The Jury Is In: Monolithic OS Design Is Flawed

Microkernel-based Designs Improve Security

Simon Biggs, Damon Lee, Gernot Heiser
Data61, CSIRO
UNSW Sydney
gernot.heiser@data61.csiro.au

ABSTRACT

The security benefits of keeping a system's trusted computing base (TCB) small has long been accepted as a truism, as has the use of internal protection boundaries for limiting the damage caused by exploits. Applied to the operating system, this argues for a small microkernel as the core of the TCB, with OS services separated into mutually-protected components (servers) – in contrast to "monolithic" designs such as Linux, Windows or MacOS. While intuitive, the benefits of the small TCB have not been quantified to date. We address this by a study of critical Linux CVEs, where we examine whether they would be prevented or mitigated by a microkernel-based design. We find that almost all exploits are at least mitigated to less than critical severity, and 40% completely eliminated by an OS design based on a verified microkernel, such as seL4.

ACM Reference Format:
Simon Biggs, Damon Lee, Gernot Heiser. 2018. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security. In *9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3265723.3265733

1 INTRODUCTION

Modern mainstream operating systems (OSes) are designed for functionality, speed and ease of development. With the increase of functionality and support for more diverse hardware, their size and complexity keeps growing. These OSes have a monolithic structure, with most services contained in the kernel, i.e. the part of the system that executes in the privileged mode of the hardware. The result is an explosive growth in the amount of privileged code, as shown in Figure 1 for Linux. The Windows kernel, while not growing as quickly, is even bigger, with a recent version said to be 60–65 MSLOC (millions of source lines of code) [Ahmed 2016] compared to 26 MSLOC for Linux.

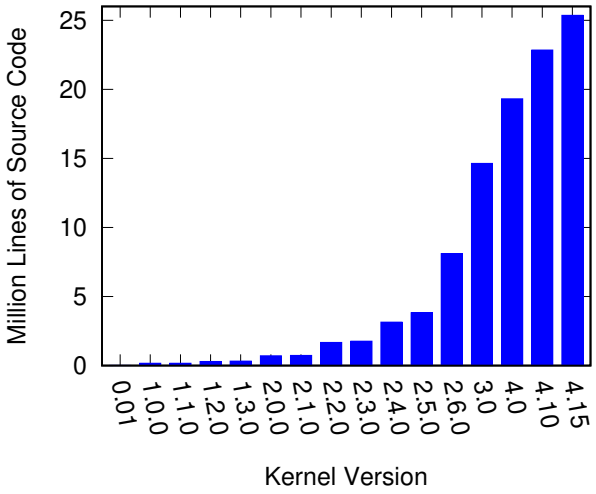

Figure 1: Linux kernel code size growth [Wikipedia].

Any code executing in privileged mode can bypass security, and is therefore inherently part of a system's *trusted computing base* (TCB) [Rushby 1981]. Hence we are experiencing a rapid growth of the TCB of mainstream OSes. As almost all code is buggy, and the number of bugs grows with the size of the code base, this TCB growth is bound to lead to a growth of the number of vulnerabilities. This is a serious problem, as OS vulnerabilities are a key enabler of cyber crime, the cost of which is estimated to reach $6 trillion ($\$6 \times 10^{12}$) by 2021 [Mason 2018].

It has long been argued that the microkernel design, with its ability to reduce TCB size, contain faults and encapsulate untrusted components, is, in terms of security, superior to monolithic systems [Hohmuth et al. 2004]. However, to the best of our knowledge, there has been no quantitative study to date that can back this claim.

In this paper we present such a study. Specifically we analyse all *critical* security bugs in the monolithic Linux kernel from the *Common Vulnerabilities and Exposures* (CVE)

APSys '18, August 27–28, 2018, Jeju Island, Republic of Korea

repository [Mitre 2018]. For each of these we analyse how a design of the same OS functionality on top of a microkernel, especially a formally verified microkernel such as seL4 [Klein et al. 2009], would retain, mitigate or eliminate the vulnerability.

The results are a stark confirmation of the arguments in favor of a small TCB. We find that 96% of critical Linux compromises would no longer be critical with a microkernel-based design, 40% would be completely eliminated by an OS based on a verified microkernel, and 29% even with an unverified microkernel.

2 BACKGROUND

2.1 Bug density

A large code base has many bugs. How many depends on factors such as the inherent (conceptual) complexity of the code, the development processes, programming languages and others. Estimates vary, Hatton [1997] estimates an average of 6 faults per kSLOC and a best-case fault density of 0.5–1/kSLOC, while Mohagheghi et al. [2004] estimate a density of 1–3/kSLOC. Kernel code is highly complex, so its bug density is unlikely to be near the lower end of estimates. But even if we take the most optimistic estimate of 0.5/kSLOC, we can expect the Linux kernel to have about 13,000 bugs, and the Windows kernel about 30,000.

2.2 Operating-system structure

These bug-density figures provide a good explanation why computer systems regularly fail to provide security. But this is not the best we can do, as had been realised decades ago. Rushby [1981] argued for a security-oriented design that uses a minimal *separation kernel* for isolating subsystems, and the Orange Book [Department of Defence 1986] requires that the TCB be minimised.

OS design for minimal TCB size implies an architecture based on a *microkernel*. Contrary to a monolithic system, such as Linux, Windows or MacOS, a microkernel keeps the code that operates in privileged mode to a bare minimum, focusing on fundamental mechanisms for managing the hardware, and implementing all actual system services and policies in user-level servers [Liedtke 1995].

Figure 2 shows the difference in design. In a monolithic system, an application obtains a service by executing a syscall instruction, which invokes the low-level system-call handler, which in turn redirects the request to the respective in-kernel service. A monolithic kernel tends to have a layered (vertical) structure [Dijkstra 1968], with more complex services building on more primitive ones. As the system evolves, layers are added and enhanced, leading to a growth in the overall kernel size and complexity.

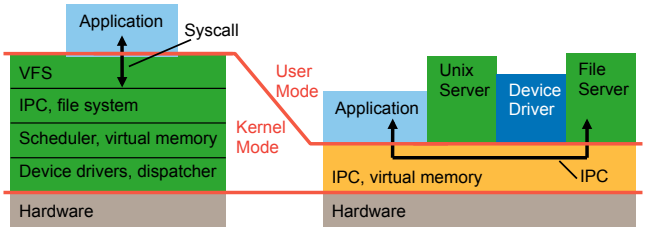

Figure 2: Monolithic vs microkernel-based OS architecture.

To invoke a service in a microkernel-based system, an application sends an inter-process communication (IPC) message to a server, which replies back by IPC. Device drivers are also user-level servers, one for each device. The microkernel provides isolation between processes and otherwise mostly acts as a fast context-switching engine. The resulting system has a horizontal structure, with applications and services running side-by-side. Functionality is added by adding servers or extending existing ones. Functionality added to one service does not affect an unrelated one.

A key property of a microkernel is that it enables a design based on *least privilege* [Saltzer and Schroeder 1975]: Services not only run in user mode, subject to kernel-imposed isolation. They also need to be furnished with only the bare minimum of privileges to do their job. For example, a device driver needs access to device registers and little more, while no other service needs access to those device registers. If a server runs with minimal privileges, then compromising it cannot escalate privileges beyond what the server holds already.

A different aspect of the same underlying principle is *fault containment*: If a server, say the IP stack, is compromised, this will affect the service it provides, but leave components that do not depend on this service completely unaffected.

There is an inherent cost to microkernels: Instead of two mode switches as in a monolithic system, the hardware cost for invoking a system service is four mode switches and two context switches: The client invokes the kernel's IPC operation (two mode switches), and so does the server (another two mode switches). To deliver the message, the kernel switches from the client's to the server's context and back (two context switches).

2.3 Microkernel pros and cons

The idea of a microkernel goes back to Brinch Hansen's Nucleus [1970], which aimed for a policy-free general-purpose substrate on which to implement the actual system. Microkernels were popular in the 1980s but were plagued by performance issues, leading to failures such as the $2bn debacle of the IBM Workplace OS [Fleisch et al. 1998], and claims

that poor performance was inherent in the approach [Chen and Bershad 1993].

This claim was put to rest by the L4 microkernel [Liedtke 1993], which showed that microkernel operations could be so fast as to add very little overhead on the services provided at user level. This is not surprising if you do the numbers: mode and context switches are in the order of dozens of cycles, and a round-trip L4 IPC typically costs of the order of 500–800 cycles, a fraction of a microsecond on modern hardware, and a tiny fraction of the cost of a typical system service.

The small performance cost is offset by a dramatically reduced TCB: A well-designed microkernel is several orders of magnitude smaller than a monolithic kernel. This is demonstrated by L4 kernels: In striking contrast to Figure 1, L4 kernels stayed close to constant in size, growing in 20 years from 6 kSLOC of the original L4 to 9 kSLOC of seL4 [Heiser and Elphinstone 2016].

Yet one might question whether it is possible to provide all functionality of a full-blown OS in a microkernel-based design. This question has been answered in the assertive many years ago, going back to the multi-personality Workplace OS built on Mach [Fleisch et al. 1998], the POSIX API provided by QNX [Leroux 2011], and more recently Google's Fuchsia OS [Google 2018] which is based on the Zircon microkernel.

2.4 OS verification and seL4

The seL4 microkernel has been designed for security from the ground up. It uses access control based on capabilities [Dennis and Van Horn 1966], which provides fine-grained control over access rights in the system, and enables a true least-privilege design.

More importantly, seL4 demonstrated that small is beautiful: Owing to its small size, it was feasible to formally prove seL4's implementation correct, and use that as the enabler of further proofs, including integrity and confidentiality enforcement [Klein et al. 2014]. seL4 is also the only protected-mode OS in the literature that has undergone a sound and high-assurance analysis of worst-case execution times [Blackham et al. 2011; Sewell et al. 2017].

As such, seL4 is presently the best approximation to a small TCB that is almost impossible to compromise, and hence the most promising base for building secure systems on top. In our study we will use a hypothetical seL4-based OS with the complete functionality of Linux to determine whether vulnerabilities can be avoided by OS design.

2.5 Vulnerability assessment: CVSS

The *common vulnerability scoring system* (CVSS) [FIRST 2015] assigns a vulnerability score in the range 0.0 to 10.0, where vulnerabilities with a score 9.0 or above are considered *critical*.

The overall score is composed from three major metrics, *base*, *temporal* and *environmental*.

The *base* metric is scored according to:

Exploitability: This scores the ease of exploiting the vulnerability, taking into account the attack vector (remote vs local vs physical access), the complexity of the attack (no preconditions vs serendipity), and the privileges required to carry it out.

Scope: Assesses whether the exploit is confined to a single component or goes beyond the compromised component.

Impact: Assesses the degree to which *confidentiality* (C), *integrity* (I) or *availability* (A) are impacted. Confidentiality is impacted if the exploit allows an attacker to steal any information, integrity is violated if the attacker can affect unauthorised modification of data, and availability is affected if the attacker can crash the system or the attacked application, or otherwise prevents the progress of an application's execution.

The *temporal* metric quantifies:

Exploit maturity: Measures the degree to which the details of the exploit are publicly available, or even incorporated into readily-accessible tools.

Remediation level: Assesses the availability and completeness of defences.

Report confidence: Assesses the degree to which the exploit is verified.

The *environmental* metrics measures:

Security requirements: Assesses the importance of the affected asset to the organisation in terms of the CIA properties.

Modified base: A catch-all metric for adjusting the base metric to special circumstances.

The full list of vulnerabilities can be found at the Mitre CVE site [Mitre 2018]. A *critical CVE* is characterised as simple to exploit, of high impact, too recent for defences to be available, and verified.

3 METHODOLOGY

Our aim is to determine how switching from a monolithic kernel to an OS based on a (verified) microkernel would affect the severity of real Linux exploits. In order to assess this, we assert that it is the kernel's core duty to provide isolation between unrelated applications that are executing concurrently: *One malicious application must not be able to compromise the confidentiality, integrity or availability of another application.* An exploit violates this isolation. In this

work we evaluate to which degree this violation is reduced or eliminated by a changed OS structure.

3.1 Threat model

In a monolithic OS, compromising one (kernel-provided) service compromises the whole system, therefore the whole multi-million-SLOC kernel is in every application's TCB. In contrast, in a microkernel-based system, the TCB of an application becomes highly dependent on the system services it uses. We need to define this set of services in order to investigate the effect of the reduced TCB of the microkernel-based system.

Specifically, we assume a hypothetical, bug-free application that performs a security-critical function. This application will require access to an IP stack, persistent storage and display output. While display output is not necessary for many programs, an attacker who gains control over the display could trick a user into further compromising their system, so we include it in our set of resources that must not be compromised. An attack through a compromised display might be very hard if not infeasible in many cases, so this is a conservative requirement which might overestimate the severity of vulnerabilities in our hypothetical application.

In all other respects, the threat model of a CVE under examination remains unchanged.

3.2 Classification

We define five mitigation categories, and assign each exploit to one of these categories. The mitigation scores indicate how the operation of our hypothetical, security-critical application exploit would be affected by a change in OS design.

The critical question to ask is whether a particular exploit could impact an OS service implemented in userspace (as opposed to only being effective if compromising the kernel) and if so, which privileges the affected component would require, and which of the CIA properties are affected.

For each exploit we consider its worst-case effect. For example, the description of many exploits indicate a possibility for arbitrary code execution. We assume that the exploit does indeed allow execution of arbitrary code.

The mitigation categories are as follows (we give specific examples in Section 4.1).

Eliminated by Microkernel (Yes): A vulnerability is placed into this category if our hypothetical application would be unaffected by this vulnerability as long as the exploit happened in a microkernel-based operating system. The attacker would not be able to violate any of the application's CIA properties. This implies that the exploit must not compromise any of the services our program requires (IP stack, files on persistent storage, display output). Vulnerabilities in this category tend to be on services which could be implemented in userspace with limited privileges.

Eliminated by Formal Verification (FV): This category is similar to the previous one, except that the exploit could affect the microkernel itself, unless the kernel is formally verified (as is the case for seL4). Vulnerabilities in this category are those which affect functions that *must* be performed by the kernel, e.g. page-table management.

Partially mitigated by Microkernel (CIA): In order to be placed in this category, a vulnerability must have compromised an operating system component which is used by our application but not necessarily the whole system. We furthermore identify which security properties of the application are compromised, i.e. the subset of {C,I,A}. We assume that the application takes steps to minimise its dependency on services provided by user-level servers. For example, an application can protect itself against a compromised IP stack if it encrypts and signs all data sent over the network. However, the IP stack could still attack availability by dropping all packets. We would assign such an exploit an "A" score.

No Impact (No): Exploits in this category are unaffected by a change in OS design. This includes hardware and bootloader vulnerabilities in addition to services that require kernel mode but are not implemented by seL4.

Insufficient Information (???): A vulnerability placed in this category did not have sufficient information to determine in which of the above categories if belongs.

3.3 Process

We perform the classification by having two authors independently examine each exploit and assign a mitigation score. Where the assessments differ, the third author examines the exploit to determine the final score.

3.4 Threats to validity

The following factors could, despite our best efforts, influence our study in a way that would make our observations misleading.

- Linux critical CVEs might not be representative of monolithic systems. This would mean that our results, while valid for Linux, would not generalise to other monolithic systems. We consider the impact of this threat **very low**: While the critical vulnerabilities we study are specific to Linux, in almost all cases similar functionality exists in the kernel of other monolithic OSes, such as Windows and MacOS. An exploit against similar functionality would most likely be of

comparable severity, as in each case kernel code is compromised.

- CVE scores are strongly affected by remote exploitability. Consequently, there may be an over-representation of network stack vulnerabilities in critical CVEs, which may bias the scores, and make the results of our study not fully representative of the most serious security threats. However, as our hypothetical application depends on network connectivity, such a networking bias would lead to our mitigation assessments being pessimistic, i.e. *underestimate* the average benefit of a microkernel-based design. We therefore consider the impact of this threat **low**.
- Critical CVEs might not be representative of all serious vulnerabilities. It is conceivable that severe (in contrast to critical) exploits might exhibit different properties in respect to how their severity and exploitability is dependent on OS structure. We consider the impact of this threat **low**: Without further study it cannot be ruled out that the nature of severe threats are different, but there is no reason to suspect that they are radically different, and in any case, critical vulnerabilities are the most concerning.
- Proper isolation of services might not be feasible, thus diminishing the benefits of microkernel-enforced exploit-containment. We consider the impact of this threat **low to medium**, as there are existing systems, such as Minix [Tanenbaum 2016] and especially QNX [Leroux 2011], which is routinely deployed in safety-critical systems, show that this isolation is feasible.
- Applications might not be designed to make full use of microkernel-based isolation. As explained in Section 3.2, we assume that applications are designed to minimise damage resulting from a compromised systems service. This is clearly a best-case scenario, and most present apps will not necessarily comply with this assumption. This is not surprising, as they are designed to run on monolithic systems, where there is no choice but to trust the whole OS, and there is no benefit from minimising exposure to the OS. This does not affect the validity of our results, which are examining the degree of security that is *achievable* in a microkernel-based OS. Hence we rate this threat **low**.
- We noted in Section 3.1 that requiring the integrity of the display was a conservative assumption that might overestimate the severity of a compromise of the microkernel-based system in many cases. If so, it would lead us to *underestimate* the security benefits of a microkernel-based approach, we therefore rate this threat as **low**.

4 RESULTS

There were a total of 115 critical CVEs for Linux as of late 2017, we assess all of them. We first provide a few examples, and then present the overall results.

4.1 Examples

4.1.1 Eliminated by Microkernel. An example of this category is CVE-2015-4001, which describes an integer signedness error in the OZWPAN driver. This driver is a USB host controller device driver which does not have a hardware device associated with it, but instead is used to communicate with a wireless peripheral over Wi-Fi. The integer signedness error can lead to the result of a subtraction becoming negative, causing a `memcpy` operation to interpret the value as an intention to copy large amounts of network-supplied data into a heap buffer. An attacker can insert a payload into a crafted packet to trigger the error and inject data. Since Linux loads the driver into the kernel, it could cause a denial of service by crashing the kernel, or could possibly execute arbitrary code with kernel privileges.

In a microkernel-based system, the driver would run as a user-level server in its own address space, and as such could not overwrite kernel memory and cause a system crash, information leakage or corruption. Furthermore, any code injection would only execute with the minimal privileges required by the driver. In a well-designed microkernel-based system, this driver would only have the ability to communicate with a Wi-Fi user-level server to interact with the device and with applications using it, but little more. Therefore, this exploit would not affect the security of our hypothetical application.

4.1.2 Eliminated by Formal Verification. CVE-2014-9803 describes an exploit where the Linux kernel on some Nexus devices mishandled execute-only pages, which allowed a crafted application to gain kernel privileges. As this operation must be performed in kernel mode, it could equally occur in a microkernel. However, in a formally verified microkernel, such as seL4, this bug could not occur.

4.1.3 Partially Mitigated by Microkernel. CVE-2015-8961 describes an exploit in the `__ext4_journal_stop` function. On Linux it can result in full file-system disclosure (C) or a kernel crash (A).

On a microkernel, the file system would be implemented as a user-level server, and this exploit would not result in a kernel crash. However since an attacker may gain access to all files, an application which stored confidential data on persistent storage could be compromised. If the data written to persistent storage was encrypted with a key that was not accessible by the file system (for example, entered by the user) then an attacker could not use this compromise to

CVE	MS	CVE	MS	CVE	MS	CVE	MS	CVE	MS	CVE	MS
1999-0461	A	2011-3191	A	2015-8967	Yes	2016-6789	Yes	2016-8437	FV	2017-0429	IA
1999-0590	No	2012-1146	FV	2016-2061	Yes	2016-6790	Yes	2016-8438	CA	2017-0430	CIA
2000-0506	Yes	2014-0100	A	2016-2062	IA	2016-7117	CIA	2016-8439	Yes	2017-0455	No
2002-1572	Yes	2014-2523	CA	2016-2063	Yes	2016-7910	CIA	2016-8440	FV	2017-0507	Yes
2002-1573	Yes	2014-2648	Yes	2016-2065	Yes	2016-7911	A	2016-8455	Yes	2017-0508	Yes
2003-0959	FV	2014-9803	FV	2016-2066	Yes	2016-7912	CA	2016-8459	IA	2017-0510	No
2004-1017	FV	2014-9870	FV	2016-2067	IA	2016-7913	Yes	2016-8479	IA	2017-0528	FV
2004-1137	A	2015-0312	Yes	2016-2068	Yes	2016-8398	A	2016-9120	Yes	2017-0561	CIA
2006-1368	A	2015-0569	CA	2016-3955	A	2016-8424	IA	2016-9313	Yes	2017-0563	No
2006-1523	FV	2015-0570	CA	2016-5344	Yes	2016-8425	IA	2016-9555	A	2017-0564	Yes
2006-1857	CA	2015-0571	Yes	2016-6758	Yes	2016-8426	IA	2016-9644	FV	2017-0605	FV
2006-6535	A	2015-0573	Yes	2016-6759	Yes	2016-8427	IA	2016-10150	Yes	2017-0648	No
2008-1673	CA	2015-1421	A	2016-6760	Yes	2016-8428	IA	2017-0306	IA	2017-7895	A
2008-3496	Yes	2015-3331	Yes	2016-6761	Yes	2016-8429	IA	2017-0307	IA	2017-8890	A
2008-3915	A	2015-4001	Yes	2016-6775	IA	2016-8430	IA	2017-0333	IA	2017-11176	FV
2008-5134	A	2015-4002	Yes	2016-6776	IA	2016-8431	IA	2017-0335	IA		
2009-0065	A	2015-8787	A	2016-6777	IA	2016-8432	IA	2017-0337	IA		
2009-4538	A	2015-8812	CIA	2016-6781	???	2016-8434	IA	2017-0338	IA		
2010-2495	A	2015-8961	CIA	2016-6782	???	2016-8435	IA	2017-0427	CA		
2010-2521	CIA	2015-8962	CIA	2016-6785	???	2016-8436	IA	2017-0428	IA		

Table 1: Mitigation scores (MS) for all critical Linux CVEs (IDs linked to details pages). Key: "Yes": prevented by microkernel-based design; "FV": prevented by formally verified kernel; "CIA" (or subset): mitigated to loss of confidentiality/integrity/availability of particular service; "No": unaffected by microkernel-based design; "???": insufficient information.

read confidential file data. Here the hypothetical application would still be secure if it had been designed to not trust the file system. However, the attacker can still crash the file system, attacking availability. Consequently we assign an "A" score to this vulnerability.

4.1.4 No Impact. An example of a vulnerability where even a formally verified microkernel cannot help is CVE-2017-0563. This vulnerability was an elevation-of-privilege exploit within the touchscreen driver. The driver requires access to the I^2C bus[1] within the phone. With access to this bus it is possible to re-flash the system-on-chip firmware or bootloader, both of which were unsigned.

By re-flashing bootloader or firmware, the attacker gains control before the kernel starts executing, and can therefore compromise even a formally verified kernel (breaking the verification assumptions of hardware operation).

4.2 Overall results

Table 1 shows the complete results, assigning a mitigation score to each of the 115 critical Linux vulnerabilities.

[1] I^2C (pronounced I-squared-C) is a multi-master, multi-slave serial computer bus which was invented by Phillips Semiconductor.

Table 2 summarises the results according to the mitigation categories. Here we further split the partially-mitigated category into *strongly* and *weakly* mitigated. We consider the mitigation strong if the exploit can only effect denial of a particular service (e.g. IP networking). If the mitigated compromise can still violate confidentiality or integrity, we consider the mitigation weak.

Of all CVEs, only three (3%) lacked information needed for classification. We exclude this small fraction from Table 2, the percentage figures here refer to the fraction of the 112 CVEs which we could classify. We find that 29% of all critical Linux vulnerabilities would be completely prevented in a

Assessment	Short	#	Fract.	Cumul.
Eliminated	Yes	33	29%	29%
Eliminated with verification	FV	12	11%	40%
Strongly mitigated	A	19	17%	57%
Weakly mitigated	C/I	43	38%	96%
Unaffected	No	5	4%	100%
Total:		112	100%	100%

Table 2: Summary of findings.

microkernel-based system, and a further 11% (a total of 40%) if the microkernel was formally verified, like seL4.

A further 17% of exploits are strongly mitigated, resulting in only the loss of one particular non-essential service, meaning that a total of 57% of critical Linux exploits would be reduced to a low severity level.

Only 38% of exploits still lead to confidentiality or integrity violations in a microkernel-based design. Even these weakly mitigated vulnerabilities would no longer lead to full system compromise, and therefore no longer be rated critical. Hence, only the 4% "unaffected" vulnerabilities remain critical in our microkernel scenario.

5 CONCLUSIONS

We have presented what is, to the best of our knowledge, the first quantitative empirical assessment of the security implications of operating system structure, i.e. monolithic vs microkernel-based design.

Our results provide very strong evidence that operating-system structure has a strong effect on security. 96% of critical Linux exploits would not reach critical severity in a microkernel-based system, 57% would be reduced to low severity, the majority of which would be eliminated altogether if the system was based on a verified microkernel. Even without verification, a microkernel-based design alone would completely prevent 29% of exploits.

Given the limited number of documented exploits, we have to assume our results to have a statistical uncertainty of about nine percentage points. Taking this into account, the results remain strong. The conclusion is inevitable: **From the security point of view, the monolithic OS design is flawed** and a root cause of the majority of compromises. It is time for the world to move to an OS structure appropriate for 21st century security requirements.

ACKNOWLEDGMENTS

We thank Charles Gray for suggesting this study, and Yuval Yarom and the anonymous reviewers for helpful feedback.

REFERENCES

Fahim Ahmed. 2016. How many lines of code does Windows 10 contain? https://www.quora.com/How-many-lines-of-code-does-Windows-10-contain

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. 2011. Timing Analysis of a Protected Operating System Kernel. In *IEEE Real-Time Systems Symposium*. Vienna, Austria, 339–348.

Per Brinch Hansen. 1970. The Nucleus of a Multiprogramming Operating System. *Commun. ACM* 13 (1970), 238–250.

J. Bradley Chen and Brian N. Bershad. 1993. The Impact of Operating System Structure on Memory System Performance. In *ACM Symposium on Operating Systems Principles*. Asheville, NC, US, 120–133.

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9 (1966), 143–155.

Department of Defence 1986. *Trusted Computer System Evaluation Criteria*. Department of Defence. DoD 5200.28-STD.

Edsger W. Dijkstra. 1968. The Structure of the "THE" Multiprogramming System. *Commun. ACM* 11 (1968), 341–346.

FIRST. 2015. CVSS v3.0 Specification Document. https://www.first.org/cvss/specification-document

Brett D. Fleisch, Mark Allan A. Co, and Chao Tan. 1998. Workplace Microkernel and OS: A Case Study. *Software: Practice and Experience* 28 (1998), 569–591.

Google. 2018. Fuchsia is not Linux. https://fuchsia.googlesource.com/docs/+/HEAD/the-book/README.md

Les Hatton. 1997. Re-Examining the Fault Density - Component Size Connection. *IEEE Software* 14, 2 (1997), 89–97.

Gernot Heiser and Kevin Elphinstone. 2016. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Transactions on Computer Systems* 34, 1 (April 2016), 1:1–1:29.

Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. 2004. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the 11th SIGOPS European Workshop*. Leuven, BE.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Transactions on Computer Systems* 32, 1 (Feb. 2014), 2:1–2:70.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *ACM Symposium on Operating Systems Principles*. ACM, Big Sky, MT, USA, 207–220.

Paul Leroux. 2011. Secure by Design: Using a Microkernel RTOS to Build Secure, Fault-Tolerant Systems. http://www.qnx.com/content/dam/qnx/whitepapers/2009/qnx$_s$ecure$_k$ernel$_w$hitepaper$_R$IM$_M$C411.67.pdf.

Jochen Liedtke. 1993. Improving IPC by Kernel Design. In *ACM Symposium on Operating Systems Principles*. Asheville, NC, USA, 175–188.

Jochen Liedtke. 1995. On μ-Kernel Construction. In *ACM Symposium on Operating Systems Principles*. Copper Mountain, CO, USA, 237–250.

John Mason. 2018. Cyber Security Statistics. https://thebestvpn.com/cyber-security-statistics-2018/

Mitre. 2018. Linux Kernel: CVE security vulnerabilities, versions and detailed reports. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor$_i$d=33

Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. 2004. An Empirical Study of Software Reuse vs. Defect-Density and Stability. In *International Conference on Software Engineering*.

John Rushby. 1981. Design and Verification of Secure Systems. In *ACM Symposium on Operating Systems Principles*. Pacific Grove, CA, USA, 12–21.

Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63 (1975), 1278–1308.

Thomas Sewell, Felix Kam, and Gernot Heiser. 2017. High-Assurance Timing Analysis for a High-Assurance Real-Time OS. *Real-Time Systems* 53 (Sept. 2017), 812–853.

Andrew S. Tanenbaum. 2016. Lessons Learned from 30 Years of MINIX. *Commun. ACM* 59, 3 (2016), 70–78.