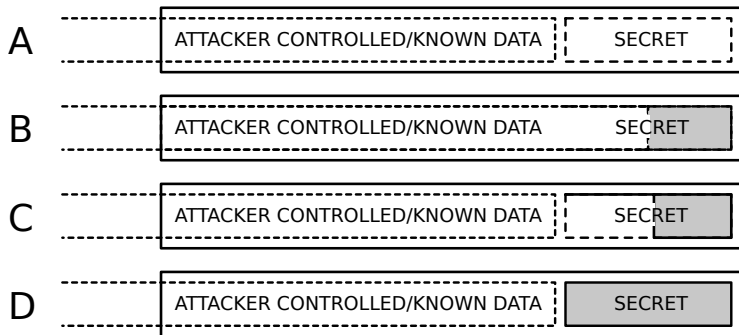
Our *alignment probing* primitive is very effective in practice. We later show how we used it to disclose a code pointer in Microsoft Edge and a password hash in nginx.

## Primitive #2: Partial reuse

When the secret has strong memory alignment properties (e.g., randomized pointers), we cannot use our alignment probing primitive to reduce the entropy to a practical brute-force range. In this scenario, we craft a primitive that allows an attacker to perform byte-by-byte probing of secret information by controlling *partial reuse* patterns. This primitive is applicable when attacker-controlled input can partially overwrite stale secret data with *predictable reuse properties*.

User-space memory allocators encourage memory reuse and do not normally zero out deallocated buffers for performance reasons. This means that a target application often reuses the same memory page and selectively overwrites the content of a reused page with new data. If the application happens to have previously stored the target secret data in that page, the attacker can then overwrite part of the secret with known data and brute force the remainder.

Figure 3.2-A shows an example of a page that previously stored the secret and is



**Figure 3.2:** The *partial reuse* primitive to leak high-entropy secrets with predictable memory reuse properties.

reused to hold attacker-controlled input data. After partially overwriting the first part of the secret with a large input, the attacker can brute force, using deduplication, only the second part (Figure 3.2-B). Given the second part of the secret, the attacker can now brute force the first part by deduplicating against a page without an overwritten secret (Figure 3.2-C). Similar to the previous primitive, the operations generalize to larger secrets and the result is full disclosure of the original data (Figure 3.2-D).

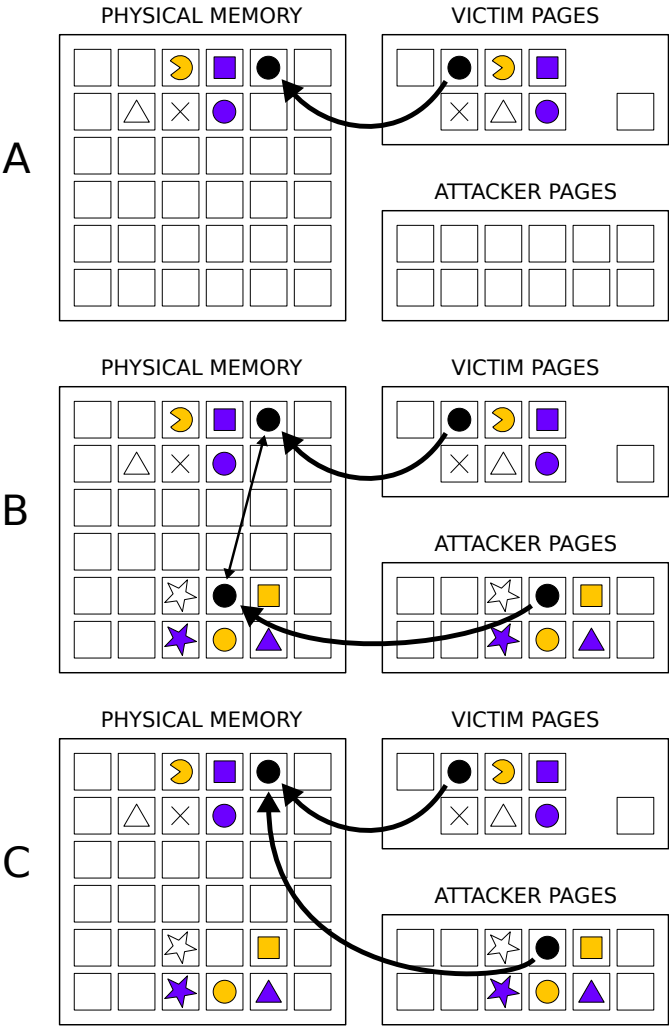
Our *partial reuse* primitive is fairly common in practice. We later show how we used it to break heap ASLR in nginx.

### Primitive #3: Birthday heap spray

Our third primitive can leak a secret even when the attacker has no control over memory alignment or reuse. The primitive relies on a generative approach that revolves around the well-known birthday paradox, which states that the probability of at least two people in a group having the same birthday is high even for modest-sized groups. This primitive is applicable when the attacker can force the application to controllably *spray* target secrets over memory.

So far, we assumed that there is only one secret that we want to leak, so if a (partially masked) secret has  $P$  possible values, we use memory deduplication to perform  $1 \times P$  comparisons between the  $P$  probe pages and the single target page—essentially brute forcing the secret. For a large  $P$ , doing so requires a prohibitively large amount of memory. In addition, it requires a large number of tests, which may lead to many false positives due to noise.

However, if the attacker can cause the target application to generate many secrets, memory deduplication provides a much stronger primitive than simple brute forcing. For instance, an attacker may generate a large number of (secret) heap pointers by creating a large number of objects from JavaScript, each referencing another object. For simplicity, we assume that the object is exactly one page in size and all fields are crafted constant and known except for one secret pointer, but other choices are



**Figure 3.3:** The *birthday heap spray* primitive to leak high-entropy heap ASLR with no attacker-controlled alignment or reuse.

possible. Whatever the page layout, its content serves as an *encoding* of the secret pointer.

Assume the attacker causes the application to generate  $S$  such pages, each with a different secret pointer (Figure 3.3-A). The attacker now also creates  $P$  probe pages, with  $P$  being roughly the same size as  $S$ . Each probe page uses the same encoding as the secret pages, except that, not knowing the secret pointers, the attacker needs to “guess” their values. Each probe page contains a different guessed value. The

idea is to find at least one of the probe pages matching *any* of the secret pages. This is a classic birthday problem with the secret and probe values playing the role of birthdays.

Since memory deduplication compares any page with any other page in each deduplication pass, it automatically tests all our  $P$  possible probe pages against the  $S$  target secret pages (Figure 3.3-B). A hit on any of our  $P$  possible values immediately exposes a target secret (Figure 3.3-C).

Our birthday primitive reduces the memory requirements of the attack by a factor of  $S$ . It is especially useful when leaking the location of randomized pointers. Note that, for exploitation purposes, it is typically not important which pointer the attacker leaks, as long as at least one of them can be leaked. We later show how we used our primitive to leak a randomized heap pointer in Microsoft Edge and subsequently craft a reliable Rowhammer exploit.

## 3.4 Microsoft Edge Internals

We discuss Microsoft Edge internals necessary to understand the attacks presented in Section 3.5. First, we look at object allocation in Microsoft Edge's JavaScript engine, Chakra. We then describe how Chakra's JavaScript arrays of interest are represented natively. With these constructs, we show how an attacker can program memory deduplication from JavaScript. Finally, we describe how an attacker can reduce noise and reliably exploit our primitives.

### 3.4.1 Object Allocation

Chakra employs different allocation strategies for objects of different sizes maintained in three buckets [132]: small, medium, and large object buckets. The small and large object buckets are relevant and we discuss them in the following.

#### Small objects

Objects with a size between 1 and 768 bytes are allocated using a slab allocator. There are different pools for objects of different sizes in increments of 16 bytes. Each pool is four pages (16,384 bytes) in size and maintains contiguously allocated objects. In some cases, Chakra combines multiple, related allocations in one pool. This is the case, for example, for JavaScript arrays with a pre-allocated size of 17 elements or less, where an 88-byte header is allocated along with  $17 \times 8$  bytes of data.

#### Large objects

Unlike small objects, large objects (i.e., larger than 8,192 bytes) are backed by a `HeapAlloc` call and are hence stored in a different memory location than their head-

ers. One important consequence is that the elements of a large object have a known page alignment. As discussed in Section 3.5.3, we rely on this property to create our probe pages.

### 3.4.2 Native Array Representation

Modern JavaScript has two different types of arrays. Regular *Arrays*, which can hold elements of any type and may even be used as a dictionary, and *TypedArrays* [95], which can only hold numerical values of a single type, have a fixed size, and cannot be used as a dictionary. TypedArrays are always backed by an *ArrayBuffer* object, which contiguously stores numerical elements using their native representation. Large ArrayBuffers are page-aligned by construction. To store regular Arrays, Chakra internally relies on several different representations. We focus here on the two representations used in our exploit.

The first representation can only be used for arrays which contain only numbers and are not used as dictionaries. With this representation, all the elements are sequentially encoded as double-precision IEEE754 floating-point numbers. This representation allows an attacker to create fake objects in the data part of the array. In particular, both pointers and small numbers can be encoded as denormalized doubles.

A second representation is used when an array may contain objects, strings, or arrays. For this representation, Microsoft Edge intelligently relies on the fact that double-precision IEEE754 floats have  $2^{52}$  different ways of encoding both  $+NaN$  and  $-NaN$ <sup>2</sup>.  $2^{52}$  is sufficient to encode single values for  $+NaN$  and  $-NaN$ , as well as 48 bit pointers and 32 bit integers. This is done by XORing the binary representation of doubles with `0xfff c000000000000` before storing them in the array. The 12 most significant bits of a double consist of a single sign bit and an 11-bit exponent. If the exponent bits are all ones, the number represents  $+NaN$  or  $-NaN$  (depending on the sign bit). The remaining 52 bits do not matter in JavaScript. As mentioned, Chakra only uses single values for  $+NaN$  and  $-NaN$ , `0x7ff8000000000000` and `0xfff8000000000000` respectively. Since a user-space memory address has at least its 17 most significant bits set to 0, no double value overlaps with pointers by construction and Chakra can distinguish between the two cases without maintaining explicit type information.

The JIT compiler determines which representation to use based on heuristics. If the heuristics decide incorrectly (e.g., a string is later inserted into an array which can only contain doubles), the representation is changed in-place.

In Section 3.5.3, we show how we used these representations to craft our birthday heap spray primitive. Further, we used the details of the second representation to improve the success rate of our Rowhammer attack in Section 3.6.

---

<sup>2</sup>Not a Number or NaN represents undefined values.

### 3.4.3 Programming Memory Deduplication from JavaScript

To program memory deduplication, we need to be able to (a) create arbitrary pages in memory and (b) identify memory pages that have been successfully deduplicated. We use TypedArrays to create arbitrary memory pages and an available high-resolution timer in JavaScript to measure slow writes to deduplicated pages.

#### Crafting arbitrary memory pages

As mentioned in Section 3.4.2, TypedArrays can store native data types in memory. If the TypedArray is large enough, then the array is page-aligned and the location of each element in the page is known. Using, for example, a large Uint8Array, we can control the content of each byte at each offset within a memory page, allowing us to craft arbitrary memory pages.

#### Detecting deduplicated pages

While JavaScript provides no access to native timestamp counters via the RDTSC instruction, we can still rely on JavaScript's `performance.now()` to gather timing measurements with a resolution of hundreds of nanoseconds.

We detect a deduplicated page by measuring lengthy COW page faults when writing to the page. We measured that writing to a deduplicated page takes around four times longer than a regular page—including calls to `performance.now()`. This timing difference in conjunction with the ability to craft arbitrary memory pages provides us with a robust side channel in Microsoft Edge.

#### Detecting deduplication passes

As previously discussed in Section 3.2, Windows calls `MiCombineAll-PhysicalMemory` every 15 minutes to deduplicate pages. To detect when a deduplication pass occurs, we create pairs of pages with unique content, and write to pages belonging to different pairs every 10 seconds. Once a number of writes take considerably longer than a moving average, we conclude that a deduplication pass has occurred.

### 3.4.4 Dealing with Noise

To minimize the noise during our measurements, we used a number of techniques that we briefly describe here.

The first technique is to avoid cold caches. We first read from the address on which we are going to perform a write to ensure the page has not been swapped out to stable storage. Further, we call `performance.now()` a few times before doing the actual measurements in order to ensure its code is present in the CPU cache.

The second technique is to avoid interference from the garbage collector (GC). We try to trigger the GC before doing any measurements. This can be done on most browsers by allocating and freeing small chunks of memory and detecting a sudden



slowdown during allocations. On Microsoft Edge, however, it is possible to make a call to `window.CollectGarbage()` to directly invoke the GC.

The third technique is to avoid interference from CPU's dynamic frequency scaling (DFS). We try to minimize noise from DFS by keeping the CPU in a busy loop for a few hundred milliseconds and ensuring that it is operating at the maximum frequency during our measurements.

Equipped with reliable timing and memory management capabilities in JavaScript, we now move on to the implementation of our primitives and our Rowhammer attack.

## 3.5 Implementation

We now discuss the implementation details of the memory deduplication-based primitives introduced in Section 3.3. Our implementation is written entirely in JavaScript and evaluated on Microsoft Edge running on Windows 10. We chose Microsoft Edge as our target platform since it is a modern browser that is designed from the ground up with security in mind. At the time of writing, Microsoft Edge is the only browser on Windows that ships as a complete 64 bit executable by default. 64 bit executables on Windows benefit from additional ASLR entropy compared to their 32 bit counterparts [90]. Nonetheless, we now show that using memory deduplication, we can leak pointers into the heap as well as pointers into interesting code regions.

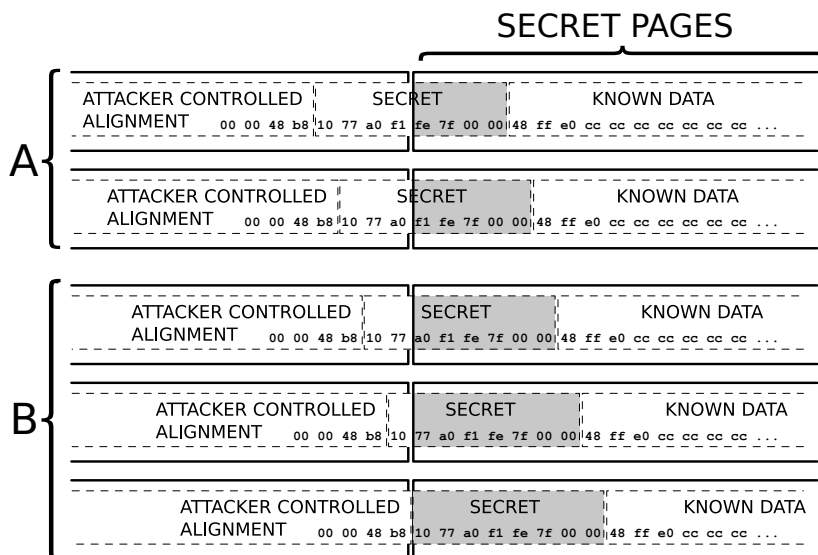
Before detailing our end-to-end implementation, we first describe the testbed we used to develop our attacks.

### 3.5.1 Testbed

We used a PC with an MSI Z87-G43 motherboard, an Intel Core i7-4770 CPU, and 8 GB of DDR3 RAM clocked at 1600MHz running Windows 10.0.10240.

### 3.5.2 Leaking Code Pointers in Edge

We used our alignment probing primitive to leak code pointers in Microsoft Edge. Like all modern browsers, Microsoft Edge employs a JIT compiler which compiles JavaScript to native code. The generated code is full of references to memory locations (i.e., both heap pointers and code pointers) and, since x86 opcodes vary in size, most of these pointers turn out to be unaligned. This is ideal for our alignment probing primitive. We can first craft a large JavaScript routine mostly filled with known instructions that do not reference any pointers. Then, right in the middle, we can cause the generation of an instruction that contains a single pointer, and surgically position this pointer right across two page boundaries. We can then incrementally leak parts of the pointer across multiple deduplication passes. Although, in principle, this strategy sounds simple, complications arise when we account for security defenses deployed by modern browsers.



**Figure 3.4:** The incremental disclosure of a code pointer through JIT code. In the first deduplication pass, we can leak the higher bits of a randomized code pointer (A) and, in the second deduplication pass, we can leak the lower bits (B).

Microsoft Edge and other browsers randomize the JIT code they generate as a mitigation against return-oriented programming (ROP). Otherwise, a deterministic code generation scheme would allow attackers to generate their own ROP gadgets at a known offset in the JIT code. Randomization techniques include using XOR to mask constant immediates with a random value and insertion of dummy opcodes in the middle of JIT code. These lead to the presence of significant randomness to leak pointers in the middle of a JavaScript function with the approach described earlier. Fortunately, randomization does not affect the end of the JIT code generated for a given JavaScript function.

At the very end of each compiled JavaScript routine, we can find some exception handling code. The last two instructions of this code load a code pointer into the RAX register and jump there. The remainder of the page is filled with `int 3` instructions (i.e., `0xcc` opcode). The code pointer always points to the same code address in `chakra.dll` and can therefore reveal the base address of this DLL. For this to work, we need to make the JIT compiler create a routine which is slightly larger than one page in size. We could then push the code address partially across this page boundary, and create a second page where all data is known except for this partial code pointer. By pushing this code pointer further over the page boundary, we can expose more and more entropy to the second page as shown in Figure 3.4. This provides us with the semantics we need for our alignment probing primitive to work.

The only problem we still need to overcome is that we do not fully control the size of the routine due to the random insertion of dummy instructions. Given that

the entropy of dummy instruction insertion is relatively low for practical reasons, we solved this by simply making the compiler generate JIT code for a few hundred identical routines. This results in a few pages for each possible alignment of our code pointer across the page boundary. At least one of them will inevitably be the desired alignment to probe for.

### Dealing with noise

Although we have a very small number of probe pages, false positives may still occur. A simple way to sift out false positives when probing for secret pages with  $k$  unknown bytes, is to probe for pages with fewer (e.g.,  $k - 1$ ) unknown bytes as well. Recall that we spray the JIT area with the same function, but due to the introduced randomness by the JIT compiler, different (last) pages end up with different number of bytes from the pointers. Hence, in the final pass we can probe for secret pages which contain six, seven, and eight bytes all at once. Since the correct guesses contain the same information, they are redundant and can thus be used to verify each other using a simple voting scheme. Figure 3.4 shows how we exploit this redundancy to reduce the noise in the first and the second deduplication pass (A and B, respectively).

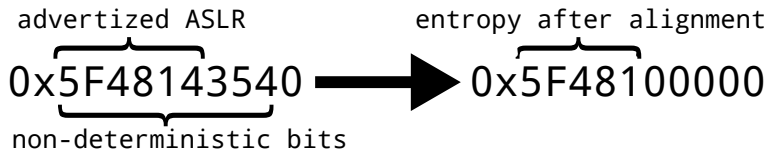
### Time and memory requirements

ASLR entropy for 64 bit DLLs on Windows 10 is 19 bits. DLL mappings may start from `0x7ff800000000` to `0x7fffffff0000`. Assuming we know the exact version of `chakra.dll` (which allows us to predict the 16 least significant bits), we can easily leak the pointer's location in two deduplication passes. In the first pass, we can leak the five most significant bytes, as shown in Figure 3.4-A. Out of these bytes only the 11 least significant bits are unknown, hence, we only need  $2^{11}$  probe pages requiring 8 MB of memory. In a second pass (Figure 3.4-B), we can probe for the remaining eight bits of entropy. Note that the memory requirement of this attack is orders of magnitude smaller than a sheer brute force, making this attack feasible in a browser setting.

In case the exact version of `chakra.dll` is unknown, we can opt to leak the pointer in three passes by leaking the two least significant bytes in the last pass. Assuming the code pointer is aligned on a 16 byte boundary, this requires  $2^{12}$  probe pages and 16 MB of memory.

### 3.5.3 Leaking Heap Pointers in Edge

While we could incrementally leak randomized code pointers using our alignment probing primitive in Microsoft Edge, we did not find a similar scenario to leak randomized heap pointers given their strong alignment properties. To leak heap pointers, using our partial reuse primitive is an option, but given the strong security de-



**Figure 3.5:** Entropy of an arbitrary randomized heap pointer before and after using the timing side channel.

fenses against use-after-free vulnerabilities deployed in modern browsers, memory reuse is not easily predictable. Hence, a different strategy is preferable.

Since we have the ability to generate many heap pointers from JavaScript, we can craft our birthday heap spray primitive (Section 3.3) to leak heap pointers. For this attack, we allocate many small heap objects and try to leak the locations of some of them. Before employing our birthday heap spray primitive to break heap ASLR, we first describe an additional novel timing side channel to further reduce the entropy in a preliminary step.

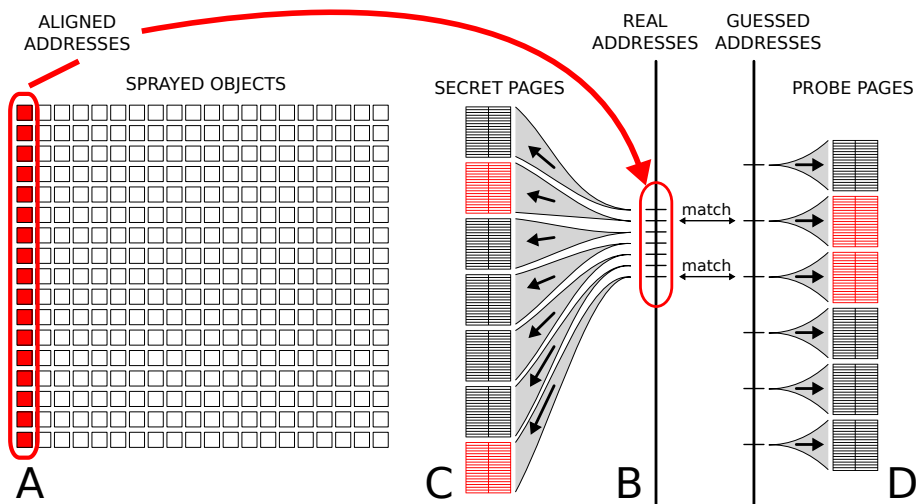
### Reducing ASLR entropy

At the time of writing, 64 bit heap memory on Windows 10 starts at addresses ranging between `0x100000000` and `0x10000000000`, and aligned at 16-page boundaries. This leaves us with 24 bits of entropy for the heap, similar to what prior work previously reported for Windows 8 [90]. Most pointers used from JavaScript, however, do not align at 16-page boundaries and can point to any 16 byte-aligned location after their base offset (see Figure 3.5 for an example). This leaves us with 36 bits of entropy for randomized heap addresses.

Mounting our birthday heap spray primitive directly on 36 bits of entropy requires  $2^{18}$  secret pages and the same number of probe pages, amounting to 1 GB of memory. Additionally, finding a signal in  $2^{18}$  pages requires a very small false positive rate, which past research shows is difficult to achieve in practice [11]. However, using a timing side channel in Microsoft Edge’s memory allocator, we can reliably detect objects that are aligned to 1 MB, reducing the entropy down to 20 bits.

Whenever Microsoft Edge’s memory allocator runs out of memory, it needs to ask the operating system for new pages. Every time this happens, the allocator asks for 1 MB of memory. These allocations happen to also be aligned at the 1 MB boundary. In order to get many consecutive allocations, we spray many small array objects (see Section 3.4.1) on the heap in a tight loop and keep a reference to them so that they are not garbage collected. We time each of these allocations and mark the ones that take longer than eight times the average to complete. If, for example, it takes 4,992 objects to fill a 1 MB slab buffer, we try to find chains of slower allocations that are 4,992 allocations apart. These are the array objects aligned at the 1 MB boundary.

There may still be a small number of false positives in our (large) set of can-



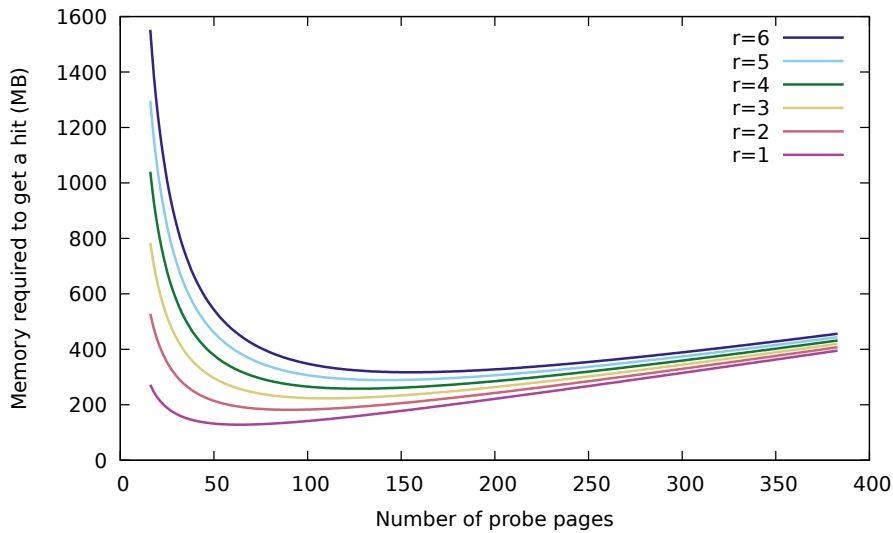
**Figure 3.6:** The *birthday heap spray* primitive to leak high-entropy heap ASLR with no attacker-controlled alignment or reuse. After finding the alignment of the sprayed heap objects via a side-channel (A), we encode the address of each 1 MB-aligned object (B) into a secret page by storing its reference multiple times (C). We then guess these secret pages by creating probe pages that mimic the layout of secret pages (D). In our current version, our guesses are 128 MB apart.

didates. This is not an issue, as these candidates will simply not match any of the probe pages that we craft for our birthday heap spray primitive.

### Birthday heap spray

Figure 3.6 summarizes the steps we followed to implement our birthday heap spray primitive. As a first step, we allocate a number of target objects aligned at the 1 MB boundary using the timing side channel introduced earlier (Figure 3.6-A). The *reference* to a 1 MB-aligned object constitutes a secret that we want to leak using our birthday heap spray primitive. Since we can create an arbitrarily large number of such objects, we can force Microsoft Edge to generate  $S$  secrets for our birthday attack.

We encode each of the  $S$  secrets in a single secret page which we describe now. Our  $S$  secret pages are backed using a large Array, as described in Section 3.4.1. We fill in the array elements (at known offsets) with references to exactly one secret per page. Using this strategy, we can force each secret page to store 512 identical 64 bit pointers. Note that probing directly for a target object’s header (without using our secret pages) incurs more entropy, since the header contains different pointers. This increases the entropy of the page that stores an object’s header significantly. When using secret pages, in contrast, the only entropy originates from a single pointer referencing one of the  $S$  secrets (Figure 3.6-B/C).



**Figure 3.7:** Birthday heap spray's reliability and memory trade-offs.

We now need to craft  $P$  probe values to guess at least one of the  $S$  secrets. We encode each value in a probe page to mimic the layout of the secret pages. To store our probe pages, we create a large `TypedArray`, which, compared to a regular `Array`, offers more controllable value representations. We fill each array page with 512 guessed 64 bit pointers similar to the secret pages (Figure 3.6-D).

After the generation step completes, a few of our  $P$  probe pages get inevitably deduplicated with some of our  $S$  secret pages. Since we can detect deduplicated probe pages using our memory deduplication side channel, we can now leak correctly guessed pointer values and break heap ASLR.

### Dealing with noise

If we want to add some redundancy in detecting an object's address, we cannot simply create extra identical probe pages. Identical probe pages get deduplicated together, resulting in false positives. However, since we have no restrictions on how to “encode” our secret into a page, we can add variations to create extra sets of secret and probe pages. One way is to fill all but one of the available slots (i.e., 511 slots) with a reference to our object, and fill the remaining slot with a different magic number for each redundant set of pages.

### Time and memory requirements

Our implementation of the birthday heap spray primitive requires only a single deduplication pass to obtain a heap pointer. For the execution of the attack, we need to

Pointer type	Memory	Dedup passes	Time
Unknown code	16 MB	3	45 Minutes
Known code	8 MB	2	30 Minutes
Heap	500 MB	1	15 Minutes
Heap + unknown code	516 MB	3	45 Minutes
Heap + known code	508 MB	2	30 Minutes

**Table 3.1:** Time and memory requirements to leak pointers in the current implementation.

allocate three chunks of memory. A first chunk is needed for the  $S$  1 MB-aligned target objects, resulting in  $S \cdot 2^{20}$  bytes. A second chunk of memory is needed for the secret pages. With a redundancy factor  $r$ , we need  $S \cdot r \cdot 2^{12}$  bytes for the secret pages. To cover 20 bits of entropy, we need  $P = \frac{2^{20}}{S}$  probe pages, each with a  $r$  redundancy factor, resulting in  $\frac{2^{20}}{S} \cdot r \cdot 2^{12}$  bytes. Figure 3.7 shows the memory requirements for different redundancy factors based on this formula. With our target redundancy factor of three (which we found sufficient and even conservative in practice), we can leak a heap pointer with only 500 MB of memory. Table 3.1 summarizes the end-to-end requirements for our attacks to leak code and heap pointers. Note that we can leak part of a code pointer and a complete heap pointer in the same deduplication pass (15 minutes). Given a known version of `chakra.dll`, we can leak both pointers in two deduplication passes (30 minutes).

### 3.5.4 Discussion

In this section, we described the implementation of our memory deduplication primitives in Microsoft Edge’s JavaScript engine. Using our alignment probing primitive, we leaked a randomized code pointer and, using our birthday heap spray primitive, we leaked a randomized heap pointer. We successfully repeated each of these attacks 10 times.

In the next section, we describe the first remote Rowhammer exploit that extensively relies on our memory deduplication primitives to disclose randomized pointers. To the best of our knowledge, ours is the first modern browser exploit that does not rely on any software vulnerability.

We did not need to employ the partial reuse primitive for our Rowhammer exploit, but, while we found that controlling memory reuse on the browser heap is non-trivial, we believe that our partial reuse primitive can be still used to leak randomized stack addresses by triggering deep functions in JIT code and partially overwriting the stack. In Section 3.7, in turn, we extensively use our partial reuse primitive to leak a 30 byte password hash from a network server—part of a class of applications which is, in contrast, particularly susceptible to controlled memory reuse attacks.

## 3.6 Rowhammering Microsoft Edge

Rowhammer [69] is a widespread DRAM vulnerability that allows an attacker to flip bits in a (victim) memory page by repeatedly reading from other (aggressor) memory pages. More precisely, repeated activations of rows of physical memory (due to repeated memory read operations) trigger the vulnerability. The bit flips are deterministic: once we identify a vulnerable memory location, it is possible to reproduce the bit flip patterns by reading again the same set of aggressor pages.

We report on the first reliable remote exploit for the Rowhammer vulnerability running entirely in Microsoft Edge. The exploit does not rely on any software vulnerability for reliable exploitation. It only relies on our alignment probing primitive and our birthday heap spray primitive to leak code and heap pointers (respectively), which we later use to create a counterfeit object. Our counterfeit object provides an attacker with read/write access to Microsoft Edge’s virtual memory address space.

To reliably craft our end-to-end exploit, we had to overcome several challenges, which we now detail in the remainder of the section. First, we describe how we triggered the Rowhammer vulnerability in Microsoft Edge running on Windows 10. Next, we describe how we used our deduplication primitives to craft a (large) counterfeit JavaScript object inside the data area of a valid (small) target object. Finally, we describe how we used Rowhammer to pivot from a reference to a valid target object to our counterfeit object, resulting in arbitrary memory read/write capabilities in Microsoft Edge.

### 3.6.1 Rowhammer Variations

In the literature, there are two main variations on the Rowhammer attack. Single-sided Rowhammer repeatedly activates a single row to corrupt its neighboring rows’ cells. Double-sided Rowhammer targets a single row by repeatedly activating both its neighboring rows. Prior research shows that double-sided Rowhammer is generally more effective than single-sided Rowhammer [111].

The authors of Rowhammer.js [57], an implementation of the Rowhammer attack in JavaScript, rely on Linux’ anonymous huge page support. A huge page in a default Linux installation is allocated using 2 MB of contiguous physical memory—which spans across multiple DRAM rows. Hence, huge pages make it possible to perform double-sided Rowhammer from JavaScript. Unfortunately, we cannot rely on huge pages in our attack since Microsoft Edge does not explicitly request huge pages from the Windows kernel.

Another option we considered was to rely on large allocations. We expected Windows to allocate contiguous blocks of physical memory when requesting large amounts of memory from JavaScript. However, Windows hands out pages from multiple memory pools in a round robin fashion. The memory pages in each of these pools belong to the same CPU cache set [95], which means that large allocations are not backed by contiguous physical pages. We later made use of this observation to



efficiently create cache eviction sets, but it is not immediately clear how we could use these memory pools to find memory pages that belong to adjunct memory rows and perform double-sided Rowhammer. Hence, we ultimately opted for single-sided Rowhammer in JavaScript.

### 3.6.2 Finding a Cache Eviction Set on Windows

The most effective way to hammer a row is to use the `clflush` instruction, which allows one to keep reading from main memory instead of the CPU cache. Another option is to find eviction sets for a specific memory location and exploit them to bypass the cache.

Since the `clflush` instruction is not available in JavaScript, we need to rely on eviction sets to perform Rowhammer. Earlier, we discovered that Windows hands out physical pages based on the underlying cache sets. As a result, the addresses that are 128 KB apart are often in the same cache set. We use this property to quickly find cache eviction sets for memory locations that we intend to hammer. Modern Intel processors after Sandy Bridge introduced a complex hash function to further partition the cache into slices [61; 84]. An address belongs to an eviction set if the address and the eviction set belong to the same cache slice. We use a cache reduction algorithm similar to [57] to find minimal eviction sets<sup>3</sup> in a fraction of a second.

To prevent our test code from interfering with our cache sets, we created two identical routines to perform Rowhammer and determine cache sets. The routines are placed one page apart in memory, which ensures the two routines are located on different cache sets. If one of the two routines interferes with an eviction set, the other one does not. In order to find our eviction set, we run our test on both routines and pick the fastest result. Likewise, before hammering, the fastest test run determines which routine to use.

### 3.6.3 Finding Bit Flips

In order to find a vulnerable memory location, we allocate a large Array filled with doubles. We make sure these doubles are encoded using the XOR pattern described in Section 3.4.2 by explicitly placing some references in the Array. This allows us to encode a double value such that all bits are set to 1<sup>4</sup>. We then find eviction sets and hammer 32 pages at a time. We read from each page two million times before moving to the next page. After hammering each set of 32 pages, we check the entire Array for bit flips.

After scanning a sufficient number of pages, we know which bits can be flipped at which offsets. Next, we need to determine what to place in the vulnerable memory locations to craft our exploit. For this purpose, our goal is to place some data in our

<sup>3</sup>Minimal eviction sets contain  $N + 1$  entries for an  $N$ -way L3 cache.

<sup>4</sup>The double value 5.5626846462679985e-309 has all bits set considering the XOR pattern discussed in Section 3.4.2.

Array which, after a bit flip, can yield a reference to a controlled counterfeit object. We now first describe how to obtain a bit-flipped reference to a counterfeit object and then how to craft a counterfeit object to achieve arbitrary memory read/write capabilities.

### 3.6.4 Exploiting Bit Flips

We developed two possible techniques to exploit bit flips in Microsoft Edge:

#### Type flipping

Given the double representation described in Section 3.4.2, a high-to-low (i.e., 1-to-0) bit flip in the 11-bit exponent of a double element in our large Array allows us to craft a reference to any address, including that of our counterfeit object's header. In essence, this bit flip changes an attacker-controlled double number into a reference that points to an attacker-crafted counterfeit object.

#### Pivoting

Another option is to directly corrupt an existing valid reference. For this purpose, we can store a reference to a valid target object in a vulnerable array location. By corrupting the lower bits of the reference, we can then pivot to our counterfeit object's header. Assuming an exploitable high-to-low bit flip, our corrupted reference will point to a lower location in memory. If we fabricate our counterfeit object's header at this location, we can then use the corrupted reference to access any memory addressable by the counterfeit object. Recall from Section 3.5.3 that we sprayed our target objects close to each other. By corrupting a reference to one of these (small) objects, we obtain a reference to the middle of the previous valid target object. Since we control the memory contents pointed by the corrupted reference (our small target objects use in-band data), we can fabricate our counterfeit object at that location.

These two attacks make it possible to exploit 23 out of every 64 high-to-low bit flips (i.e., 36% of the bit flips are exploitable). We now describe how we create our counterfeit object before summarizing our attack.

### 3.6.5 Creating a Counterfeit JavaScript Object

To craft a valid large counterfeit object, we rely on the code pointer to the Chakra's binary we leaked using our alignment probing primitive, and on the heap pointer we leaked using our birthday heap spray primitive. Our counterfeit object (of type `Uint8Array`) resides inside a JavaScript Array containing only IEEE754 double values. As discussed in Section 3.4.2, this type of array does not XOR its values with a constant, allowing us to easily craft arbitrary pointers inside the array (or any other non-*NaN* value).

To craft our counterfeit `Uint8Array` object, we need a valid `vtable` pointer. We obtain the latter by simply adding a fixed offset to our leaked code pointer. Other important fields in the `Uint8Array` object are its size and a pointer to its out-of-band data buffer. We obtain the latter by simply using our leaked heap address. These fields are sufficient to allow a compiled JavaScript routine to use our `Uint8Array` object. The generated assembler performs a type comparison on the `vtable` pointer field and performs bound checking on the size field. Note that the crafted counterfeit object does not violate any of the CFI rules in Microsoft Edge [133].

At this stage, since we control the out-of-band data buffer location our counterfeit `Uint8Array` points to, we can read or write from anywhere in the Microsoft Edge's address space with valid virtual mappings. To reliably identify all the valid mappings, we can first use our counterfeit object to dump the contents of the current heap and find heap, stack or code pointers that disclose additional virtual mapping locations in the address space. We can now get access to the newly discovered locations by crafting additional counterfeit objects (using the current counterfeit object) and discover new pointers. Alternating disclosure steps with pointer chasing steps allows us to incrementally disclose the valid virtual mappings and control the entirety of the address space, as also shown in prior work [29; 34; 82; 119].

### 3.6.6 Dealing with Garbage Collection

Using the counterfeit objects directly (as done above) provides us with arbitrary read/write access to Microsoft Edge's address space, but as soon as a garbage collection pass checks our counterfeit object, the browser may crash due to inconsistent state in the garbage collector.

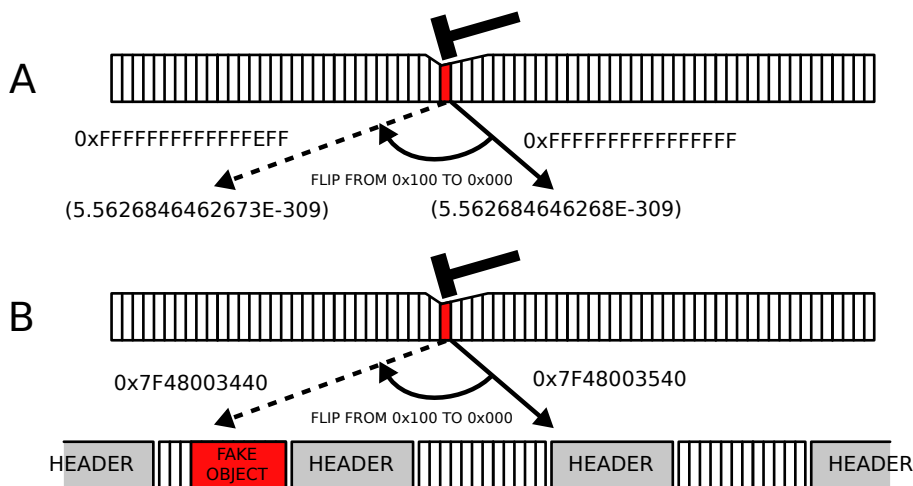
To avoid this scenario, we have to minimize the amount of time that we use the counterfeit object directly. For this purpose, *we only use the counterfeit object to corrupt the header of other valid objects* and we immediately restore the (corrupted) reference to our counterfeit object afterwards.

To this end, we use the (leaked) pointer to a valid target object's header as the backing data buffer of our crafted counterfeit `Uint8Array` object. This allows us to corrupt the size field of the target (array) object and operate out-of-bounds accesses from the corresponding valid array references. This again grants us arbitrary memory read/write access to the underlying heap and, using the incremental disclosure strategy described earlier, to the entirety of Microsoft Edge's address space.

### 3.6.7 Putting the Pieces Together

Using Figure 3.8, we now summarize the steps of our end-to-end Rowhammer attack with the pivoting technique described in Section 3.6.4. The attack using the type flipping technique is similar and we omit it for brevity.

As shown in Figure 3.8-A, at this stage of the attack, we have access to a bit flip inside a controlled array. We can now trigger the bit flip and pivot to our counterfeit



**Figure 3.8:** By flipping a bit in an object pointer, we can pivot to the attacker’s counterfeit object. First, we identify a vulnerable memory location within an array (A). After finding an exploitable bit flip, we store a valid object reference at the vulnerable memory location and pivot to a counterfeit object with Rowhammer (B).

object. For this purpose, we store a reference to a valid object at the vulnerable location inside the large double array we created earlier (Section 3.6.3). We choose our valid object in a way that, when triggering a bit flip, its reference points to our counterfeit object, as shown in Figure 3.8-B.

With the arbitrary read/write primitive provided by our counterfeit object, gaining code execution is achievable even under a strong CFI implementation, as shown by [25].

### Time and memory requirements

To leak the code and heap pointers necessary for our Rowhammer attack, we need 508 MB of memory and 30 minutes for two deduplication passes, as reported in Table 3.1 (assuming a known version of `chakra.dll`). In addition, for the Rowhammer attack, we need 1 GB of memory to find bit flips and 32 MB of memory for our cache eviction sets. The time to find an exploitable bit flip, finally, depends on the vulnerable DRAM chips considered, with prior large-scale studies reporting times ranging anywhere from seconds to hours in practice [69].

### 3.6.8 Discussion

In this section, we showed how an attacker can use our deduplication primitives to leak enough information from the browser and craft a reliable Rowhammer exploit. Our exploit does not rely on any software vulnerability and runs entirely in the browser, increasing its impact significantly. We later show how an in-browser attacker can

use our primitives to also attack a process outside the browser sandbox and present mitigation strategies in Section 3.8.

Finally, we note that, to trigger bit flips using Rowhammer, we had to increase our DRAM's refresh time, similar to Rowhammer.js [57]. However, we believe that more vulnerable DRAMs will readily be exploitable without modifying the default settings. We are currently investigating the possibility of double-sided Rowhammer in Microsoft Edge using additional side channels and more elaborate techniques to induce bit flips with the default DRAM settings.<sup>5</sup>

## 3.7 System-wide Exploitation

In the previous sections, we focused on a JavaScript-enabled attacker using our primitives to conduct an advanced exploitation campaign inside the browser. In this section, we show how the same attacker can break out of the browser sandbox and use our primitives for system-wide exploitation targeting unrelated processes on the same system. We focus our analysis on network servers, which accept untrusted input from the network and thus provide an attacker with an entry point to control memory alignment and reuse.

We consider an attacker running JavaScript in the browser and seeking to fulfill three system-wide goals: (i) fingerprinting the target network server version running on the same system; (ii) disclosing the password hash of the `admin` network server user; (iii) disclosing the heap (randomized using 64 bit ASLR) by leaking a heap pointer. We show that crafting our primitives to conduct all such attacks is remarkably easy for our attacker, despite the seemingly constrained attack environment. This is just by exploiting the strong spatial and temporal memory locality characteristics of typical high-performance network servers.

For our attacks, we use the popular `nginx` web server (v0.8.54) as an example. We use the 64 bit version of `nginx` running in `Cygwin` as a reference for simplicity, but beta `nginx` versions using the native Windows API are also available. We configure `nginx` with a single root-level (`\0`-terminated) password file containing a randomly generated HTTP password hash for the `admin` user and with 8 KB request-dedicated memory pools (`request_pool_size` configuration file directive). Using 8 KB pools (4 KB by default in our `nginx` version) is a plausible choice in practice, given that the maximum HTTP request header length is 8 KB and part of the header data is stored in a single pool. Before detailing the proposed deduplication-based attacks, we now briefly summarize `nginx`' memory allocation behavior.

---

<sup>5</sup>After submission of the camera-ready version of the paper this chapter is based on, we were able to trigger bit-flips at normal refresh rates by using multiple JavaScript worker-threads to do the hammering.

### 3.7.1 Memory Allocation Behavior

nginx implements two custom memory allocators on top of the standard `malloc` implementation, a slab allocator and a region-based (or pool) allocator [13]. We focus our analysis here on nginx' pool allocator, given that it manages both user (and thus attacker-controllable) data and security-sensitive data, and also tends to allocate many small objects consecutively in memory.

The pool allocator maintains a number of independent pools, each containing logically and temporally related objects. Clients are provided with `create`, `alloc`, and `destroy` primitives to respectively create a new pool, allocate objects in it, and destroy an existing pool (with all the allocated objects). Each pool internally maintains two singly linked lists of data blocks (`ngx_pool_t`) and large blocks (`ngx_pool_large_t`). The latter are only used for large chunks (larger than 4 KB)—rarely allocated during regular execution—so we focus our analysis on the former.

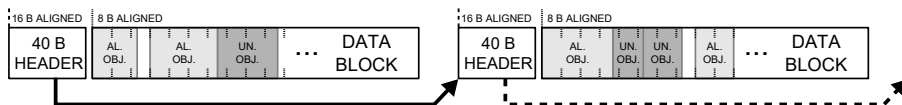
Figure 3.9 exemplifies how nginx manages data blocks in a single pool and (small) objects with the contained data blocks. Each data block is allocated through the standard `malloc` allocator, using a fixed 8 KB size and a 16 byte alignment. The data block is prepended with a 40 byte pool header, while the rest of the block is entirely dedicated to object allocations. Within a block, objects are allocated consecutively similar to a simple buffer allocator. For this purpose, the pool header maintains the pointer to the next free chunk in the corresponding data block.

The pool allocator offers two allocation primitives to allocate objects in each pool: `palloc` (which allocates *unaligned objects* at the byte boundary) and `malloc` (which allocates *aligned objects* at the 8 byte boundary). Whatever the allocation primitive, the allocator checks if there is space available for the requested object size in the current (i.e., last allocated) data block. If sufficient space is available, the object is allocated right after the last allocated object. Otherwise (provided no space is available in other data blocks), the allocator `mallocs` and appends a new data block to the pool's data block list. In such a case, the object is allocated in the new data block right after the header.

Note that, once allocated, an object (or a data block) lives until the pool is destroyed, as the allocator offers no object- or block-level deallocation primitives. This simple design is ideal to allocate logically related objects with a limited and well-determined lifetime. Among others, nginx maintains one pool for each connection (*connection pool*) and one pool for each HTTP request (*request pool*). The request pool, in particular, contains much client-controlled and security-sensitive data, which an attacker can abuse to fulfill its goals, as we show in Section 3.7.4 and Section 3.7.5.

### 3.7.2 Controlling the Heap

To craft our primitives, an attacker needs to control the layout of data in memory and ensure that the target data are not overwritten before a deduplication pass oc-



**Figure 3.9:** Aligned and unaligned objects allocated in a single nginx pool.

curs. While this is trivial to implement in the attacker-controlled memory area in a browser, it is slightly more complicated for server programs with many concurrent connections and allocators that promote memory reuse for efficiency.

Despite the challenges, we now show how an attacker can spray the heap of a network server such as nginx to reliably force the memory allocator to generate a long-lived page-sized data pattern in memory. The pattern contains attacker-controlled data followed by a target secret. In particular, we show an attacker can fulfill two goals: (i) achieving a target pattern alignment to retain a controlled number of secret bytes before the page boundary, and (ii) ensuring that the target pattern is not overwritten by other data.

To achieve our first goal, we can simply spray the heap using thousands of parallel HTTP requests, ensuring some of the generated patterns will land on a target alignment within a page with high probability. To verify this intuition, we issued 100,000 HTTP requests using 1,000 concurrent connections to nginx. Our results confirmed that we can easily gain 22 unique patterns with a given target alignment on the heap (median of 11 runs) in a matter of seconds.

To achieve our second goal, we need to prevent at least some of the pages hosting the generated pattern from being overwritten by other data. This is not possible within the same HTTP request (the request pool allows for no internal reuse by construction), but it is, in principle, possible after the server has finished handling the request. Since each data block is allocated through the standard `malloc` allocator, reuse patterns depend on the underlying system allocator.

Standard `malloc` allocators are based on free lists of memory blocks (e.g, `ptmalloc` [2]) and maintain per-size free lists in MRU (Most Recently Used) fashion. This strategy strongly encourages reuse patterns across blocks of the same size. Since the size of the data block in the request pool is unique in nginx during regular execution, this essentially translates to attacker-controlled request data blocks being likely only reused by other requests' data blocks. Some interference may occur with allocators coalescing neighboring free blocks, but the interference is low in practice, especially for common server programs such as nginx which use very few fixed and sparse allocation sizes.

Hence, the main question that we need to answer is whether patterns with the target alignment are overwritten by requests from other clients. Since the underlying memory allocators maintain their free lists using MRU, we expect that under normal load only the most recently used blocks are reused. As a result, an attacker flooding the server with an unusually large number of parallel requests can force the allocator to reach regions of the *deep heap* which are almost never used during execu-

tion. To verify this intuition, this time we issued 100,000 HTTP requests using 100 (compared to the previous 1,000) concurrent connections to nginx. Our results confirmed that only three unique patterns with a given target alignment (median of 11 runs) could be found on the heap, overwriting only the 14% of the patterns sprayed on the heap by an attacker using an order of magnitude larger number of concurrent requests.

We use our heap spraying technique when disclosing password hashes in Section 3.7.4 and heap pointers in Section 3.7.5.

### 3.7.3 Server Fingerprinting

To fingerprint a running nginx instance, an attacker needs to find one or more unique non-file-backed memory pages to deduplicate. We note that there may be other ways to fingerprint running server programs, for instance, by sending a network request on well-known ports and looking for unique response patterns. In some cases, this is, however, not possible or not version-accurate. In addition, server fingerprinting is much more efficient with memory deduplication. In a single pass, an attacker can efficiently look for many running vulnerable programs or simply for programs with high exposure to deduplication-based attacks from a database. We stress that none of the attacks presented here exploit any software vulnerabilities.

In many running programs, it is easy for an attacker to find memory pages for fingerprinting purposes in the data section. Many of such pages are written to in a predictable way during the early stages of execution and never change again once the program reaches a steady state. Such access patterns are particularly common for server programs, which initialize most of their data structures during initialization and exhibit read-only behavior on a large fraction of them after reaching a steady state [53].

To confirm our intuition in nginx, we compared the contents of all the pages in its data segment after initialization (baseline) against the content of the same pages after running all the tests of the publicly available nginx test suite [5]. Our results showed that three out of the total eight data pages always remain identical, despite the test suite relying on a very different (and peculiar) nginx configuration compared to the standard one used in our original baseline. The attacker can abuse any of these three data pages (e.g., the data page at offset 0x2000), or all of them for redundancy, to detect our version of nginx running next to the host browser on the same system. Once the attacker has fingerprinted the target version, she can start sending network requests from a remote client (after scanning for the server port) to craft our primitives.

### 3.7.4 Password Disclosure

To disclose the HTTP password hash of the `admin` user using our alignment probing primitive, an attacker needs to first control the alignment of the password hash in



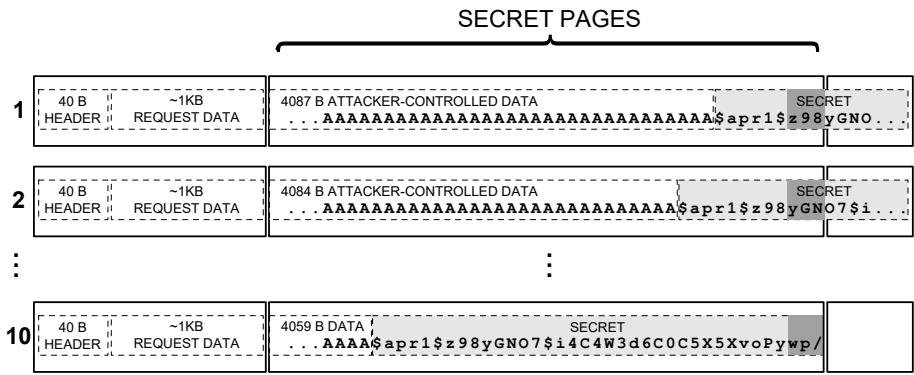


Figure 3.10: nginx password hash disclosure using alignment probing.

memory and predict neighboring data. For both conditions to happen, the attacker needs to control data which is logically allocated close to the target password hash and predict the memory allocation behavior. Both constraints are easy to satisfy in network servers.

To satisfy the first constraint, an attacker can rely on the input password provided in the HTTP request as control data. The intuition is that the target password hash is generally allocated close to the input password for authentication purposes. We confirmed this intuition in nginx (which stores the target password hash right next to the input password), but also in other common network servers such as vsftpd, proftpd, pure-ftpd, and sshd. To satisfy the second constraint, we rely on our heap spraying technique discussed in Section 3.7.2.

In nginx, the target password hash is allocated in the request pool right after the input password and with no alignment restrictions. In particular, on a typical (and short) HTTP authentication request for the admin user with the last (Authorization) header including the input password (such as the one issued by `wget --user=admin --password=P`), nginx allocates only a single data block in the request pool. The data block consecutively stores the 40 byte pool header, around 1 KB worth of request-dependent data objects, the input password, and the target password hash. The input password is base64-encoded by the client and stored in decoded form in memory by nginx. The target password hash is by default stored in memory by nginx as follows: `$apr1$S$H`, where `apr1` is the format (MD5), `S` is the 8 byte salt, and `H` is the 22 byte base64-encoded password hash value.

To craft a alignment probing primitive, an attacker can arbitrarily increase the size of the input password (up to 4 KB) one byte at the time (even invalid base64 strings are tolerated by nginx). This would progressively shift the target password hash in memory, allowing the attacker to control its alignment and mount a deduplication-based disclosure attack. As a result of input password decoding and some data redundancy, however, a given target password hash can only be shifted at the 3 byte granularity by increasing the input size. Nevertheless, this is sufficient for an attacker

to incrementally disclose three bytes of the password hash at the time. Figure 3.10 outlines the different stages of the attack.

To start off the attack, an attacker can send a HTTP request with a known (decoded) password pattern of 4,087 bytes. If this pattern happens to be allocated at the page boundary, then the remaining nine bytes of the page will be filled with the `$apr1$` string followed by the first three bytes of the salt within the same data block. Once the crafted page-aligned pattern is in memory, the attacker can use memory deduplication to disclose the first three bytes of the salt in a single pass. Given that the target password hash is encoded using 6 bit base64 symbols, this requires crafting  $2^{18}$  probe pages. The attacker can then proceed to incrementally disclose the other bytes of the salt first and the password hash then, by gradually reducing the input password size in the HTTP request and shifting the target password hash towards lower addresses three bytes at a time.

There are three issues with this naive version of the attack: (i) the target pattern is not necessarily page-aligned, (ii) the target pattern may be overwritten by requests from other clients, and (iii)  $2^{18}$  probe pages require 1 GB of memory without redundancy, which is large and prone to noise. To address all these issues, we rely on our heap spraying technique. Instead of issuing one request, we issue 100,000 request with our target alignment over 1,000 open connections. This allows us to reach the deep heap with our desired alignment, addressing (i) and (ii). Furthermore, thanks to the abundant redundancy when spraying the heap, the attacker can easily find many page-aligned patterns with all the three possible target password hash alignments. This enables a resource-constrained attacker to disclose two (rather than three) bytes of the password hash at the time, reducing the required memory to only 16 MB in exchange for extra deduplication passes (15 instead of 10).

### 3.7.5 Heap Disclosure

To leak a heap pointer (randomized using 64 bit ASLR), the alignment probing primitive used earlier is insufficient. Given that pointers are always stored in aligned objects within a data block, the attacker would be, in principle, left with guessing eight bytes at the time. In practice, Windows ASLR only uses 24 bits of entropy for the base of the heap, resulting in 36 bits of uncertainty in the lowest five bytes of arbitrary heap pointers. This is still problematic for our alignment probing primitive.

To lower the entropy, however, the attacker can deploy our partial reuse primitive by exploiting predictable memory reuse patterns. Our primitive further requires the attacker to control the alignment of a target heap pointer and some known pattern in memory. All these requirements are easy to satisfy when abusing nginx' pool allocator. To exemplify our attack, we consider the same HTTP authentication request as in our password hash disclosure attack, but we assume a (randomly crafted) invalid user in the request.

When an invalid user is specified, nginx refuses to load the target password hash into memory right after the provided input password (as done normally) and logs

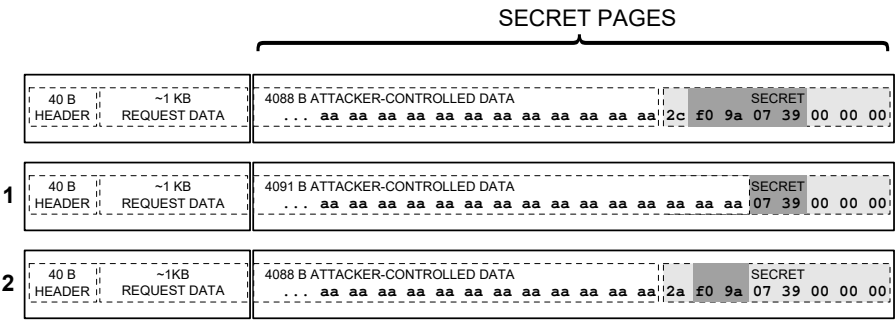


Figure 3.11: ngx heap disclosure using partial reuse.

an error. The error logging function (`ngx_http_write_filter`), however, immediately allocates a 8 byte-aligned buffer object (`ngx_buf_t`) of 52 bytes and an unaligned log buffer in the request pool. Since the allocation behavior is deterministic, the attacker can control memory reuse patterns and partially overwrite pointers inside the buffer object to lower their entropy and incrementally disclose heap addresses. We use the first (pointer) field (`pos`) in the `ngx_buf_t` buffer object as a reference to demonstrate the attack, whose stages are outlined in Figure 3.11.

To start off the attack, the attacker can specify an input password with a known (decoded) pattern of 4,088 bytes. By spraying the heap, many instances of this pattern will be allocated at the page boundary, with the remaining eight bytes of the page filled with the `pos` pointer. The attacker can then send a second request of 4,091 bytes, which, in all the page-aligned pattern instances, will reuse (and override) the lowest three bytes of the old `pos` pointer data, while forcing the pool allocator to align the new `pos` pointer to eight bytes after the page boundary. This strategy leaves only 16 bits of uncertainty left in the old pointer (the first byte and the last three bytes are now known), sufficiently lowering the entropy to disclose two pointer bytes in a single memory deduplication pass.

In a second stage, the attacker can repeat the first step above to disclose the remaining lowest three bytes (the rest are now known). To lower the entropy, the attacker can rely on the fact that the `pos` pointer is always pointed into the beginning of the log buffer, i.e., exactly 52 bytes away. In other words, a `pos` pointer allocated right before the page boundary will always contain the value `0x30` in its lowest 12 bits. This leaves only 12 bits of uncertainty left in the target pointer, which the attacker can easily disclose in a second memory deduplication pass. In total, to disclose a heap pointer we require 256 MB of memory without redundancy and two deduplication passes.

### 3.7.6 Dealing with Noise

We fingerprinted ngx as described using three unique data pages, providing us with a redundancy factor of three to battle noise. The password hash and heap disclosure

Attack	Memory	Dedup passes	Time
Fingerprinting	12 KB	1	15 Minutes
Password disclosure	48 MB	15	225 Minutes
Heap disclosure	768 MB	2	30 Minutes

**Table 3.2:** Time and memory requirements for our deduplication-based attacks against nginx.

attacks described above, however, have no redundancy.

To add redundancy to our two attacks, we rely on the attacker’s ability to control the content of the request and create different target patterns in memory. To this end, we can simply issue our requests using three different request types (e.g., using different input passwords) in a round-robin fashion. As discussed in Section 3.7.2, on average, 19 pattern pages remain in the deep heap with the desired target alignment in a steady state. Given three different request types, on average, we still obtain 6.3 memory pages in the deep heap, each with the desired target alignment but with a different attacker-controlled pattern. On the JavaScript side, the attacker can now use this redundancy to create additional pages (that target different patterns) to increase the reliability of the attacks.

### 3.7.7 Time and Memory Requirements

Table 3.2 summarizes the time and memory requirements for our three deduplication-based attacks against nginx. The reported numbers assume a redundancy factor of three to deal with noise.

## 3.8 Mitigation

The main motivation behind memory deduplication is to eliminate memory pages with similar content and use physical memory more efficiently. To maximize the deduplication rate, a memory deduplication system seeks to identify candidates with *any* possible content. The implicit assumption here is that many different page contents contribute to the deduplication rate. However, this property also allowed us to craft the powerful attack primitives detailed in this chapter.

We now show this assumption is overly conservative in the practical cases of interest. More specifically, we show that only deduplicating *zero pages* is sufficient to achieve a nearly optimal deduplication rate, while completely eliminating the ability to program memory deduplication and perform dangerous computations. Table 3.3 compares the achievable deduplication rate of full deduplication with that of zero page deduplication in Microsoft Edge, measured in percentage of saved memory. In each experiment, we opened eight tabs visiting the most popular websites<sup>6</sup>. We then changed the number of websites across tabs to emulate the user’s behavior and

<sup>6</sup>[https://en.wikipedia.org/wiki/List\\_of\\_most\\_popular\\_websites](https://en.wikipedia.org/wiki/List_of_most_popular_websites)

Website Diversity	Full Deduplication	Zero Pages Only
1	0.13	0.12
2	0.13	0.11
4	0.14	0.13
8	0.14	0.12

**Table 3.3:** Full deduplication rate versus deduplication rate of zero pages alone under different settings in Microsoft Edge.

measure its impact on the deduplication rate. We call this metric “Website Diversity”. For example, with diversity of eight, each tab opens a different website, and with diversity of one, each tab opens the same website. According to our measurements, deduplicating zero pages alone can retain between 84% and 93% of the deduplication rate of full deduplication. We hence recommend deduplicating zero pages alone for sensitive, network-facing applications such as browsers. In highly security-sensitive environments, full memory deduplication is generally not advisable.

## 3.9 Related work

We discuss previous work on side channels over shared caches (Section 3.9.1) and deduplication (Section 3.9.2). We then look at the Rowhammer vulnerability (Section 3.9.3) we used in our end-to-end attack on Microsoft Edge.

### 3.9.1 Side Channels over Shared Caches

Recently accessed memory locations remain in the last-level cache (LLC) shared across different cores. Accessing cached locations is considerably faster than loading them directly from memory. This timing difference has been abused to create a side channel and disclose sensitive information.

The FLUSH+RELOAD attack [131] leaks data from a sensitive process, such as one using cryptographic primitives, by exploiting the timing differences when accessing cached data. Irazoqui et al. [62] improve this attack, retrieving cryptographic keys in the cloud with a combination of FLUSH+RELOAD and a memory deduplication side channel. Using a similar attack, Zhang et al. [135] leak sensitive data to hijack user accounts and break SAML single sign-on.

The “RELOAD” part of the FLUSH+RELOAD attack assumes the attacker has access to victims’ code pages either via the shared page cache or some form of memory deduplication. The PRIME+PROBE attack [96; 80] lifts this requirement by only relying on cache misses from the attacker’s process to infer the behavior of the victim’s process when processing secret data.

Oren et al. [95] use the PRIME+PROBE attack in a sandboxed browser tab to leak sensitive information (e.g., key presses) from a user’s browser. By performing three

types of PRIME+PROBE attacks on the CPU caches and the TLB, Hund et al. [61] map the entire address space of a running Windows kernel, breaking kernel-level ASLR.

To perform PRIME+PROBE, the attacker needs the mapping of memory locations to cache sets. This mapping is complex and difficult to reverse engineer in modern Intel processors [61]. Maurice et al. [84] use performance counters to simplify the reverse engineering process. As discussed in Section 3.6.2, we instead rely on the behavior of Windows' page allocator to quickly construct the cache eviction sets for our Rowhammer exploit. To the best of our knowledge, this is the first example of an attack using a side channel other than timing to construct such eviction sets in a sandboxed browser.

In response to numerous cache side-channel attacks, Kim et al. [68] propose a low-overhead cache isolation technique to avoid cross-talk over shared caches. By dynamically switching between diversified versions of a program, Crane et al. [32] change the mapping of program locations to cache sets, making it difficult to perform cache attacks. These techniques, however, have not (yet) become mainstream.

### 3.9.2 Side Channels over Deduplication

Side channels over data deduplication systems can be created over stable storage or main memory.

#### Stable storage

File-based storage deduplication has been previously shown to provide a side channel to leak information on existing files and their content [60; 91]. The first instance warning users about this issue is a Microsoft Knowledge Base article that mentions a malicious user can use the deduplication side channel to leak secret information over shared deduplicated storage [4].

Harnik et al. [60] show that file deduplication at the provider's site can allow an attacker to fingerprint which files the provider stores and brute force their content if a major fraction of each file is already known. Mulazzani et al. [91] implement a similar attack on Dropbox, a popular cloud file storage service.

#### Main memory

There are several cross-VM attacks that rely on VMM-based memory deduplication to fingerprint operating systems [97] or applications [123], detect cryptographic libraries [63], and create covert channels for stealthy backdoors [129]. Gruss et al. [56] show it is possible to perform a similar attack in a sandboxed browser tab to detect running applications and open websites.

CAIN [11] can leak randomized code pointers of neighboring VMs using the memory deduplication side channel incorporated into VMMs. CAIN, however,

needs to brute force all possible pointer values to break ASLR. Rather than relying on memory deduplication, Xu et al. [130] show that malicious VMMs can purposefully force page faults in a VM with encrypted memory to retrieve sensitive information.

All these previously published attacks rely on the assumption that programming memory deduplication only allows for a single-bit side channel per page. As we showed in this chapter, by controlling the alignment/reuse of data in memory or mounting birthday attacks, memory deduplication can be programmed to leak high-entropy information much more efficiently. For example, by applying our alignment probing primitive to JIT code, we can leak code pointers with significantly lower memory requirements than a purely brute-force approach; by using our partial reuse primitive and our birthday heap spray primitive, we can leak high-entropy heap data pointers inside a server and a browser program (respectively) for the first time through a side channel.

### 3.9.3 Rowhammer Timeline

The Rowhammer bug was first publicly disclosed by Kim et al. [69] in June 2014. While the authors originally speculated on the security aspects of Rowhammer, it was not clear whether it was possible to fully exploit Rowhammer until later. Only in March 2015, Seaborn and Dullien [112] published a working Linux kernel privilege escalation exploit using Rowhammer. Their native exploit relies on the ability to spray physical memory with page-table entries, so that a single bit flip can probabilistically grant an attacker-controlled process access to memory storing its own page-table information. Once the process can manipulate its own page tables, the attacker gains arbitrary read and write capabilities over the entire physical memory of the machine. In July 2015, Gruss et al. [57] demonstrated the ability to flip bits inside the browser using Rowhammer.

In this chapter, we showed that our memory deduplication primitives can provide us with derandomized pointers to code and heap. We used these pointers to craft the first reliable remote Rowhammer exploit in JavaScript.

## 3.10 Conclusions

Adding more and more functionality to operating systems leads to an ever-expanding attack surface. Even ostensibly harmless features like memory deduplication may prove to be extremely dangerous in the hands of an advanced attacker. In this chapter, we have shown that deduplication-based primitives can do much more harm than merely providing a slow side channel. An attacker can use our primitives to leak password hashes, randomized code and heap pointers, and start off reliable Rowhammer attacks. We find it extremely worrying that an attacker who simply times write operations and then reads from an unrelated addresses can reliably “own”

a system with all defenses up, even if the software is entirely free of bugs. Our conclusion is that we should introduce complex features in an operating system only with the greatest care (and after a thorough examination for side channels), and that full memory deduplication inside the operating system is a dangerous feature that is best turned off. In addition, we have shown that full deduplication is an overly conservative choice in the practical cases of interest and that deduplicating only zero pages can retain most of the memory-saving benefits of full deduplication while addressing its alarming security problems.

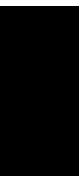
## Disclosure Statement

We have worked with our contacts at Microsoft to devise immediately deployable solutions against the security problems of memory deduplication evidenced in this chapter.

## Acknowledgments

We would like to thank our shepherd, Ilya Mironov, and the anonymous reviewers for their valuable comments. We would also like to thank the authors of Rowhammer.js for open-sourcing their implementation, which helped us create cache eviction sets inside Microsoft Edge. Finally, we would like to thank Matt Miller and Manuel Costa from Microsoft for the attention devoted to the issues raised in this chapter and commitment to devise immediately deployable solutions. This work was supported by the European Commission through project H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 644571, by NWO through the project VICI “Dowser”, and by the European Research Council through project ERC-2010-StG 259108 “Rosetta”.





## Flip Feng Shui: Hammering a Needle in the Software Stack

### Abstract

### Abstract

We introduce Flip Feng Shui (FFS), a new exploitation vector which allows an attacker to induce bit flips over *arbitrary* physical memory in a *fully controlled* way. FFS relies on hardware bugs to induce bit flips over memory and on the ability to surgically control the physical memory layout to corrupt attacker-targeted data anywhere in the software stack. We show FFS is possible today with very few constraints on the target data, by implementing an instance using the *Rowhammer bug* and *memory deduplication* (an OS feature widely deployed in production). Memory deduplication allows an attacker to reverse-map any physical page into a virtual page she owns as long as the page's contents are known. Rowhammer, in turn, allows an attacker to flip bits in controlled (initially unknown) locations in the target page.

We show FFS is extremely powerful: a malicious VM in a practical cloud setting can gain unauthorized access to a co-hosted victim VM running OpenSSH. Using FFS, we exemplify end-to-end attacks breaking OpenSSH public-key authentication, and forging GPG signatures from trusted keys, thereby compromising the Ubuntu/Debian update mechanism. We conclude by discussing mitigations and future directions for FFS attacks.

## 4.1 Introduction

The demand for high-performance and low-cost computing translates to increasing complexity in hardware and software. On the hardware side, the semiconductor industry packs more and more transistors into chips that serve as a foundation for our modern computing infrastructure. On the software side, modern operating systems are packed with complex features to support efficient resource management in cloud and other performance-sensitive settings.

Both trends come at the price of reliability and, inevitably, security. On the hardware side, components are increasingly prone to failures. For example, a large fraction of the DRAM chips produced in recent years are prone to bit flips [69; 111], and hardware errors in CPUs are expected to become mainstream in the near future [17; 28; 73; 116]. On the software side, widespread features such as memory or storage deduplication may serve as side channels for attackers [11; 20; 60]. Recent work analyzes some of the security implications of both trends, but so far the attacks that abuse these hardware/software features have been fairly limited—probabilistic privilege escalation [111], in-browser exploitation [20; 58], and selective information disclosure [11; 20; 60].

In this chapter, we show that an attacker abusing modern hardware/software properties can mount much more sophisticated and powerful attacks than previously believed possible. We describe Flip Feng Shui (FFS), a new exploitation vector that allows an attacker to induce bit flips over *arbitrary* physical memory in a *fully controlled* way. FFS relies on two underlying primitives: (i) the ability to induce bit flips in controlled (but not predetermined) physical memory pages; (ii) the ability to control the physical memory layout to reverse-map a target physical page into a virtual memory address under attacker control. While we believe the general vector will be increasingly common and relevant in the future, we show that an instance of FFS, which we term dFFS (i.e., deduplication-based FFS), can already be implemented on today’s hardware/software platforms with very few constraints. In particular, we show that by abusing Linux’ memory deduplication system (KSM) [9] which is very popular in production clouds [11], and the widespread Rowhammer DRAM bug [69], an attacker can *reliably* flip a single bit in *any* physical page in the software stack with known contents.

Despite the complete absence of software vulnerabilities, we show that a practical Flip Feng Shui attack can have devastating consequences in a common cloud setting. An attacker controlling a cloud VM can abuse memory deduplication to seize control of a target physical page in a co-hosted victim VM and then exploit the Rowhammer bug to flip a particular bit in the target page in a fully controlled and reliable way without writing to that bit. We use dFFS to mount end-to-end corruption attacks against OpenSSH public keys, and Debian/Ubuntu update URLs and trusted public keys, all residing within the page cache of the victim VM. We find that, while dFFS is surprisingly practical and effective, existing cryptographic software is wholly unequipped to counter it, given that “*bit flipping is not part of their threat*

*model*". Our end-to-end attacks completely compromise widespread cryptographic primitives, allowing an attacker to gain full control over the victim VM.

Summarizing, we make the following contributions:

- We present FFS, a new exploitation vector to induce hardware bit flips over arbitrary physical memory in a controlled fashion (Section 4.2).
- We present dFFS, an implementation instance of FFS that exploits KSM and the Rowhammer bug and we use it to bit-flip RSA public keys (Section 4.3) and compromise authentication and update systems of a co-hosted victim VM, granting the attacker unauthorized access and privileged code execution (Section 4.4).
- We use dFFS to evaluate the time requirements and success rates of our proposed attacks (Section 4.5) and discuss mitigations (Section 4.6).

The videos demonstrating dFFS attacks can be found in the following URL:

<https://vusec.net/projects/flip-feng-shui>

## 4.2 Flip Feng Shui

To implement an FFS attack, an attacker requires a *physical memory massaging primitive* and a *hardware vulnerability* that allows her to flip bits on certain locations on the medium that stores the users' data. Physical memory massaging is analogous to virtual memory massaging where attackers bring the virtual memory into an exploitable state [46; 47; 121], but instead performed on physical memory. Physical memory massaging (or simply *memory massaging*, hereafter) allows the attacker to steer victim's sensitive data towards those physical memory locations that are amenable to bit flips. Once the target data land on the intended vulnerable locations, the attacker can trigger the hardware vulnerability and corrupt the data via a controlled bit flip. The end-to-end attack allows the attacker to flip *a bit of choice* in *data of choice anywhere* in the software stack in a controlled fashion. With some constraints, this is similar to a typical arbitrary memory write primitive used for software exploitation [25], with two key differences: (i) the end-to-end attack requires no software vulnerability; (ii) the attacker can overwrite arbitrary physical (not just virtual) memory on the running system. In effect, FFS transforms an underlying hardware vulnerability into a very powerful software-like vulnerability via three fundamental steps:

1. *Memory templating*: identifying physical memory locations in which an attacker can induce a bit flip using a given hardware vulnerability.
2. *Memory massaging*: steering targeted sensitive data towards the vulnerable physical memory locations.
3. *Exploitation*: triggering the hardware vulnerability to corrupt the intended data for exploitation.

In the remainder of this section, we detail each of these steps and outline FFS’s end-to-end attack strategy.

### 4.2.1 Memory Templating

The goal of the memory templating step is to fingerprint the hardware bit-flip patterns on the running system. This is necessary, since the locations of hardware bit flips are generally unknown in advance. This is specifically true in the case of Rowhammer; every (vulnerable) DRAM module is unique in terms of physical memory offsets with bit flips. In this step, the attacker triggers the hardware-specific vulnerability to determine which physical pages, and which offsets within those pages are vulnerable to bit flips. We call the combination of a vulnerable page and the offset a *template*.

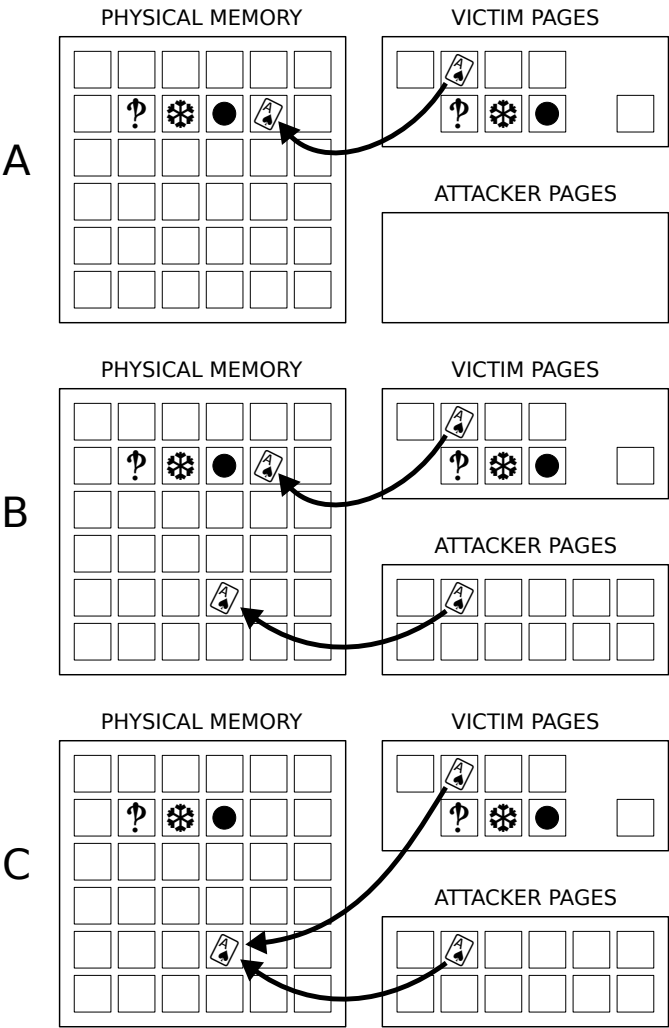
Probing for templates provides the attacker with knowledge of *usable* bit flips. Thanks to Flip Feng Shui, any template can potentially allow the attacker to exploit the hardware vulnerability over physical memory in a controlled way. The usefulness of such an exploit, however, depends on the direction of the bit flip (i.e., one-to-zero or zero-to-one), the page offset, and the contents of the target victim page. For each available template, the attacker can only craft a Flip Feng Shui primitive that corrupts the target data page with the given *flip* and *offset*. Hence, to surgically target the victim’s sensitive data of interest, the attacker needs to probe for matching templates by repeatedly exploiting the hardware vulnerability over a controlled physical page (i.e., mapped in her virtual address space). To perform this step efficiently, our own dFFS implementation relies on a variant of double-sided Rowhammer [111]. Rowhammer allows an attacker to induce bit flips in vulnerable memory locations by repeatedly reading from memory pages located in adjacent rows. We discuss the low-level details of the Rowhammer vulnerability and our implementation in Section 4.4.2.

### 4.2.2 Memory Massaging

To achieve bit flips over arbitrary contents of the victim’s physical memory, FFS abuses modern memory management patterns and features to craft a memory massaging primitive. Memory massaging allows the attacker to map a desired victim’s physical memory page into her own virtual memory address space in a controllable way.

Given a set of templates and the memory massaging primitive, an ideal version of FFS can corrupt any of the victim’s memory pages at an offset determined by the selected template.

While memory massaging may be nontrivial in the general case, it is surprisingly easy to abuse widely deployed memory deduplication features to craft practical FFS attacks that corrupt any of the victim’s memory pages with *known contents* (similar to our dFFS implementation). Intuitively, since memory deduplication merges



**Figure 4.1:** Memory deduplication can provide an attacker control over the layout of physical memory.

system-wide physical memory pages with the same contents, an attacker able to craft the contents of any of the victim’s memory pages can obtain a memory massaging primitive and map the target page into her address space.

Figure 4.1 shows how an attacker can control the physical memory location of a victim VM’s memory page. At first, the attacker needs to predict the contents of the victim VM’s page that she wants to control (Figure 4.1-A). Once the target page is identified, the attacker VM creates a memory page with the same contents as the

victim VM's memory page and waits for the memory deduplication system to scan both pages (Figure 4.1-B). Once the two physical pages (i.e., the attacker's and the victim's pages) are identified, the memory deduplication system returns one of the two pages back to the system, and the other physical page is used to back both the attacker and the victim's (virtual) pages. If the attacker's page is used to back the memory of the victim page, then, in effect, the attacker controls the physical memory location of the victim page (Figure 4.1-C).

There are additional details necessary to craft a memory massaging primitive using a real-world implementation of memory deduplication (e.g., KSM). Section 4.4.1 elaborates on such details and presents our implementation of memory massaging on Linux.

### 4.2.3 Exploitation

At this stage, FFS already provides the attacker with templated bit flips over the victim's physical memory pages with known (or predictable) contents. The exploitation surface is only subject to the available templates and their ability to reach interesting locations for the attacker. As we will see, the options are abundant.

While corrupting the memory state of running software of the victim is certainly possible, we have opted for a more straightforward, yet extremely powerful exploitation strategy. We consider an attacker running in a cloud VM and seeking to corrupt interesting contents in the page cache of a co-hosted victim VM. In particular, our dFFS implementation includes two exploits that corrupt sensitive file contents in the page cache in complete absence of software vulnerabilities:

1. Flipping SSH's `authorized_keys`: assuming the RSA public keys of the individuals accessing the victim VM are known, an attacker can use dFFS to induce an exploitable flip in their public keys, making them prone to factorization and breaking the authentication system.
2. Flipping apt's `sources.list` and `trusted.gpg`: Debian/Ubuntu's apt package management system relies on the `sources.list` file to operate daily updates and on the `trusted.gpg` file to check the authenticity of the updates via RSA public keys. Compromising these files allows an attacker to make a victim VM download and install arbitrary attacker-generated packages.

In preliminary experiments, we also attempted to craft an exploit to bit-flip SSH's `moduli` file containing Diffie-Hellman group parameters and eavesdrop on the victim VM's SSH traffic. The maximum group size on current distributions of OpenSSH is 1536. When we realized that an exploit targeting such 1536-bit parameters would require a nontrivial computational effort (see Appendix 4.9 for a formal analysis), we turned our attention to the two more practical and powerful exploits above.

In Section 4.3, we present a cryptanalysis of RSA moduli with a bit flip as a result of our attacks. In Section 4.4, we elaborate on the internals of the exploits,

and finally, in Section 4.5, we evaluate their success rate and time requirements in a typical cloud setting.

### 4.3 Cryptanalysis of RSA with Bit Flips

RSA [107] is a public-key cryptosystem: the sender encrypts the message with the public key of the recipient (consisting of an exponent  $e$  and a modulus  $n$ ) and the recipient decrypts the ciphertext with her private key (consisting of an exponent  $d$  and a modulus  $n$ ). This way RSA can solve the key distribution problem that is inherent to symmetric encryption. RSA can also be used to digitally sign messages for data or user authentication: the signing operation is performed using the private key, while the verification operation employs the public key.

Public-key cryptography relies on the assumption that it is computationally infeasible to derive the private key from the public key. For RSA, computing the private exponent  $d$  from the public exponent  $e$  is believed to require the factorization of the modulus  $n$ . If  $n$  is the product of two large primes of approximately the same size, factorizing  $n$  is not feasible. Common sizes for  $n$  today are 1024 to 2048 bits.

In this chapter we implement a fault attack on the modulus  $n$  of the victim: we corrupt a single bit of  $n$ , resulting in  $n'$ . We show that with high probability  $n'$  will be easy to factorize. We can then compute from  $e$  the corresponding value of  $d'$ , the private key, that allows us to forge signatures or to decrypt. We provide a detailed analysis of the expected computational complexity of factorizing  $n'$  in the following<sup>1</sup>.

RSA perform computations modulo  $n$ , where  $t$  is the bitlength of  $n$  ( $t = 1 + \lfloor \log_2 n \rfloor$ ). Typical values of  $t$  lie between 512 (export control) and 8192, with 1024 and 2048 the most common values. We denote the  $i$ th bit of  $n$  with  $n[i]$  ( $0 \leq i < t$ ), with the least significant bit (LSB) corresponding to  $n[0]$ . The unit vector is written as  $e_i$ , that is  $e_i[i] = 1$  and  $e_i[j] = 0$ , for  $j \neq i$ . The operation of flipping the  $i$ th bit of  $n$  results in  $n'$ , or  $n' = n \oplus e_i$ . Any integer can be written as the product of primes, hence  $n = \prod_{j=1}^s p_j^{\gamma_j}$ , where  $p_i$  are the prime factors of  $n$ ,  $\gamma_i$  is the multiplicity of  $p_i$  and  $s$  is the number of distinct prime factors. W.l.o.g. we assume that  $p_1 > p_2 > \dots > p_s$ .

In the RSA cryptosystem, the modulus  $n$  is the product of two odd primes  $p_1, p_2$  of approximate equal size, hence  $s = 2$ , and  $\gamma_1 = \gamma_2 = 1$ . The encryption operation is computed as  $c = m^e \bmod n$ , with  $e$  the public exponent, and  $m, c \in [0, n - 1]$  the plaintext respectively the ciphertext. The private exponent  $d$  can be computed as  $d = e^{-1} \bmod \lambda(n)$ , with  $\lambda(n)$  the Carmichael function, given by  $\text{lcm}(p_1, p_2)$ . The best known algorithm to recover the private key is to factorize  $n$  using the General Number Field Sieve (GNFS) (see e.g. [85]), which has complexity  $O(L_n[1/3, 1.92])$ , with

$$L_n[a, b] = \exp((b + o(1))(\ln n)^a (\ln \ln n)^{1-a}) .$$

<sup>1</sup> A similar analysis for Diffie-Hellman group parameters with bit flips can be found in Appendix 4.9.



For a 512-bit modulus  $n$ , Adrian et al. estimate that the cost is about 1 core-year [6]. The current record is 768 bits [70], but it is clear that 1024 bits is within reach of intelligence agencies [6].

If we flip the LSB of  $n$ , we obtain  $n' = n - 1$ , which is even hence  $n' = 2 \cdot n''$  with  $n''$  a  $t - 1$ -bit integer. If we flip the most significant bit of  $n$ , we obtain the odd  $t - 1$ -bit integer  $n'$ . In all the other cases we obtain an odd  $t$ -bit integer  $n'$ . We conjecture that the integer  $n''$  (for the LSB case) and the integers  $n'$  (for the other cases) have the same distribution of prime factors as a random odd integer. To simplify the notation, we omit in the following the LSB case, but the equations apply with  $n'$  replaced by  $n''$ .

Assume that an attacker can introduce a bit flip to change  $n$  into  $n'$  with as factorization  $n = \prod_{j=1}^{s'} p_j^{\tilde{\gamma}_j}$ . Then  $c' = m'^e \bmod n'$ . The Carmichael function can be computed as

$$\lambda(n') = \text{lcm} \left( \left\{ p_i^{\tilde{\gamma}_i - 1} \cdot (p_i - 1) \right\} \right).$$

If  $\gcd(e, \lambda(n')) = 1$ , the private exponent  $d'$  can be found as  $d' = e^{-1} \bmod \lambda(n')$ . For prime exponents  $e$ , the probability that  $\gcd(e, \lambda(n')) > 1$  equals  $1/e$ . For  $e = 3$ , this means that 1 in 3 attacks fails, but for the widely used value  $e = 2^{16} + 1$ , this is not a concern. With the private exponent  $d'$  we can decrypt or sign any message. Hence the question remains how to factorize  $n'$ . As it is very likely that  $n'$  is not the product of two primes of almost equal size, we can expect that factorizing  $n'$  is much easier than factorizing  $n$ .

Our conjecture implies that with probability  $2/\ln n'$ ,  $n'$  is prime and in that case the factorization is trivial. If  $n'$  is composite, the best approach is to find small factors (say up to 16 bits) using a greatest common divisor operation with the product of the first primes. The next step is to use Pollard's  $\rho$  algorithm (or Brent's variant) [85]: this algorithm can easily find factors up to 40...60 bits. A third step consist of Lenstra's Elliptic Curve factorization Method (ECM) [75]: ECM can quickly find factors up to 60...128 bits (the record is a factor of about 270 bits<sup>2</sup>). Its complexity to find the smallest prime factor  $p'_s$  is equal to  $O(L_{p'_s}[1/2, \sqrt{2}])$ . While ECM is asymptotically less efficient than GNFS (because of the parameter  $1/2$  rather than  $1/3$ ), the complexity of ECM depends on the size of the smallest prime factor  $p'_s$  rather than on the size of the integer  $n'$  to factorize. Once a prime factor  $p'_i$  is found,  $n'$  is divided by it, the result is tested for primality and if the result is composite, ECM is restarted with as argument  $n'/p'_i$ .

The complexity analysis of ECM depends on the number of prime factors and the distribution of the size of the second largest prime factor  $p'_2$ : it is known that its expected valued is  $0.210 \cdot t$  [71]. The Erdős–Kac theorem [45] states that the number  $\omega(n')$  of distinct prime factors of  $n'$  is normally distributed with mean and variance  $\ln \ln n'$ : for  $t = 1024$  the mean is about 6.56, with standard deviation 2.56. Hence it is unlikely that we have exactly two prime factors (probability 3.5%), and even

<sup>2</sup>[https://en.wikipedia.org/wiki/Lenstra\\_elliptic\\_curve\\_factorization](https://en.wikipedia.org/wiki/Lenstra_elliptic_curve_factorization)

less likely that they are of approximate equal size. The probability that  $n'$  is prime is equal to 0.28%. The expected size of the second largest prime factor  $p'_2$  is 215 bits and the probability that it has less than 128 bits is 0.26 [71]. In this case ECM should be very efficient. For  $t = 2048$ , the probability that  $n'$  is prime equals 0.14%. The expected size of the second largest prime factor  $p'_2$  is 430 bits; the probability that  $p'_2$  has less than 228 bits is 0.22 and the probability that it has less than 128 bits is about 0.12. Similarly, for  $t = 4096$ , the expected size of the second largest prime factor  $p'_2$  is 860 bits. The probability that  $p'_2$  has less than 455 bits is 0.22.

The main conclusion is that *if  $n$  has 1024-2048 bits, we can expect to factorize  $n'$  efficiently with a probability of 12 – 22% for an arbitrary bit flip, but larger moduli should also be feasible.* As we show in Section 4.5, given a few dozen templates, we can easily factorize any 1024 bit to 4096 bit modulus with one (or more) of the available templates.

## 4.4 Implementation

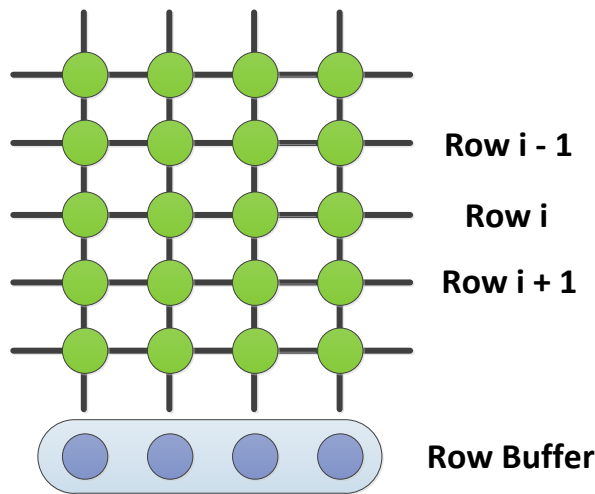
To implement dFFS reliably on Linux, we need to understand the internals of two kernel subsystems, kernel same-page merging [9] (KSM) and transparent huge pages [8], and the way they interact with each other. After discussing them and our implementation of the Rowhammer exploit (Sections 4.4.1, 4.4.2, and 4.4.3), we show how we factorized corrupted RSA moduli in Section 4.4.4 before summarizing our end-to-end attacks in Section 4.4.5.

### 4.4.1 Kernel Same-page Merging

KSM, the Linux implementation of memory deduplication, uses a kernel thread that periodically scans memory to find memory pages with the same contents that are candidates for merging. It then keeps a single physical copy of a set of candidate pages, marks it read-only, and updates the page-table entries of all the other copies to point to it before releasing their physical pages to the system.

KSM keeps two red-black trees, termed “stable” and “unstable”, to keep track of the merged and candidate pages. The merged pages reside in the stable tree while the candidate contents that are not yet merged are in the unstable tree. KSM keeps a list of memory areas that are registered for deduplication and goes through the pages in these areas in the order in which they were registered. For each page that it scans, it checks if the stable tree already contains a page with the same contents. If so, it updates the page-table entry for that page to have it point to the physical page in the stable tree and releases the backing physical page to the system. Otherwise, it searches the unstable tree for a match and if it finds one, promotes the page to the stable tree and updates *the page-table entry of the match to make it point to this page*. If no match is found in either one of the trees, the page is added to the unstable tree. After going through all memory areas, KSM dumps the unstable tree before starting again. Further details on the internals of KSM can be found in [9].





**Figure 4.3:** DRAM's internal organization.

#### 4.4.2 Rowhammer inside KVM

Internally, DRAM is organized in rows. Each row provides a number of physical cells that store memory bits. For example, in an x86 machine with a single DIMM, each row contains 1,048,576 cells that can store 128 kB of data. Each row is internally mapped to a number of chips on the DIMM as shown in Figure 4.2.

Figure 4.3 shows a simple organization of a DRAM chip. When the processor reads a physical memory location, the address is translated to an offset on row  $i$  of the DRAM. Depending on the offset, the DRAM selects the proper chip. The selected chip then copies the contents of its row  $i$  to the row buffer. The contents at the correct offset within the row buffer is then sent on the bus to the processor. The row buffer acts as a cache: if the selected row is already in the row buffer, there is no need to read from the row again.

Each DRAM cell is built using a transistor and a capacitor. The transistor controls whether the contents of the cell is accessible, while the capacitor can hold a charge which signifies whether the stored content is a high or low bit. Since capacitors leak charge over time, the processor sends refresh commands to DIMM rows in order to recharge their contents. On top of the refresh commands, every time a row is read by the processor, the chip also recharges its cells.

As DRAM components have become smaller, they keep a smaller charge to signify stored contents. With a smaller charge, the error margin for identifying whether the capacitor is charged (i.e., the stored value) is also smaller. Kim et al. [69] showed that the smaller error margin, in combination with unexpected charge exchange between cells of different rows, can result in the cell to “lose” its content. To trigger this DRAM reliability issue, an attacker needs fast activations of DRAM rows which

causes a cell in adjacent rows to lose enough charge so that its content is cleared. Note that due to the row buffer, at least two rows need to activate one after the other in a tight loop for Rowhammer to trigger. If only one row is read from, the reads can be satisfied continually from the row buffer, without affecting the row charges in the DRAM cells.

**Double-sided Rowhammer.** Previous work [111] reported that if these two “aggressor” rows are selected in a way that they are one row apart (e.g., row  $i - 1$  and  $i + 1$  in Figure 4.3), the chances of charge interaction between these rows and the row in the middle (i.e., row  $i$ ) increases, resulting in potential bit flips in that row. This variant of Rowhammer is named double-sided Rowhammer. Apart from additional speed for achieving bit flips, it provides additional reliability by isolating the location of most bit flips to a certain (victim) row.

To perform double-sided Rowhammer inside KVM, we need to know the host physical addresses inside the VM. This information is, however, not available in the guest: guest physical addresses are mapped to host virtual addresses which can be mapped to any physical page by the Linux kernel. Similar to [58], we rely on transparent huge pages [8] (THP). THP is a Linux kernel feature that runs in the background and merges virtually contiguous normal pages (i.e., 4 kB pages) into huge pages (i.e., 2 MB pages) that rely on contiguous pieces of physical memory. THP greatly reduces the number of page-table entries in suitable processes, resulting in fewer TLB<sup>3</sup> entries. This improves performance for some workloads.

THP is another (weak) form of memory massaging: it transparently allows the attacker control over how the system maps guest physical memory to host physical memory. Once the VM is started and a certain amount of time has passed, THP will transform most of the VM’s memory into huge pages. Our current implementation of dFFS runs entirely in the userspace of the guest and relies on the default-on THP feature of both the host and the guest. As soon as the guest boots, dFFS allocates a large buffer with (almost) the same size as the available memory in the guest. The THP in the host then converts guest physical addresses into huge pages and the THP in the guest turns the guest virtual pages backing dFFS’s buffer into huge pages as well. As a result, dFFS’s buffer will largely be backed by huge pages all the way down to host physical memory.

To make sure that the dFFS’s buffer is backed by huge pages, we request the guest kernel to align the buffer at a 2 MB boundary. This ensures that if the buffer is backed by huge pages, it starts with one: on the x86\_64 architecture, the virtual and physical huge pages share the lowest 20 bits, which are zero. The same applies when transitioning from the guest physical addresses to host physical addresses. With this knowledge, dFFS can assume that the start of the allocated buffer is the start of a memory row, and since multiple rows fit into a huge page, it can successively

<sup>3</sup>TLB or translation lookaside buffer is a general term for processor caches for page-table entries to speed up the virtual to physical memory translation

perform double-sided Rowhammer on these rows. To speed up our search for bit flips during double-sided Rowhammer on each two rows, we rely on the row-conflict side channel for picking the hammering addresses within each row [101]. We further employed multiple threads to amplify the Rowhammer effect.

While THP provides us with the ability to efficiently and reliably induce Rowhammer bit flips, it has unexpected interactions with KSM that we will explore in the next section.

### 4.4.3 Memory Massaging with KSM

In Section 4.2.2, we discussed the operational semantics of KSM. Here we detail some of its implementation features that are important for dFFS.

#### Interaction with THP

As we discussed earlier, KSM deduplicates memory pages with the same contents as soon as it finds them. KSM currently does not support deduplication of huge pages, but what happens when KSM finds matching contents within huge pages?

A careful study of the KSM shows that KSM always prefers reducing memory footprint over reducing TLB entries; that is, KSM breaks down huge pages into smaller pages if there is a small page inside with similar contents to another page.

This specific feature is important for an efficient and reliable implementation of dFFS, but has to be treated with care. More specifically, we can use huge pages as we discussed in the previous section for efficient and reliable double-sided Rowhammer, while retaining control over which victim page we should map in the middle of our target (vulnerable) huge page.

KSM, however, can have undesired interactions with THP from dFFS's point of view. If KSM finds pages in the attacker VM's memory that have matching contents, it merges them with each other or with a page from a previously started VM. In these situations, KSM breaks THP by releasing one of its smaller pages to the system. To avoid this, dFFS uses a technique to avoid KSM during its templating phase. KSM takes a few tens of seconds to mark the pages of dFFS's VM as candidates for deduplication. This gives dFFS enough time to allocate a large buffer with the same size as the VM's available memory (as mentioned earlier) and write unique integers at a pre-determined location within each (small) page of this buffer as soon as its VM boots. The entropy present within dFFS's pages then prohibits KSM to merge these pages which in turn avoids breaking THP.

#### On dFFS Chaining

Initially, we planned on chaining memory massaging primitive and FFS to induce an arbitrary number of bit flips at many desired locations of the victim's memory page. After using the first template for the first bit flip, the attacker can write to the merged memory page to trigger a copy-on-write event that ultimately unmerges the

two pages (i.e., the attacker page from the victim page). At this stage, the attacker can use dFFS again with a new template to induce another bit flip.

However, the implementation of KSM does not allow this to happen. During the copy-on-write event, the victim's page remains in the stable tree, even if it is the only remaining page. This means that subsequent attempts for memory massaging results in the victim page to control the location of physical memory, disabling the attacker's ability for chaining FFS attacks.

Even so, based on our single bit flip cryptanalysis on public keys and our evaluation in Section 4.5, chaining is not necessary for performing successful end-to-end attacks with dFFS.

#### 4.4.4 Attacking Weakened RSA

For the two attacks in this chapter, we generate RSA private keys, i.e., the private exponents  $d'$  corresponding to corrupted moduli  $n'$  (as described in Section 4.3). We use  $d'$  to compromise two applications: OpenSSH and GPG.

Although the specifics of the applications are very different, the pattern to demonstrate each attack is the same and as follows:

1. Obtain the file containing the RSA public key  $(n, e)$ . This is application-specific, but due to the nature of public key cryptosystems, generally unprotected. We call this the input file.
2. Using the memory templating step of Section 4.2.1 we obtain a list of templates that we are able to flip within a physical page. We flip bits according to the target templates to obtain corrupted keys. For every single bit-flip, we save a new file. We call these files the corrupted files. According to the templating step, dFFS has the ability to create any of these corrupted files in the victim by flipping a bit in the page cache.
3. One by one, we now read the (corrupted) public keys for each corrupted file. If the corrupted file is parsed correctly *and* the public key has a changed modulus  $n' \neq n$  and the same  $e$ , this  $n'$  is a candidate for factorization.
4. We start factorizations of all  $n'$  candidates found in the previous step. As we described in Section 4.3, the best known algorithm for our scenario is ECM that finds increasingly large factor in an iterative fashion. We use the Sage [38] implementation of ECM for factorizing  $n'$ . We invoke an instance of ECM per available core for each corrupted key with a 1 hour timeout (all available implementations of ECM run with a single thread).
5. For all successful factorizations, we compute the private exponent  $d'$  corresponding to  $(n', e)$  and generate the corresponding private key to the corrupted public key. How to compute  $d'$  based on the factorization of  $n'$  is described in Section 4.3. We can then use the private key with the unmodified application. This step is application-specific and we will discuss it for our case studies shortly.

We now describe our end-to-end attacks that put all the pieces of dFFS together using two target applications: OpenSSH and GPG.

#### 4.4.5 End-to-end Attacks

**Attacker model.** The attacker owns a VM co-hosted with a victim VM on a host with DIMMs susceptible to Rowhammer. We further assume that memory deduplication is turned on—as is common practice in public cloud settings [11]. The attacker has the ability to use the memory deduplication side-channel to fingerprint low-entropy information, such as the IP address of the victim VM, OS/library versions, and the usernames on the system (e.g., through `/etc/passwd` file in the page cache) as shown by previous work [63; 97; 123]. The attacker’s goal is to compromise the victim VM without relying on any software vulnerability. We now describe how this model applies with dFFS in two important and widely popular applications.

##### OpenSSH

One of the most commonly used authentication mechanisms allowed by the OpenSSH daemon is an RSA public key. By adding a user’s RSA public key to the SSH `authorized_keys` file, the corresponding private key will allow login of that user without any other authentication (such as a password) in a default setting. The public key by default includes a 2048 bit modulus  $n$ . The complete key is a 372-byte long base64 encoding of  $(n, e)$ .

The attacker can initiate an SSH connection to the victim with a correct victim username and an arbitrary private key. This interaction forces OpenSSH to read the `authorized_keys` file, resulting in this file’s contents getting copied into the page cache at the right time as we discussed in Section 4.4.1. Public key cryptosystems by definition do not require public keys to be secret, therefore we assume an attacker can obtain a victim public key. For instance, GitHub makes the users’ submitted SSH public keys publicly available [52].

With the victim’s public key known and in the page cache, we can initiate dFFS for inducing a bit flip. We cannot flip just any bit in the memory page caching the `authorized_keys`; some templates *will* break the base64 encoding, resulting in a corrupted file that OpenSSH does not recognize. Some flips, however, decode to a valid  $(n', e)$  key that we can factorize. We report in Section 4.5 how many templates are available on average for a target public key.

Next, we use a script with the PyCrypto RSA cryptographic library [79] to operate on the corrupted public keys. This library is able to read and parse OpenSSH public key files, and extract the RSA parameters  $(n, e)$ . It can also generate RSA keys with specific parameters and export them as OpenSSH public  $(n', e)$  and private  $(n', d')$  keys again. All the attacker needs to do is factorize  $n'$  as we discussed in Section 4.4.4.



Once we know the factors of  $n'$ , we generate the private key  $(n', d')$  that can be used to login to the victim VM using an unmodified OpenSSH client.

## GPG

The GNU Privacy Guard, or GPG, is a sophisticated implementation of, among others, the RSA cryptosystem. It has many applications in security, one of which is the verification of software distributions by verifying signatures using trusted public keys. This is the larger application we intend to subvert with this attack.

Specifically, we target the apt package distribution system employed by Debian and Ubuntu distribution for software installation and updates. apt verifies package signatures after download using GPG and trusted public keys stored in `trusted.gpg`. It fetches the package index from sources in `sources.list`.

Our attack first steers the victim to our malicious repository. The attacker can use dFFS to achieve this goal by inducing a bit flip in the `sources.list` file that is present in the page cache after an update. `sources.list` holds the URL of the repositories that are used for package installation and update. By using a correct template, the attacker can flip a bit that results in a URL that she controls. Now, the victim will seek the package index and packages at an attacker-controlled repository.

Next, we use our exploit to target the GPG trusted keys database. As this file is part of the software distribution, the stock contents of this file is well-known and we assume this file is unchanged or we can guess the new changes. (Only the pages containing the keys we depend on need be either unchanged or guessed.) This file resides in the page cache every time the system tries to update as a result of a daily cron job, so in this attack, no direct interaction with the victim is necessary for bringing the file in the page cache. Our implicit assumption is that this file remains in the page cache for the next update iteration.

Similar to OpenSSH, we apply bit flip mutations in locations where we can induce bit flips according to the memory templating step. As a result, we obtain the corrupted versions of this file, and each time check whether GPG will still accept this file as a valid keyring and that one of the RSA key moduli has changed as a result of our bit flip. Extracting the key data is done with the GPG `--list-keys --with-key-data` options.

For every bit-flip location corresponding to a corrupted modulus that we can factorize, we pick one of these mutations and generate the corresponding  $(n', d')$  RSA private key, again using PyCrypto. We export this private key using PyCrypto as PEM formatted key and use `pem2openpgp` [51] to convert this PEM private key to the GPG format. Here we specify the usage flags to include signing and the same generation timestamp as the original public key. We can then import this private key for use for signing using an unmodified GPG.

It is important that the Key ID in the private keyring match with the Key ID in the `trusted.gpg` file. This Key ID is not static but is based on a hash computed from the public key data, a key generation timestamp, and several other fields. In

order for the Key ID in the private keyring to match with the Key ID in the public keyring, these fields have to be identical and so the setting of the creation timestamp is significant.

One significant remark about the Key ID changing (as a result of a bit flip) is that this caused the self-signature on the public keyring to be ignored by GPG! The signature contains the original Key ID, but it is now attached to a key with a different ID due to the public key mutation. As a result, *GPG ignores the attached signature as an integrity check of the bit-flipped public key and the self-signing mechanism fails to catch our bit flip*. The only side-effect is harmless to our attack – GPG reports that the trusted key is not signed. `apt` ignores this without even showing a warning. After factorizing the corrupted public key modulus, we successfully verified that the corresponding private key can generate a signature that verifies against the bit-flipped public key stored in the original `trusted.gpg`.

We can now sign our malicious package with the new private key and the victim will download and install the new package without a warning.

## 4.5 Evaluation

We evaluated dFFS to answer the following three key questions:

- What is the success probability of the dFFS attack?
- How long does the dFFS attack take?
- How much computation power is necessary for a successful dFFS attack?

We used the following methodology for our evaluation. We first used a Rowhammer testbed to measure how many templates are available in a given segment of memory and how long it takes us to find a certain template. We then executed the end-to-end attacks discussed in Section 4.4.5 and report on their success rate and their start-to-finish execution time. We then performed an analytical large-scale study of the factorization time, success probability, and computation requirements of 200 RSA public keys for each of the 1024, 2048 and 4096-bit moduli with 50 bit flips at random locations (i.e., 30,000 bit flipped public keys in total).

We used the following hardware for our Rowhammer testbed and for the cluster that we used to conduct our factorization study:

**Rowhammer testbed.** Intel Haswell i7-4790 4-core processor on an Asus H97-Pro mainboard with 8 GB of DDR3 memory.

**Factorization cluster.** Up to 60 nodes, each with two Intel Xeon E5-2630 8-core processors with 64 GB of memory.

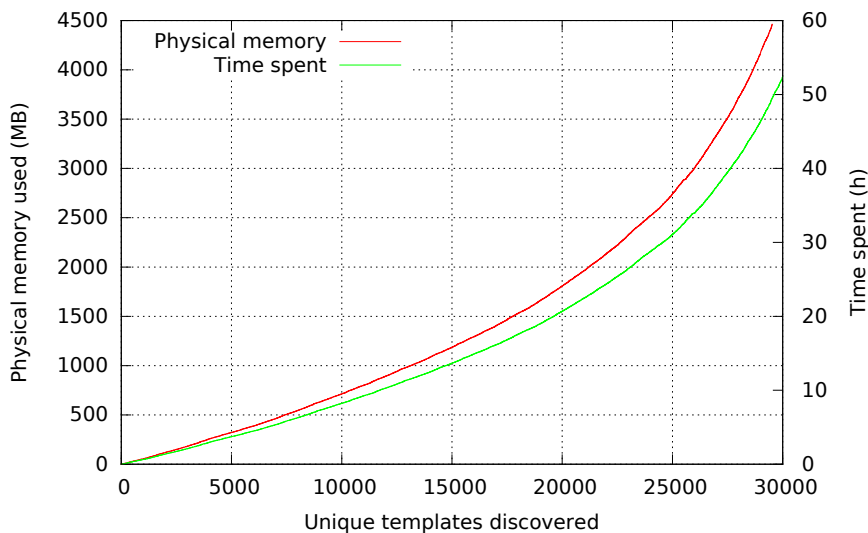


Figure 4.4: Required time and memory for templating.

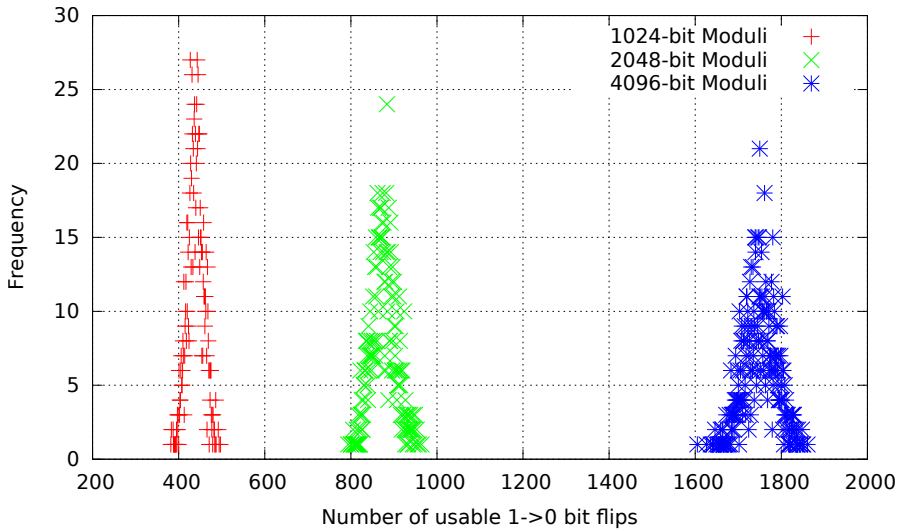
#### 4.5.1 dFFS on the Rowhammer Testbed

**Memory templating.** Our current implementation of Rowhammer takes an average of 10.58 seconds to complete double-sided Rowhammer for each target row. Figure 4.4 shows the amount of time and physical memory that is necessary for discovering a certain number of templates. Note that, in our testbed, we could discover templates for almost any bit offset (i.e., 29,524 out of 32,768 possible templates). Later, we will show that we only need a very small fraction of these templates to successfully exploit our two target programs.

**Memory massaging.** dFFS needs to wait for a certain amount of time for KSM to merge memory pages. KSM scans a certain number of pages in each waking period. On the default version of Ubuntu, for example, KSM scans 100 pages every 20 milliseconds (i.e., 20 MB). Recent work [20] shows that it is possible to easily detect when a deduplication pass happens, hence dFFS needs to wait at most the sum of memory allocated to each co-hosted VM. For example, in our experiments with one attacker VM and one victim VM each with 2 GB of memory, KSM takes at most around 200 seconds for a complete pass.

#### 4.5.2 The SSH Public Key Attack

Figure 4.5 shows the number of possible templates to perform the dFFS attack on the SSH `authorized_keys` file with a single randomly selected RSA public key, for 1024, 2048 and 4096-bit public keys. For this experiment, we assumed 1-to-



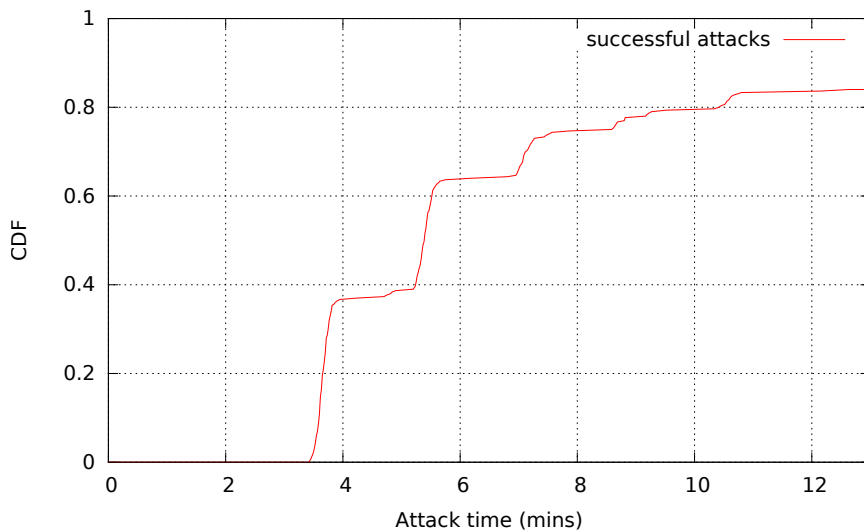
**Figure 4.5:** Number of usable 1-to-0 bit flips usable in the SSH `authorized_keys` file for various modulus sizes.

0 bit flips since they are more common in our testbed. For DRAM chips that are susceptible to frequent 0-to-1 bit flips, these numbers double. For our experiment we focused on 2048-bit public keys as they are the default length as generated by the `ssh-keygen` command.

To demonstrate the working end-to-end attack, measure its reliability, and measure the elapsed time distribution, we automatically performed the SSH attack 300 times from an attacker VM on a victim VM, creating the keys and VMs from scratch each time. Figure 4.6 shows the CDF of successful attacks with respect to the time they took. In 29 cases (9.6%), the Rowhammer operation did not change the modulus at all (the attacker needs to retry). In 19 cases (6.3%), the Rowhammer operation changed the modulus other than planned. The remaining 252 (84.1%) were successful the first time. All the attacks finished within 12.6 minutes with a median of 5.3 minutes.

### 4.5.3 The Ubuntu/Debian Update Attack

We tried factorizing the two bit-flipped 4096 bit **Ubuntu Archive Automatic Signing** RSA keys found in the `trusted.gpg` file. Out of the 8,192 trials (we tried both 1-to-0 and 0-to-1 flips), we could factorize 344 templates. We also need to find a bit flip in the URL of the Ubuntu or Debian update servers (depending on the target VM's distribution) in the page cache entry for `apt's sources.list` file. For `ubuntu.com`, 29 templates result in a valid domain name, and for `debian.org`, 26 templates result in a valid domain name. Table 4.1 shows examples of domains that



**Figure 4.6:** CDF of successful automatic SSH attacks.

are one bit flip away from `ubuntu.com`.

Performing the update attack on our Rowhammer testbed, we could trigger a bit flip in the page cache entry of `sources.list` in 212 seconds, converting `ubuntu.com` to `ubunvu.com`, a domain which we control. Further, we could trigger a bit flip in the page cache entry of `trusted.gpg` that changed one of the RSA public keys to one that we had pre-computed a factorization in 352 seconds. At this point, we manually sign the malicious package with our GPG private key that corresponds to the mutated public key. When the victim then updates the package database and upgrades the packages, the malicious package is downloaded and installed without warning. Since the current version of dFFS runs these steps sequentially, the entire end-to-end attack took 566 seconds. We have prepared a video of this attack which is available at: <https://vusec.net/projects/flip-feng-shui>

Growingly concerned about the impact of such practical attacks, we conservatively registered all the possible domains from our Ubuntu/Debian list.

#### 4.5.4 RSA Modulus Factorization

Figure 4.7 shows the average probability of successful factorizations based on the amount of available compute hours. We generated this graph using 200 randomly generated 2048-bit RSA keys, each with a bit flip in 50 distinct trials (i.e., 10,000 keys, each with a bit flip). For this experiment, we relied on the ECM factorization tool, discussed in Section 4.3, and varied its user-controlled timeout parameter between one second and one hour. For example, with a timeout of one second for a

**Table 4.1:** Examples of domains that are one bit flip away from ubuntu.com that we purchased.

ubuftu.com	ubunt5.com	ubunte.com
ubunu.u.com	ubunvu.com	ubunpu.com
ubun4u.com	ubuntw.com	ubuntt.com

key with a bit flip, we either timeout or the factorization succeeds immediately. In both cases, we move on to the next trial of the same key with a different bit flip.

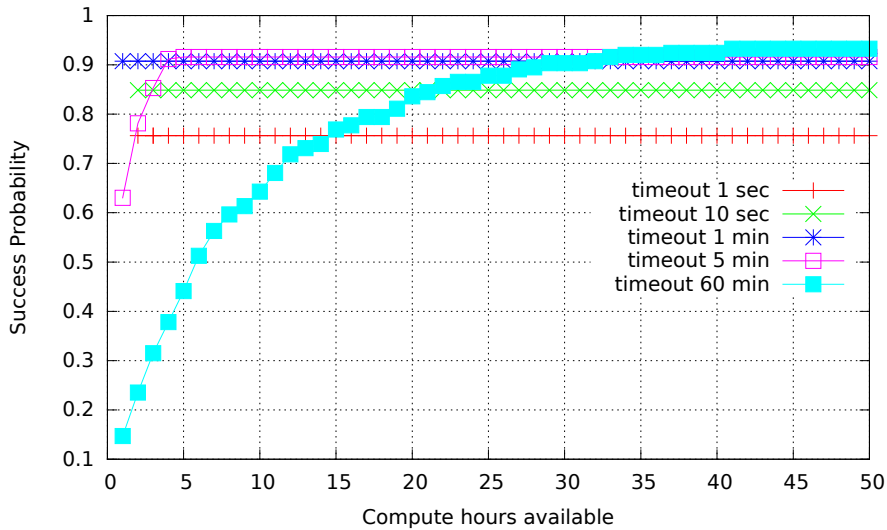
This graph shows that, with 50 bit flips, the average factorization success probability is between 0.76 for a timeout of one second and 0.93 for a timeout of one hour. Note that, for example, with a timeout of one second, we can try 50 templates in less than 50 seconds, while achieving a successful factorization in as many as 76% of the public keys. A timeout of one minute provides a reasonable trade-off and can achieve a success rate of 91% for 2048-bit RSA keys.

Figure 4.8 shows the cumulative success probability of factorization as more templates become available for 1024-bit, 2048-bit and 4096-bit keys. For 4096-bit keys, we need around 50 templates to be able to factorize a key with high probability (0.85) with a 1-hour timeout. With bit-flipped 2048-bit RSA public keys, with only 48 templates, we achieved a success probability of 0.99 with a 1 hour timeout. This proves that for 2048-bit keys (ssh-keygen’s default), *only a very small fraction of the templates from our testbed is necessary for a successful factorization*. For 1024-bit keys, we found a successful factorization for all keys after just 32 templates.

Some DRAM modules may only have a small number of bit flips [69], so an interesting question is: what is the chance of achieving a factorization using only a single template? Figure 4.9 answers this question for 1024-bit, 2048-bit and 4096-bit moduli separately. To interpret the figure, fix a point on the horizontal axis: this is the probability of a successful factorization using a single bit flip within 1 hour. Now read the corresponding value on the vertical axis, which shows the probability that a public key follows this success rate. For example, on average, 15% of 2048-bit RSA public keys can be factored using only a single bit flip with probability 0.1. As is expected, the probability to factor 4096-bit keys with the same 1-hour timeout is lower, and for 1024-bit keys higher. The fact that the distributions are centered around roughly 0.22, 0.11, and 0.055 are consistent with our analytical results in 4.3, which predict the factorization cost is linear in the bitlength of the modulus.

## 4.6 Mitigations

Mitigating Flip Feng Shui is not straightforward as hardware reliability bugs become prevalent. While there is obviously need for new testing methods and certification on the hardware manufacturer’s side [7], software needs to adapt to fit Flip Feng Shui in its threat model. In this section, we first discuss concrete mitigations against



**Figure 4.7:** Compute power and factorization timeout trade-off for 2048-bit RSA keys.

dFFS before suggesting how to improve software to counter FFS attacks.

#### 4.6.1 Defending against dFFS

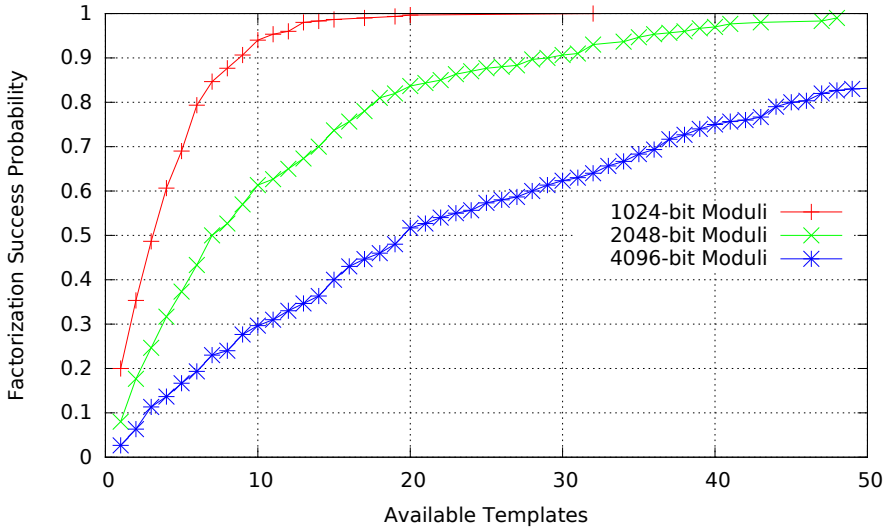
We discuss both hardware and software solutions for defending against dFFS.

##### In Hardware

We recommend DRAM consumers perform extensive Rowhammer testing [3] to identify vulnerable DRAM modules. These DRAM modules should be replaced, but if this is not possible, reducing DRAM refresh intervals (e.g., by half) may be sufficient to protect against Rowhammer [111]. However, this also reduces DRAM performance and consumes additional power.

Another option is to rely on memory with error-correcting codes (ECC) to protect against single bit flips. Unfortunately, we have observed that Rowhammer can occasionally induce multiple flips in a single 64-bit word confirming the findings of the original Rowhammer paper [69]. These multi-flips can cause corruption even in presence of ECC. More expensive multi-ECC DIMMs can protect against multiple bit flips, but it is still unclear whether they can completely mitigate Rowhammer.

A more promising technology is *directed row refresh*, which is implemented in low-power DDR4 [10] (LPDDR4) and some DDR4 implementations. LPDDR4 counts the number of activations of each row and, when this number grows beyond a particular threshold, it refreshes the adjunct rows, preventing cell charges from falling below the error margin. Newer Intel processors support a similar feature for



**Figure 4.8:** CDF of success rate with increasing templates.

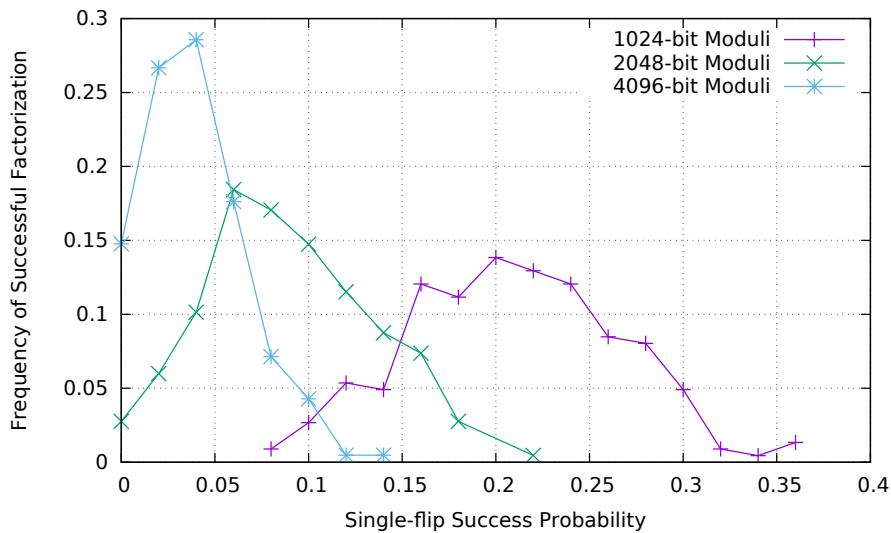
DDR3, but require compliant DIMMs. While these fixes mitigate Rowhammer, replacing most of current DDR3 deployments with LPDDR4 or secure DDR4 DIMMs (some DDR4 DIMMs are reported to be vulnerable to Rowhammer [1]), is not economically feasible as it requires compatible mainboards and processors. As a result, a software solution is necessary for mitigating Rowhammer in current deployments.

### In Software

The most obvious mitigation against dFFS is disabling memory deduplication. In fact, this is what we recommend in security-sensitive environments. Disabling memory deduplication completely, however, wastes a substantial amount of physical memory that can be saved otherwise [9; 103; 118].

Previous work [20] showed that deduplicating zero pages alone can retain between 84% and 93% of the benefits of full deduplication in a browser setting. Limiting deduplication to zero pages and isolating their Rowhammer-prone surrounding rows was our first mitigation attempt. To understand whether zero-page deduplication retains sufficient memory saving benefits in a cloud setting, we performed a large-scale memory deduplication study using 1,011 memory snapshots of *different* VMs from community VM images of Windows Azure [105]. Table 4.2 presents our results. Unfortunately, zero-page deduplication only saves 46% of the potential 79%. This suggests that deduplicating zero pages alone is insufficient to eradicate the wasteful redundancy in current cloud deployments. Hence, we need a better strategy that can retain the benefits of full memory deduplication without resulting in a memory massaging primitive for the attackers.





**Figure 4.9:** Probability mass function of successful factorizations with one flip.

**A strawman design** A possible solution is to rely on a deduplication design that, for every merge operation, randomly allocates a new physical page to back the existing duplicate pages. When merge operations with existing shared pages occur, such design would need to randomly select a new physical page and update all the page-table mappings for all the sharing parties.

This strawman design eliminates the memory massaging primitive that is necessary for dFFS under normal circumstances. However, this may be insufficient if an attacker can find different primitives to control the physical memory layout. For example, the attacker’s VM can corner the kernel’s page allocator into allocating pages with predictable patterns if it can force the host kernel into an out-of-memory (OOM) situation. This is not difficult if the host relies on over-committed memory to pack more VMs than available RAM, a practice which is common in cloud settings and naturally enabled by memory deduplication. For example, the attacker can trigger a massive number of unmerge operations and cause the host kernel to approach an OOM situation. At this point, the attacker can release vulnerable memory pages to the allocator, craft a page with the same contents as the victim page, and wait for a merge operation. Due to the near-OOM situation, the merge operation happens almost instantly, forcing the host kernel to predictably pick one of the previously released vulnerable memory pages (i.e., templates) to back the existing duplicate pages (the crafted page and the victim page). At this stage, the attacker has again, in effect, a memory massaging primitive.

**A better design** To improve on the strawman design, the host needs to ensure enough memory is available not to get cornered into predictable physical memory

**Table 4.2:** Memory savings with different dedup strategies.

Strategy	Required memory	Savings
No dedup	506 GB	0%
Zero-page dedup	271 GB	46%
Full dedup	108 GB	79%

reuse patterns. Given a desired level of entropy  $h$ , and the number of merged pages  $m_i$  for the  $i$ th VM, the host needs to ensure  $A = 2^h + \text{Max}(m_i)$  memory pages are available or can easily become available (e.g., page cache) to the kernel's page allocator at all times. With an adequate choice of  $h$ , it may become difficult for an attacker to control the behavior of the memory deduplication system. We have left the study of the right parameters for  $h$  and the projected  $A$  for real systems to future work. We also note that balancing entropy, memory, and performance when supporting a truly random and deduplication-enabled physical memory allocator is challenging, and a promising direction for future work.

#### 4.6.2 Mitigating FFS at the Software Layer

The attacks presented in this chapter provide worrisome evidence that even the most security-sensitive software packages used in production account for no attacker-controlled bit flips as part of their threat model. While there is certainly room for further research in this direction, based on our experience, we formulate a number of suggestions to improve current practices:

- Security-sensitive information needs to be checked for integrity in software right before use to ensure the window of corruption is small. In all the cases we analyzed, such integrity checks would be placed on a slow path with essentially no application performance impact.

Certificate chain formats such as X.509 are automatically integrity checked as certificates are always signed [30]. This is a significant side benefit of a certification chain with self-signatures.

- The file system, due to the presence of the page cache, should not be trusted. Sensitive information on stable storage should include integrity or authenticity information (i.e., a security signature) for verification purposes. In fact, this simple defense mechanism would stop the two dFFS attacks that we presented in this chapter.
- Low-level operating system optimizations should be included with extra care. Much recent work [19; 20; 56; 80; 130] shows that benign kernel optimizations such as transparent huge pages, vsyscall pages, and memory deduplication can become dangerous tools in the hands of a capable attacker. In the case of FFS, any feature that allows an untrusted entity to control the layout or reuse of

data in physical memory may provide an attacker with a memory massaging primitive to mount our attacks.

## 4.7 Related Work

We categorize related work into three distinct groups discussed below.

### 4.7.1 Rowhammer Exploitation

Pioneering work on the Rowhammer bug already warned about its potential security implications [69]. One year later, Seaborn published the first two concrete Rowhammer exploits, in the form of escaping the Google Native Client (NaCl) sandbox and escalating local privileges on Linux [111]. Interestingly, Seaborn's privilege escalation exploit relies on a weak form of memory massaging by probabilistically forcing a OOMing kernel to reuse physical pages released from user space. dFFS, in contrast, relies on a deterministic memory massaging primitive to map pages from co-hosted VMs and mount fully reliable attacks. In addition, while mapping pages from kernel space for local privilege escalation is possible, dFFS enables a much broader range of attacks over nearly arbitrary physical memory.

Furthermore, Seaborn's exploits relied on Intel x86's CLFLUSH instruction to evict a cache line from the CPU caches in order to read directly from DRAM. For mitigation purposes, CLFLUSH was disabled in NaCl and the same solution was suggested for native CPUs via a microcode update. In response to the local privilege exploit, Linux disabled unprivileged access to virtual-to-physical memory mapping information (i.e., `/proc/self/pagemap`) used in the exploit to perform double-sided Rowhammer. Gruss et al. [58], however, showed that it is possible to perform double-sided Rowhammer from the browser, without CLFLUSH, and without pagemap, using cache eviction sets and transparent huge pages (THP). dFFS relies on nested THP (both in the host and in the guest) for reliable double-sided Rowhammer. In our previous work [20], we took the next step and implemented the first reliable Rowhammer exploit in the Microsoft Edge browser. Our exploit induces a bit flip in the control structure of a JavaScript object for pivoting to an attacker-controlled counterfeit object. The counterfeit object provides the attackers with arbitrary memory read and write primitives inside the browser.

All the attacks mentioned above rely on one key assumption: the attacker already *owns* the physical memory of the victim to make Rowhammer exploitation possible. In this chapter, we demonstrated that, by abusing modern memory management features, it is possible to completely lift this assumption with alarming consequences. Using FFS, an attacker can seize control of nearly arbitrary physical memory in the software stack, for example compromising co-hosted VMs in complete absence of software vulnerabilities.

### 4.7.2 Memory Massaging

Sotirov [121] demonstrates the power of controlling virtual memory allocations in JavaScript, bypassing many protections against memory errors with a technique called Heap Feng Shui. Mandt [83] demonstrates that it is possible to control reuse patterns in the Windows 7 kernel heap allocator in order to bypass the default memory protections against heap-based attacks in the kernel. Inspired by these techniques, our Flip Feng Shui demonstrates that an attacker abusing benign and widespread memory management mechanisms allows a single bit flip to become a surprisingly dangerous attack primitive over physical memory.

Memory spraying techniques [50; 65; 104; 109] allocate a large number of objects in order to make the layout of memory predictable for exploitation purposes, similar, in spirit, to FFS. Govindavajhala and Appel [54] sprayed the entire memory of a machine with specially-crafted Java objects and showed that 70% of the bit flips caused by rare events cosmic rays and such will allow them to escape the Java sandbox. This attack is by its nature probabilistic and, unlike FFS, does not allow for fully controllable exploitation.

Memory deduplication side channels have been previously abused to craft increasingly sophisticated information disclosure attacks [11; 20; 56; 63; 97; 123]. In this chapter, we demonstrate that memory deduplication has even stronger security implications than previously shown. FFS can abuse memory deduplication to perform attacker-controlled page-table updates and craft a memory massaging primitive for reliable hardware bit flip exploitation.

### 4.7.3 Breaking Weakened Cryptosystems

Fault attacks have been introduced in cryptography by Boneh et al. [16]; their attack was highly effective against implementations of RSA that use the Chinese Remainder Theorem. Since then, many variants of fault attacks against cryptographic implementations have been described as well as countermeasures against these attacks. Seifert was the first to consider attacks in which faults were introduced in the RSA modulus [113]; his goal was limited to forging signatures. Brier et al. [23] have extended his work to sophisticated methods to recover the private key; they consider a setting of uncontrollable faults and require many hundreds to even tens of thousands of faults. In our attack setting, the attacker can choose the location and observe the modulus, which reduces the overhead substantially.

In the case of Diffie-Hellman, the risk of using it with moduli that are not strong primes or hard-to-factor integers was well understood and debated extensively during the RSA versus DSA controversy in the early 1990s (e.g., in a panel at Eurocrypt'92 [37]). Van Oorschot and Wiener showed how a group order with small factors can interact badly with the use of small Diffie-Hellman exponents [126]. In 2015, the Logjam attack [6] raised new interest in the potential weaknesses of Diffie-Hellman parameters.

In this chapter, we performed a formal cryptanalysis of RSA public keys in the presence of bit flips. Our evaluation of dFFS with bit-flipped default 2048-bit RSA public keys confirmed our theoretical results. dFFS can induce bit flips in RSA public keys and factorize 99% of the resulting 2048-bit keys given enough Rowhammer-induced bit flips. We further showed that we could factor 4.2% of the two 4096 bit **Ubuntu Archive Automatic Signing Keys** with a bit flip. This allowed us to generate enough templates to successfully trick a victim VM into installing our packages. For completeness, we also included a formal cryptanalysis of Diffie-Hellman exponents in the presence of bit flips in Appendix 4.9.

## 4.8 Conclusions

Hardware bit flips are commonly perceived as a vehicle of production software failures with limited exploitation power in practice. In this chapter, we challenged common belief and demonstrated that an attacker armed with Flip Feng Shui (FFS) primitives can mount devastatingly powerful end-to-end attacks even in complete absence of software vulnerabilities. Our FFS implementation (dFFS) combines hardware bit flips with novel memory templating and massaging primitives, allowing an attacker to controllably seize control of arbitrary physical memory with very few practical constraints.

We used dFFS to mount practical attacks against widely used cryptographic systems in production clouds. Our attacks allow an attacker to completely compromise co-hosted cloud VMs with relatively little effort. Even more worryingly, we believe Flip Feng Shui can be used in several more forms and applications pervasively in the software stack, urging the systems security community to devote immediate attention to this emerging threat.

## Disclosure

We have cooperated with the National Cyber Security Centre in the Netherlands to coordinate disclosure of the vulnerabilities to the relevant parties.

## Acknowledgments

We would like to thank our anonymous reviewers for their valuable feedback. This work was supported by Netherlands Organisation for Scientific Research through the NWO 639.023.309 VICI “Dowsing” project, Research Council KU Leuven under project C16/15/058, the FWO grant G.0130.13N, and by the European Commission through projects H2020 ICT-32-2014 “SHARCS” under Grant Agreement No. 644571 and H2020 ICT-2014-645622 “PQCRYPTO”.

## 4.9 Cryptanalysis of Diffie-Hellman with Bit Flips

This section describes how one can break Diffie-Hellman by flipping a bit in the modulus. Similar to RSA, Diffie-Hellman cryptosystem performs computations modulo  $n$ . In the Diffie-Hellman key agreement scheme [41], however, the modulus  $n$  is prime or  $s = \gamma_1 = 1$ . It is very common to choose strong primes, which means that  $q = (n-1)/2$  is also prime; this is also the approach taken by OpenSSH. Subsequently a generator  $g$  is chosen of order  $q$ . In the Diffie-Hellman protocol the client chooses a random  $x \in [1, n-1]$  and computes  $g^x \bmod n$  and the server chooses a random  $y \in [1, n-1]$  and computes  $g^y \bmod n$ . After exchanging these values, both parties can compute the shared secret  $g^{xy} \bmod n$ . The best known algorithm to recover the shared secret is to solve the discrete logarithm problem to find  $x$  or  $y$  using the GNFS, which has complexity  $O(L_n[1/3, 1.92])$ . For a 512-bit modulus  $n$ , the pre-computation cost is estimated to be about 10 core-years; individual discrete logarithms  $\bmod n$  can subsequently be found in 10 minutes [6]. The current record is 596 bits [21]; again 1024 bits seems to be within reach of intelligence agencies [6].

By flipping a single bit of  $n$ , the parties compute  $g^x \bmod n'$  and  $g^y \bmod n'$ . It is likely that recovering  $x$  or  $y$  is now much easier. If we flip the LSB,  $n' = n-1 = 2q$  with  $q$  prime and  $g$  will be a generator. In the other cases  $n'$  is a  $t$ -bit or  $(t-1)$ -bit odd integer; we conjecture that its factorization has the same form as that of a random odd integer of the same size. It is not necessarily the case the  $g$  is a generator  $\bmod n'$ , but with very high probability  $g$  has large multiplicative order.

The algorithm to compute a discrete logarithm in  $Z_{n'}$  to recover  $x$  from  $y = g^x \bmod n'$  requires two steps.

1. Step 1 is to compute the factorization of  $n'$ . This is the same problem as the one considered in Section 4.3.
2. Step 2 consists in computing the discrete logarithm of  $g^x \bmod n'$ : this can be done efficiently by computing the discrete logarithms modulo  $g^x \bmod p_i'^{\tilde{\gamma}_i}$  and by combining the result using the Chinese remainder theorem. Note that except for the small primes, the  $\tilde{\gamma}_i$  are expected to be equal to 1 with high probability. Discrete logarithms  $\bmod p_i'^{\tilde{\gamma}_i}$  can in turn be computed starting from discrete logarithms  $\bmod p_i'$  ( $\tilde{\gamma}_i$  steps are required). If  $p_i' - 1$  is smooth (that is, it is of the form  $p_i' - 1 = \prod_{j=1}^r q_j^{\delta_j}$  with  $q_j$  small), the Pohlig-Hellman algorithm [102] can solve this problem in time  $O\left(\sum_{j=1}^r \delta_j \sqrt{q_j}\right)$ . If  $n'$  has prime factors  $p_i'$  for which  $p_i' - 1$  is not smooth, we have to use for those primes GNFS with complexity  $O(L_{p_i'}[1/3, 1.92])$ .

The analysis is very similar to that of Section 4.3, with as difference that for RSA we can use ECM to find all small prime factors up to the second largest one  $p_2'$ . With a simple primality test we verify that the remaining integer is prime and if so the factorization is complete. However, in the case of the discrete logarithm algorithm we have to perform in Step 2 discrete logarithm computations modulo the largest prime  $p_1'$ . This means that if  $n'$  would prime (or a small multiple of a prime), Step 1

would be easy but we have not gained anything with the bit flip operation. It is known that the expected bitlength of the largest prime factor  $p'_1$  of  $n'$  is  $0.624 \cdot t$  [71] (0.624 is known as the Golomb–Dickman constant). A second number theoretic result by Dickman shows that the probability that all the prime factors  $p'_i$  of an integer  $n'$  are smaller than  $n'^{1/u}$  has asymptotic probability  $u^{-u}$  [40].

For  $t = 1024$ , the expected size of the largest prime factor  $p'_1$  of  $n'$  is 639 bits and in turn the largest prime factor of  $p'_1 - 1$  is expected to be 399 bits ( $1024 \cdot 0.624^2$ ). Note that  $p'_1 - 1$  can be factored efficiently using ECM as in the RSA case. If  $p'_1 - 1$  has 639 bits, the probability that it is smooth (say has factors less than 80 bits) is  $8^{-8} = 2^{-24}$ , hence Pohlig–Hellman cannot be applied. We have to revert to GNFS for a 399-bit integer. However, with probability  $2^{-2} = 1/4$  all the factors of  $n'$  are smaller than 512 bits: in that case the largest prime factor of  $p'_1 - 1$  is expected to be 319 bits, but again with probability  $1/4$  all prime factors are smaller than 256 bits. Hence with probability  $1/16$  GNFS could solve the discrete logarithm in less than 1 core hour.

For  $t = 2048$ , the expected size of the largest prime factor  $p'_1$  of  $n'$  is 1278 bits and the largest prime factor of  $p'_1 - 1$  is expected to be 797 bits – this is well beyond the current GNFS record. However, with probability  $3 \cdot 10^{-3} = 0.037$  all prime factors of  $n'$  are smaller than 638 bits. Factoring  $p'_1 - 1$  is feasible using ECM, given that the its second largest prime factor is expected to be 134 bits. The largest prime factor of  $p'_1 - 1$  is expected to be 398 bits. The discrete logarithm problem modulo the largest factor can be solved using GNFS in about 1 core-month. With probability  $4 \cdot 10^{-4} = 3.9 \cdot 10^{-3}$  all prime factors of  $n'$  are smaller than 512 bits, and in that case the largest prime factor of  $p'_1 - 1$  is expected to be 319 bits, which means that GNFS would require a few core-hours.

Even if it would not be feasible to compute the complete discrete logarithm there are special cases: if  $x$  or  $y$  have substantially fewer than  $t$  bits, it is sufficient to recover only some of the discrete logarithms  $\text{mod } p'_i$  and the hardest discrete logarithm  $p_1$  can perhaps be skipped; for more details, see [6; 126].

The main conclusion is that breaking discrete logarithms with the bit flip attack is more difficult than factorizing, but for 1024 bits an inexpensive attack is feasible, while for 2048 bits the attack would require a moderate computational effort, the results of which are widely applicable. It is worth noting that this analysis is applicable to the DH key agreement algorithm in use by OpenSSH, defaulting to 1536-bit DH group moduli in the current OpenSSH (7.2), bitflipped variants of which can be pre-computed by a moderately equipped attacker, and applied to all OpenSSH server installations. The consequences of such an attack are decryption of a session, including the password if used, adding another attractive facet to attacks already demonstrated in this chapter.

## Conclusion

In this thesis, we have introduced and explored novel methods of exploitation that make use of interactions between abstraction layers. We have presented three ways in which unforeseen interactions between abstraction layers can lead to security vulnerabilities, or make them worse.

If we want to improve security, it is insufficient to make individual components secure. We also need to consider how the different components of a system interact unexpectedly, and how these interactions may produce new avenues for exploitation.

As the next step, we must ask whose task it is to fix cross layer vulnerabilities like those we have presented in this thesis. Dumping the responsibility of addressing undetectable random hardware errors onto an application developer is just not going to work. This remains an open question.

### 1. *SROP*.

In Chapter 2, we have discussed sigreturn oriented programming, a novel, Turing complete technique for programming a novel type of weird machine. We show it is possible to write code-reuse exploits for UNIX/Linux programs which only use code that the operating system has made available in every process. With sigreturn oriented programming, we abuse the stub that allows programs to resume normal execution after a signal handler has completed. Sigreturn oriented programming is a generic technique, as we demonstrated by using it for an exploit, a backdoor, and a code-signing bypass. Moreover, it works on a wide variety of operating systems and different architectures. For several of these systems, sigreturn oriented programming permits exploitation without any precise knowledge about the executable. Moreover, the exploit is reusable, as it does not depend much on the victim process at all.

Sigreturn oriented programming represents a convenient, portable technique to program arbitrary code even in strongly protected machines. The number of gadgets needed is minimal and in many systems those gadgets are in a fixed



location. In some cases this technique even allows for exploits that work regardless of what specific version of a vulnerable program is being exploited. As such, it ranks among the lowest hanging fruit currently available to attackers on UNIX systems. It is important to emphasize that even if kernels are patched to eliminate these fixed-location gadgets, the usefulness for backdoors is undiminished. In summary, we believe that sigreturn oriented programming is a powerful addition to the attackers' arsenal.

Our exploration of sigreturn oriented programming shows how the choice of implementation of an obscure part of the operating system can influence the ease with which a vulnerable application program can be exploited. This serves as an example of how the unforeseen interaction between two independently developed pieces in a system can have an effect on its security.

## 2. *Dedup Est Machina.*

In Chapter 3, we have examined the security implications of memory deduplication and its interaction with Rowhammer [69], a hardware flaw, to create exploits for programs that themselves need not have exploitable vulnerabilities. We have shown how memory deduplication can be viewed as a programmable weird machine capable of leaking secrets.

Adding more and more functionality to operating systems leads to an ever-expanding attack surface. Even ostensibly harmless features like memory deduplication may prove to be extremely dangerous in the hands of an advanced attacker. Deduplication-based primitives can do much more harm than merely providing a slow side channel. An attacker can use our primitives to leak password hashes, randomized code and heap pointers, and start off reliable Rowhammer attacks. We find it extremely worrying that an attacker who simply times write operations and then reads from an unrelated addresses can reliably gain control over a system with all defenses up, even if the software is entirely free of bugs. Our conclusion is that we should introduce complex features in an operating system only with the greatest care (and after a thorough examination for side channels), and that full memory deduplication inside the operating system is a dangerous feature that is best turned off. In addition, we have shown that full deduplication is an overly conservative choice in the practical cases of interest and that deduplicating only zero pages can retain most of the memory-saving benefits of full deduplication while addressing its alarming security problems.

One contribution of our work is to show how deduplication can be abused in a more powerful way than via a simple 1 bit oracle. By interacting with an application, an attacker can exert some control over the very pages it is trying to leak.

Setting up guess pages and manipulating a victim program into creating pages that are suitable to leak secrets can be viewed as programming a weird machine. Namely one that compares any two pages and leaves a measurable

trace. It is worth noting that it is a well-behaved part of the system that performs the computation that is useful to us.

The final contribution is that we show how to exploit a browser by serving a web page without making use of any flaws in said browser. Instead, we use memory deduplication as an OS side channel and combine it with a hardware flaw (Rowhammer) to create an arbitrary read/write exploit primitive. The browser serves as a playground to manipulate state, in order to exploit flaws in different layers.

### 3. *Flip Feng-shui*.

Hardware bit flips are commonly perceived as a vehicle of production software failures with limited exploitation power in practice. In Chapter 4, we challenge this common belief and demonstrate that an attacker armed with Flip Feng Shui (FFS) primitives can mount devastatingly powerful end-to-end attacks even in complete absence of software vulnerabilities. Our FFS implementation (dFFS) combines hardware bit flips with novel memory templating and massaging primitives, allowing an attacker to controllably seize control of arbitrary physical memory with very few practical constraints.

We use dFFS to mount practical attacks against widely used cryptographic systems in production clouds. Our attacks allow an attacker to completely compromise co-hosted cloud VMs with relatively little effort. Even more worryingly, we believe Flip Feng Shui can be used in several more forms and applications pervasively in the software stack, urging the systems security community to devote immediate attention to this emerging threat.

Compared to the previous chapter, we have crafted a different exploit primitive from the same (well-behaved) weird machine. Instead of creating a memory leak primitive, we use deduplication to perform an action, similar to Heap Feng-shui, to get the right data into the right place. Combining this with Rowhammer gives us a write primitive.

## Aftermath of vulnerability disclosure

Since the publication of the papers which are part of this thesis, there have been a few developments relevant to mitigating the problems we identified.

### 1. *SROP*.

We have submitted a patchset [18] to the Linux kernel mailing list (LKML) to implement stack canaries (as described in section 2.10). This patchset was rejected in fear of breaking userspace programs. The effort was then picked up by others, leading to a second patchset, which tried to address the possible ABI breakage [12], sadly this was also never merged. However, this patchset led OpenBSD to adopt a similar approach [35], making the use of Sigreturn

Oriented Programming on OpenBSD much less attractive, hopefully to the point that it does not give an attacker any additional advantages.

## 2. *Dedup Est Machina.*

As stated in the paper, we have worked with our contacts at Microsoft to help solve security problems caused by memory deduplication.

As a result, Microsoft temporarily disabled memory deduplication with security update MS16-092 on July 16 2016. The vulnerability, as it applies to Windows operating systems was given the Common Vulnerabilities and Exposures identifier *CVE-2016-3272*.

Microsoft has since partially re-enabled memory deduplication [31], allowing only pages from within the same security domain to be merged. This re-enables intra-process attacks like the ones we have shown in Microsoft Edge. It is possible that mitigation against intra-process leakage is seen as futile, especially with the emergence of micro-architectural side-channels like the various Spectre variants [72], that allow for similar information leaks, and which cannot be completely mitigated in the foreseeable future.

However, restricting deduplication only to pages within security domains does not completely eliminate the risk of leaking information between domains, as we have explored in [31].

Rowhammer remains a problem, and the subject of a lot of ongoing research [27; 36; 39; 49; 48; 55; 77; 124; 125].

## 3. *Flip Feng-shui.*

As a result of our cooperation with the National Cyber Security Centre in the Netherlands, they have published a fact-sheet detailing the risks of Flip Feng Shui to organizations which make use of shared ICT services [93]. They have coordinated the disclosure of the issue with many parties including OpenSSH, GnuPG, VM monitor vendors (Oracle, Redhat, Xen, VMware), and Debian and Ubuntu, all before the paper was public. All these parties have responded. In particular, GnuPG changed the way signatures are verified to guard against single bitflips [59].

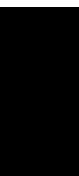
# Future Directions

1. *Side-channel resistant systems.* Side channels have long been treated as either too impractical to exploit in real-world scenarios, or too difficult to solve. But changes in our computing environments have made them more dangerous in practice. Current systems, both hardware and software, have been largely designed without taking side channels into account. One possible research direction could be to develop mechanisms (in hardware or software) allowing us to more easily reason about the existence and impact of side channels.

2. ***Rowhammer & other hardware flaws.*** Rowhammer continues to be an unsolved problem and the field of finding flaws baked into silicon is ever expanding. Hardware flaws can turn even a formally verified program into a system of which correctness cannot be guaranteed. Attempts to fix Rowhammer thus far have been done behind closed doors by hardware vendors. Only after the hardware was already in stores could researchers reverse engineer its implementation and subsequently show it could still be broken.

Further work could be done to standardize mitigations against common hardware flaws, like Rowhammer, so that a thorough security analysis can be undertaken in the design phase already, giving system integrators more confidence in the correctness of their products.

3. ***Continued research into cross-layers problems.*** Newly discovered vulnerability classes by definition defy prior categorization and require novel methods to be found. When I started my PhD studies, I did not initially plan for my research to extensively focus on attacks, but following up on interesting leads meant that I stumbled upon the subjects that defined my thesis. When you stumble upon something complex you do not understand, chances are other people also have not explored it. Researching cross-layer problems requires considering how the different components of a system interact in unexpected manners.



## Contributions to Papers

### **Framing signals - Return to portable exploits.**

**Bosman, E.:** Conceptualization, Software, Writing - Original Draft, Writing - Review & Editing, Methodology, (initial) Investigation, Validation. **Bos, H.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology.

### **Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector.**

**Bosman, E.:** Conceptualization, Software, Writing - Original Draft, Writing - Review & Editing, Methodology, (initial) Investigation, Validation. **Razavi, K.:** Supervision, Writing - Original Draft, Writing - Review & Editing, Investigation, Validation. **Bos, H.:** Supervision, Writing- Original Draft, Writing - Review & Editing, Methodology. **Giuffrida, C.:** Supervision, Writing - Original Draft, Writing - Review & Editing, Investigation, Methodology.

### **Flip Feng Shui: Hammering a Needle in the Software Stack.**

**Bosman, E.:** Conceptualization, Writing - Review & Editing, Methodology, (initial) Investigation. **Gras, B.:** Software, Writing - Original Draft, Methodology, Software, Investigation, Validation. **Razavi, K.:** Supervision, Writing - Original Draft, Writing - Review & Editing, Investigation, Validation. **Preneel, B.:** Formal analysis, Writing - Original Draft **Giuffrida, C.:** Supervision, Writing - Original Draft, Writing - Review & Editing, Methodology. **Bos, H.:** Supervision, Writing - Original Draft, Writing - Review & Editing, Methodology.



## References

- [1] DDR4 Rowhammer mitigation. <http://www.passmark.com/forum/showthread.php?5301-Rowhammer-mitigation&p=19553>. Accessed on 17.2.2016.
- [2] ptmalloc. <http://www.malloc.de/en>.
- [3] Troubleshooting Memory Errors – MemTest86. <http://www.memtest86.com/troubleshooting.htm>. Accessed on 17.2.2016.
- [4] Administrators implementing SIS with sensitive data should not share folders. <https://support.microsoft.com/en-us/kb/969199>.
- [5] nginx test suite. <https://github.com/catap/nginx-tests>.
- [6] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella Béguelin, and Paul Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. CCS'15, 2015.
- [7] Barbara P. Aichinger. DDR Compliance Testing - Its time has come! In *JEDEC's Server Memory Forum*, 2014.
- [8] Andrea Arcangeli. Transparent hugepage support. *KVM Forum*, 2010.
- [9] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *OLS, OLS'09*, 2009.
- [10] JEDEC Solid State Technology Association. Low Power Double Data 4 (LPDDR4). JESD209-4A, Nov 2015.
- [11] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN:



- Silently Breaking ASLR in the Cloud. In *WOOT*, WOOT'15, 2015.
- [12] Scotty Bauer. [patchv2 0/2] srop mitigation: Signal cookies. <https://lwn.net/Articles/674861/>.
- [13] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA*, 2002.
- [14] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th conference on USENIX Security Symposium*, SSYM'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association. URL <http://seclab.cs.sunysb.edu/sandeep/research.html>.
- [15] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0564-8. doi: 10.1145/1966913.1966919. URL <http://doi.acm.org/10.1145/1966913.1966919>.
- [16] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14(2), 2001.
- [17] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.
- [18] Erik Bosman. [patch 1/4] srop mitigation: Architecture independent code for signal canaries. <https://lkm1.org/lkm1/2014/5/15/660>.
- [19] Erik Bosman and Herbert Bos. Framing signals—a return to portable shell-code. SP'14.
- [20] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. SP'16, 2016.
- [21] Cyril Bouvier, Pierrick Gaudry, Laurent Imbert, Hamza Jeljeli, and Emmanuel Thomé. Discrete logarithms in  $GF(p)$  – 180 digits. <https://listserv.nodak.edu/cgi-bin/wa.exe?A2=ind1406&L=NMBRTHRY&F=&S=&P=3161>. June 2014.
- [22] Sergey Bratus. What are weird machines? <https://www.cs.dartmouth.edu/~sergey/wm/>.
- [23] Eric Brier, Benoît Chevallier-Mames, Mathieu Ciet, and Christophe Clavier. Why one should also secure RSA public key elements. CHES'06, 2006.
- [24] Bulba and Kil3r. Bypassing StackGuard and StackShield. <http://www.phrack.org/issues.html?issue=56&id=5article>.

- [25] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *USENIX Security*, SEC'15, 2015.
- [26] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Drop: Detecting return-oriented programming malicious code. In *ICISS*, ICISS '09, pages 163–177, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10771-9. doi: 10.1007/978-3-642-10772-6\_13. URL [http://dx.doi.org/10.1007/978-3-642-10772-6\\_13](http://dx.doi.org/10.1007/978-3-642-10772-6_13).
- [27] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P*, May 2019. URL [https://download.vusec.net/papers/eccploit\\_sp19.pdf](https://download.vusec.net/papers/eccploit_sp19.pdf). Best Practical Paper Award, Pwnie Award Nomination for Most Innovative Research, DCSR Paper Award Runner-up.
- [28] Cristian Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4), 2003.
- [29] 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 *CCS*, 2015.
- [30] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Technical report, May 2008. URL <http://tools.ietf.org/html/rfc5280>.
- [31] Andreas Costi, Brian Johannesmeyer, Erik Bosman, Cristiano Giuffrida, and Herbert Bos. On the Effectiveness of Same-Domain Memory Deduplication. In *EuroSec*, April 2022. URL [https://download.vusec.net/papers/dedupestoreturns\\_eurosec22.pdf](https://download.vusec.net/papers/dedupestoreturns_eurosec22.pdf).
- [32] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *NDSS*, NDSS'15, 2015.
- [33] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Ropdefender: a detection tool to defend against return-oriented programming attacks. In *ASIACCS*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0564-8. doi: 10.1145/1966913.1966920. URL <http://doi.acm.org/10.1145/1966913.1966920>.
- [34] 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 *NDSS*, 2015.
- [35] Theo de Raadt. Srop mitigation. <https://marc.info/?l=openbsd-tech&m=146281531025185>.
- [36] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In *USENIX Security*, August 2021. URL [https://download.vusec.net/papers/smash\\_sec21.pdf](https://download.vusec.net/papers/smash_sec21.pdf). Pwnie Award Nomination for Most Under-Hyped Research, Best Faculty of Science Master Thesis Award.
- [37] Yvo Desmedt, Peter Landrock, Arjen K. Lenstra, Kevin S. McCurley, Andrew M. Odlyzko, Rainer A. Rueppel, and Miles E. Smid. The Eurocrypt '92 Controversial Issue: Trapdoor Primes and Moduli (Panel). Eurocrypt'92, 1992.
- [38] The Sage Developers. Sage Mathematics Software (Version). <http://www.sagemath.org>. Accessed on 17.2.2016.
- [39] Andrea Di Dio, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Copy-on-Flip: Hardening ECC Memory Against Rowhammer Attacks. In *NDSS*, February 2023. URL [https://download.vusec.net/papers/cof\\_ndss23.pdf](https://download.vusec.net/papers/cof_ndss23.pdf).
- [40] Karl Dickman. On the frequency of numbers containing prime factors of a certain relative magnitude. *Arkiv forr Matematik, Astronomi och Fysik*, 1930.
- [41] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6), 1976.
- [42] Thomas Dullien. Exploitation and state machines: Programming the 'weird machine', revisited. In *Infiltrate Conference*, Miami, FLA, April 2011.
- [43] Thomas Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 8(2):391–403, 2020. doi: 10.1109/TETC.2017.2785299.
- [44] Brandon Edwards. Dos? then who was phone? (asterisk exploit related to cve-2012-5976). <http://blog.exodusintel.com/tag/asterisk-exploit/>.
- [45] Paul Erdős and Mark Kac. The Gaussian Law of Errors in the Theory of Additive Number Theoretic Functions. *American Journal of Mathematics*, 62 (1), 1940.
- [46] Chris Evans. The poisoned NUL byte, 2014 edition). <http://googleprojectzero.blogspot.nl/2014/08/the-poisoned-nul-byte-2014-edition.html>. Accessed on 17.2.2016.

- [47] Justin N. Ferguson. Understanding the heap by breaking it. In *Black Hat USA*, 2007.
- [48] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *S&P*, May 2018. URL [https://download.vusec.net/papers/glitch\\_sp18.pdf](https://download.vusec.net/papers/glitch_sp18.pdf). Pwnie Award Nomination for Most Innovative Research, DCSR Paper Award Runner-up.
- [49] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *S&P*, May 2020. URL [https://download.vusec.net/papers/trrespass\\_sp20.pdf](https://download.vusec.net/papers/trrespass_sp20.pdf). Best Paper Award, Pwnie Award for Most Innovative Research, IEEE Micro Top Picks Honorable Mention, DCSR Paper Award.
- [50] Francesco Gadaleta, Yves Younan, and Wouter Joosen. ESSoS'10, 2010.
- [51] Daniel Kahn Gillmor. pem2openpgp - translate PEM-encoded RSA keys to OpenPGP certificates. Accessed on 17.2.2016.
- [52] githubssh. GitHub Developer – Public Keys. <https://developer.github.com/v3/users/keys/>. Accessed on 17.2.2016.
- [53] Cristiano Giuffrida, Calin Iorgulescu, and Andrew S. Tanenbaum. Mutable Checkpoint-Restart: Automating Live Update for Generic Server Programs. In *Middleware*, 2014.
- [54] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. *SP '03*, 2003.
- [55] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261, Los Alamitos, CA, USA, may 2018. IEEE Computer Society. doi: 10.1109/SP.2018.00031. URL <https://doi.ieeecomputersociety.org/10.1109/SP.2018.00031>.
- [56] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed Javascript. In *ESORICS*, ESORICS'15. 2015.
- [57] Daniel Gruss, Clementine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in JavaScript. <http://arxiv.org/abs/1507.06955>.
- [58] Daniel Gruss, Clementine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. *DIMVA'16*, 2016.
- [59] The GNU Privacy Guard. gpgv: Tweak default options for extra secu-

- rity. <https://git.gnupg.org/cgi-bin/gitweb.cgi?p=gnupg.git;a=commit;h=e32c575e0f3704e7563048eea6d26844bdfc494b>.
- [60] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security and Privacy Magazine, special issue of Cloud Security*, 8, 2010.
  - [61] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel-space ASLR. In *IEEE S&P*, 2013.
  - [62] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 Strikes Back. *ASIA CCS'15*, 2015.
  - [63] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Know Thy Neighbor: Crypto Library Detection in Cloud. In *PETS, PETS'15*, 2015.
  - [64] Sergey Bratus Julian Bangert. Page fault liberation army or gained in translation. <http://events.ccc.de/congress/2012/Fahrplan/events/5265.en.html>.
  - [65] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. Ret2Dir: Rethinking Kernel Isolation. *SEC'14*, 2014.
  - [66] Linux Kernel. How to use the Kernel Samepage Merging feature. <https://www.kernel.org/doc/Documentation/vm/ksm.txt>. Accessed on 17.2.2016.
  - [67] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC, ACSAC '06*, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2716-7. doi: 10.1109/ACSAC.2006.9. URL <http://dx.doi.org/10.1109/ACSAC.2006.9>.
  - [68] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud. In *USENIX Security, SEC'12*, 2012.
  - [69] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA, ISCA'14*, 2014.
  - [70] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman J. J. te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. *CRYPTO'10*,

- 2010.
- [71] Donald E. Knuth and Luis Trabb-Pardo. Analysis of a Simple Factorization Algorithm. *Theoretical Computer Science*, 3(3), 1976.
  - [72] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.
  - [73] Dmitrii Kuvaiskii, Rasha Fagheh, Pramod Bhatotia, Pascal Felber, and Christof Fetzter. HAFT: Hardware-assisted Fault Tolerance. EuroSys'16, 2016.
  - [74] Gabriel Lawrence and Chris Frohoff. Marshalling pickles: how deserializing objects will ruin your day. AppSec California 2015, talk <https://frohoff.github.io/appseccali-marshalling-pickles/>.
  - [75] Hendrik W. Lenstra. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126, 1987.
  - [76] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *EuroSys*, EuroSys '10, pages 195–208, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi: 10.1145/1755913.1755934. URL <http://doi.acm.org/10.1145/1755913.1755934>.
  - [77] Moritz Lipp, Michael Schwarz, Lukas Raab, Lukas Lamster, Misiker Tadesse Aga, Clémentine Maurice, and Daniel Gruss. Nethammer: Inducing rowhammer faults through network requests. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 710–719. IEEE, 2020.
  - [78] David Litchfield. Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server. 2003.
  - [79] Dwayne Litzenberger. PyCrypto - The Python Cryptography Toolkit). <https://www.dlitz.net/software/pycrypto/>. Accessed on 17.2.2016.
  - [80] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE S&P*, SP'15, 2015.
  - [81] Vincenzo Lozzo, Tim Kornau, and Ralf-Philipp Weinmann. Everybody be cool, this is a roppery! In *Black Hat USA*, Las Vegas, USA, 2010.
  - [82] 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 *CCS*, 2015.
  - [83] Tarjei Mandt. Kernel Pool Exploitation on Windows 7. In *Black Hat Europe*, 2011.

- [84] Clementine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurelien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *RAID*, RAID'15, 2015.
- [85] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. 1996.
- [86] Microsoft. Reducing runtime memory in Windows 8. <http://goo.gl/IP5610>. Accessed on 17.2.2016.
- [87] Microsoft. Cache and Memory Manager Improvements in Windows Server 2012. <http://goo.gl/SbhoFC>. Accessed on 17.2.2016.
- [88] Microsoft. Data execution prevention.
- [89] Matt Miller. Preventing the exploitation of seh overwrites. *Uninformed Journal*, 5, 2006.
- [90] Matt Miller and Ken Johnson. Exploit mitigation improvements in Windows 8. In *Black Hat USA*, 2012.
- [91] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, 2011.
- [92] Urban Müller. Brainfuck—an eight-instruction turing-complete programming language. Available at the Internet address <http://www.muppetlabs.com/breadbox/bff/>, 1993.
- [93] Nationaal Cyber Security Centrum (NCSC). Use virtualisation wisely. [https://english.ncsc.nl/binaries/ncsc-en/documenten/factsheets/2019/juni/01/factsheet-use-virtualisation-wisely/Factsheet\\_Use-virtualisation-wisely-v1-0.pdf](https://english.ncsc.nl/binaries/ncsc-en/documenten/factsheets/2019/juni/01/factsheet-use-virtualisation-wisely/Factsheet_Use-virtualisation-wisely-v1-0.pdf).
- [94] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, ACSAC '10, pages 49–58. ACM, December 2010. ISBN 978-1-4503-0133-6.
- [95] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, CCS'15, 2015.
- [96] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, CT-RSA'06, 2006.
- [97] R. Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *IPCCC*, IPCCC'11, 2011.

- [98] Vasilis Pappas. kbouncer: Efficient and transparent rop mitigation. In *USENIX Security*, 2013.
- [99] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE S&P, SP '12*, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.41. URL <http://dx.doi.org/10.1109/SP.2012.41>.
- [100] PaX Project. Address Space Layout Randomization. <http://pax.grsecurity.net/docs/aslr.txt>.
- [101] Peter Pessl, Daniel Gruss, Clementine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. SEC'16, 2016.
- [102] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over  $GF(p)$  and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1), 1978.
- [103] Shashank Rachamalla, Dabadatta Mishra, and Purushottam Kulkarni. Share-o-meter: An empirical analysis of KSM based memory sharing in virtualized systems. In *HiPC, HiPC'13*, 2013.
- [104] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of the 18th conference on USENIX security symposium, SEC'09*, 2009. URL <http://dl.acm.org/citation.cfm?id=1855768.1855779>.
- [105] Kaveh Razavi, Gerrit van der Kolk, and Thilo Kielmann. Prebaked uVMs: Scalable, Instant VM Startup for IaaS Clouds. ICDCS '15, 2015.
- [106] Dennis Ritchie. Unix version 6 – signal.s. <http://minnie.tuhs.org/cgi-bin/utree.pl?file=V6/usr/source/s5/signal.s>.
- [107] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2), 1978.
- [108] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *ACSAC, ACSAC '09*, pages 60–69, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3919-5. doi: 10.1109/ACSAC.2009.16. URL <http://dx.doi.org/10.1109/ACSAC.2009.16>.
- [109] Jurgen Schmidt. JIT Spraying: Exploits to beat DEP and ASLR. In *Black Hat Europe*, 2010.
- [110] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit



- hardening made easy. In *USENIX Security*, 2011.
- [111] Mark Seaborn. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. In *Black Hat USA*, 2015.
  - [112] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. <http://goo.gl/0MJcj3>.
  - [113] Jean-Pierre Seifert. On authenticated computing and RSA-based authentication. CCS'05, 2005.
  - [114] 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'07, 2007. doi: <http://doi.acm.org/10.1145/1315245.1315313>. URL <http://doi.acm.org/10.1145/1315245.1315313>.
  - [115] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS'04)*, 2004. doi: <http://doi.acm.org/10.1145/1030083.1030124>. URL <http://portal.acm.org/citation.cfm?doid=1030124#>.
  - [116] Noam Shalev, Eran Harpaz, Hagar Porat, Idit Keidar, and Yaron Weinsberg. CSR: Core Surprise Removal in Commodity Operating Systems. ASP-LOS'16, 2016.
  - [117] Rebecca Shapiro. The care and feeding of weird machines found in executable metadata. <http://events.ccc.de/congress/2012/Fahrplan/events/5195.en.html>.
  - [118] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide Page Deduplication in Virtual Environments. In *HPDC*, HPDC'12, 2012.
  - [119] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE S&P*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.45. URL <http://dx.doi.org/10.1109/SP.2013.45>.
  - [120] Solar Designer. Getting around non-executable stack (and fix). <http://seclists.org/bugtraq/1997/Aug/63>.
  - [121] Alexander Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe*, 2007.
  - [122] Joel Spolsky. The law of leaky abstractions. <https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky->

abstractions/.

- [123] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication As a Threat to the Guest OS. In *EuroSec*, EUROSEC'11, 2011.
- [124] Andrei Tatar, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer. In *RAID*, September 2018. URL <https://download.vusec.net/papers/hammertime RAID18.pdf>. Best Paper Award.
- [125] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, October 2016. URL <https://vvdveen.com/publications/drammer.pdf>. Pwnie Award for Best Privilege Escalation Bug, Android Security Reward, CSAW Best Paper Award, DCSR Paper Award.
- [126] Paul C. van Oorschot and Michael J. Wiener. On Diffie-Hellman key agreement with short exponents. *Eurocrypt'96*, 1996.
- [127] Julien Vanegue. The Automated Exploitation Grand Challenge, Tales of Weird Machines. In *H2C2 conference*, Sao Paulo, Brazil, October 2013.
- [128] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on ios: when benign apps become evil. In *USENIX Security*, SEC'13, pages 559–572, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL <http://dl.acm.org/citation.cfm?id=2534766.2534814>.
- [129] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. A Covert Channel Construction in a Virtualized Environment. In *CCS*, CCS'12, 2012.
- [130] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side-Channels for Untrusted Operating Systems. In *IEEE S&P*, SP'15, 2015.
- [131] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security*, SEC'14, 2014.
- [132] Mark Yason. MemGC: Use-after-free exploit mitigation in Edge and IE on Windows 10. <http://goo.gl/yUw0vY>.
- [133] Zhang Yunhai. Bypass Control Flow Guard Comprehensively. In *Black Hat USA*, 2015.
- [134] Michal Zalewski. Delivering Signals for Fun and Profit. <http://lcamtuf.coredump.cx/signals.txt>.
- [135] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-

Tenant Side-Channel Attacks in PaaS Clouds. In *CCS, CCS'14*, 2014.

## Summary

Abstractions are fundamental to software engineering. Modern systems are simply too complex for any single person to completely keep in their head, and abstractions are a great way to divide up required functionality into separate components. This allows for division of labor and simplifies reasoning about the correctness of parts of the system.

However, hiding implementation details behind abstractions can conceal problems that implementations might have. On top of that, confusion about what exact functionality an abstraction provides can also lead to flaws that can be exploited by attackers.

Exploit development is a software engineering discipline in itself, and as such, also often makes use of abstractions. While an attacker has to work with the implementations of their target system, they are not restricted to viewing this system through the same abstractions. Sometimes un(der)specified behaviors in the implementation of one abstraction, which in isolation seem innocuous, can interact with other parts of a system to create a new flaw, or exacerbate an existing one.

This dissertation explores three ways in which unforeseen interactions between abstraction layers can lead to security vulnerabilities, or make them worse.

First, we explore how the choice of implementation of an obscure part of UNIX/Linux operating system enables an exploitation technique called sigreturn oriented programming. It turns out that this gives attackers a lot of control over a program's execution state, simplifying writing exploits greatly. In some cases, it even allows for exploits that work regardless of what specific version of a vulnerable program is being exploited.

Next, we examine the security implications of Memory Deduplication, a performance optimization commonly found in both virtualization environments and operating systems. Our work shows how Deduplication can be abused as a side-channel

in a more powerful way than was previously known. Furthermore, by combining this side-channel to leak data, with a hardware flaw called Rowhammer to cause a useful corruption informed by this data, we show that it is possible to exploit a browser without making use of any flaws in that browser.

Finally we combine Rowhammer and Deduplication in a different way to create a precise memory corruption primitive in a technique we call Flip Feng-shui. Here, Deduplication is used as a means to get more control over an otherwise random memory corruption. This enables us to corrupt cryptographic key material, and configuration files, and allows us to take over a virtual machine from another virtual machine that runs on the same hardware.