PROTECT: Container Process Isolation using System Call Interception

Thu Yein Win*, Fung Po Tso‡, Quentin Mair†, Huaglory Tianfield†

*Faculty of Business, Computing and Applied Sciences, University of Gloucestershire, UK
†Department of Computer, Communication and Interactive Systems, Glasgow Caledonian University, UK
‡Department of Computer Science, Loughborough University, UK
Email: twin@glos.ac.uk; p.tso@lboro.ac.uk; q.mair@gcu.ac.uk; h.tianfield@gcu.ac.uk

***Abstract*—Virtualization is the underpinning technology enabling cloud computing service provisioning, and container-based virtualization provides an efficient sharing of the underlying host kernel libraries amongst multiple guests. While there has been research on protecting the host against compromise by malicious guests, research on protecting the guests against a compromised host is limited. In this paper, we present an access control solution which prevents the host from gaining access into the guest containers and their data. Using system call interception together with the built-in AppArmor mandatory access control (MAC) approach the solution protects guest containers from a malicious host attempting to compromise the integrity of data stored therein. Evaluation of results have shown that it can effectively prevent hostile access from host to guest containers while ensuring minimal performance overhead.**

***Keywords*-Virtualization Security, Cloud Security, Container Virtualization, Access Control, System Call Interception**

I. Introduction

The elasticity of resource allocation in cloud computing is mainly attributed to virtualization. The most prevalent form of virtualization is through a hypervisor or Virtual Machine Monitor (VMM). Hypervisors are assumed to be trustworthy in maintaining isolation using built-in operating system (OS) security tools. However provisioning of isolation among them tends to incur significant performance overhead, due to its large trusted computing base (TCB).

Container-based systems however only implement partial isolation supporting a shared namespace augmented with an access control mechanism that limits the ability of one guest virtual machine (VM) to manipulate the object owned by another VM [5]. While this allows for a light-weight alternative, the isolation amongst guests tends to be weak.

We have recently discovered two vulnerabilities in which system calls that are not namespace aware can be exploited: (1) the ability to gain access to the guest containers' internal file-system through a running container process, and (2)the ability to identify the processes running within the guest container.

In this paper we propose a novel container virtualization security approach which addresses these vulnerabilities. Using system call interception with built-in AppArmor mandatory access control (MAC), it ensures the isolation between a compromised host and guest containers.

In summary, the contribution of this paper is threefold:

- We present two vulnerabilities to demonstrate a flawed namespace isolation in the latest Linux kernel for container virtualization.
- We design and implement a security solution which tackles the namespace-unaware system calls vulnerabilities using system call interception.
- We extensively evaluate our scheme and show that it can efficiently address these vulnerabilities with minimal system overhead.

The rest of this paper is organized as follows. Section II provides an overview of the two main virtualization technologies, namely hypervisor-based virtualization and container-based virtualization. Section III provides a detailed discussion on the two recently-discovered security vulnerabilities in container-based virtualization solutions, before discussing the security approaches in existing research in Section IV. The design rationale and implementation details are discussed in depth in Section V, with its performance discussed in Section VI. Section VII compares our approach with other virtualization security solutions in existing research, before arriving at the conclusion in Section VIII.

II. Virtualization Technologies

Virtualization facilitates the sharing of a server's physical resources through resource emulation and can be broadly categorized into hypervisor-based virtualization, and container-based virtualization.

Hypervisor-based virtualization enables the sharing of a server's physical resources through device emulation. Each guest VM runs its own operating system, with the hypervisor regulating their access to the underlying physical resources. While this provides isolation between the guest VMs and the host, it tends to incur significant performance overhead as the number of guests increases.

Container-based virtualization overcomes this through virtualizing the host operating system kernel and libraries. This enables multiple guest containers to run on top of the host operating system without having to install guest

operating systems themselves [10]. Isolation between the guest containers and the host operating system is achieve through the use of kernel namespaces and control groups (*cgroups*).

Different components in a Linux operating system are organized into six global namespaces: file-system, network, user, IPC (inter-process communication), hostname and process namespaces [8]. Using the `chroot` command, kernel namespace enables the host operating system's global namespaces to be shared across multiple guest containers [6].

While kernel namespaces allow a container to have its own root directory and processes, control groups (*cgroups*) facilitate the fine-grained allocation of physical resources to a container [6].

Compared to hypervisor-based virtualization, container-based virtualization provides reduced performance overhead [7] and eliminates the need to install an additional kernel for the guests.

III. NAMESPACE VULNERABILITIES

In current Linux kernels, the namespaces and control groups (*cgroups*) provide resource isolation between host and guest containers. However, the host can be subjected to a "root break out" attack. This allows a root user running in the guest container to break out and become the root user of the host itself [11] [12].

While application sandboxing techniques such as SECCOMP (Secure Computing mode) have been built into container-based virtualization solutions such as Linux Containers (LXC), they are limited in preventing the host from identifying the processes running within the guest containers and getting access to their root directories via the *proc* directory.

We have recently discovered two security vulnerabilities, namely *host break-in* and *illegal container process manipulation*, which violate the isolation principle between the host and the guest container. While the former allows a root on the host to access a guest container's file-system through an opened container process, the latter allows the host to identify any running container processes and manipulate their states.

A. Host break-in

Given a guest container `Container1` running on the host `LXCHost`, a user space program `nano` with *pid* 1517 is executed on it. In the container's home directory, there is a text file called *helloworld.txt* containing the text as shown in the following command line snippet.

```
ubuntu@Container1:~ $ nano &
[1] 1517
ubuntu@Container1:~ $ cat helloworld.txt
Hello world from root!!
ubuntu@Container1:~ $
```

Next we launch the *host break-in* attack by running the `ps` command to list the current running processes, and filtering out the guest container processes by their AppArmor security context (which is `lxc_container_default`). This allows the user to identify the process's PID and use it to locate its entry in the `proc` directory. In this example, the *helloworld.txt* file in the guest container has a PID of 5612 and can then be accessed from the host via `/proc/5612/root/home/ubuntu/helloworld.txt`. The following command line snippet demonstrates that the *host break-in* attack has been successfully launched and the text file *helloworld.txt* within the guest has been accessed.

```
thu@LXCHost:~ $ ps -eZ | grep \
> lxc_container_default | grep nano
lxc-container-default 5612  ... nano
thu@LXCHost:~ $ cd /proc/5612/root
thu@LXCHost:/proc/5612/root$ sudo cat /home
/ubuntu/helloworld.txt
Hello world from root!!
thu@LXCHost:/proc/5612/root$
```

B. Illegal Container Process Manipulation

This second vulnerability presents a more significant threat to the guest container, as the host user does not need to be root in order to trigger the vulnerability. Using the same scenario, a user space program `nano` is executed in the guest container's user space as shown in the following verbatim:

```
ubuntu@Container1:~ $ nano &
[1] 1517
```

Through the same technique as mentioned in the previous subsection, the user on the host can identify the `nano` process running within the guest container. Any regular (even non-root) users can the use the obtained PID to terminate it by executing the `kill` command from the host terminal as shown in the following verbatim:

```
thu@LXCHost:~ $ ps -eZ | \
> grep lxc-container-default
lxc-container-default
5221  pts/2  00:00:00  nano
thu@LXCHost:~ $ kill -9 5221
thu@LXCHost:~ $
```

This results in the termination of the `nano` process which was previously running in the guest container as shown below:

```
ubuntu@Container1:~ $
[1]+ Killed nano
ubuntu@Container1:~ $
```

C. Potential implications of these vulnerabilities

The presence of the above security vulnerabilities presents significant impediments to the widespread adoption of

container-based virtualization as an alternative to hypervisor-based virtualization.

Using the host break-in vulnerability, the host can access the guest container files and manipulate them without the guest's knowledge. In addition, the host can use this to install a malicious shell script into the guest container and manipulate its `rc.local` file executed on startup. This presents a threat to the integrity and privacy of the data stored within the guest containers.

Similarly the presence of the illegal container process manipulation vulnerability violates the host-guest isolation, as the host is able to alter a guest container's process state through commands such as `kill`.

IV. Access Control and System Call Table

The proposed approach features the use of three main built-in OS mechanisms, namely Mandatory Access Control (MAC), AppArmor, and the system call table.

A. Mandatory Access Control (MAC)

Using the Bell-LaPadula (BLP) access control model [13] a MAC-based access control solution typically consists of three components, namely reference monitor, enforcement hooks and access control policies.

1) Reference monitor: The reference monitor is a security module which is responsible monitoring all resource access requests [15]. It uses the access enforcement hooks to intercept any resource access request and grants access based on a set of pre-defined access policies [16].

2) Access enforcement hooks: Placed at critical points within the kernel, access enforcement hooks are invoked when a user or a process makes a resource access request. They intercept any resource access requests and pass them to the reference monitor.

3) Access control policies (ACM): Defined by the security administrator, access control policies are a set of rules which determine the resources which a user is allowed access to.

B. AppArmor

AppArmor is a Mandatory Access Control (MAC) solution designed to provide fined-grained resource access control.

AppArmor follows a file path-based approach (as opposed to the label-based approach as used in SELinux) in defining access control policies for processes. An AppArmor profile contains the directory paths which a given process can access as well as the operations (i.e., read, write, execute, etc) that can be done within them.

C. System call table

Exported during the kernel compilation process, it is a kernel data structure which contains pointers to the various system call functions which are scattered across different locations in the kernel space.

When a process requires the execution of a system call from the kernel, it places the system call number in the EAX register (RAX register) and places the required system call parameters onto the subsequent registers (i.e., EBX, ECX, etc). It then triggers the SYSENTER instruction against the host CPU, resulting its moving into the more privileged Ring-0 and triggering the trap handler. The handler places the system call arguments on to the kernel stack, before looking up the system call table using the value in the EAX register to determine the system call to be triggered. The system call function is then triggered by passing the values in the kernel stack to the function.

D. Limitations of existing access control techniques

While the existing techniques are effective in controlling guest resource access in a virtualization environment, they are limited in a number of ways.

One of the limitations is that the existing access control measures are designed with the assumption that the guest VM is the source of attack. However, they are limited in detecting malicious access from the host. Given that containers run on the host user space, an attacker can compromise the host through a software vulnerability and in turn manipulate the container state.

In addition, the existing solutions typically leverage the access control mechanisms which are built into the hypervisor. Virtualization platforms such as KVM and Xen use sVirt and XSM (Xen Security Modules) respectively to enable the system administrator to define access control policies using SELinux (Security-enhanced Linux) providing a bidirectional access control between the host and the guest VM. While containers use sandboxing tools such SECCOMP (Secure Computing mode) to control the system calls which guest containers can issue, it does not provide the same functionality. This allows a malicious host to gain illegal guest resource access using the aforementioned vulnerabilities.

V. Design and Implementation

A. System Architecture

Since the ultimate goal of container virtualization is to provide highly efficient virtualization with low overhead, we have considered the following design guidelines in the design of our proposed approach:

- $r1$ - Efficiency: The system should not incur too much overhead to impact the efficiency of the system and container.
- $r2$ - Transparency: Both host and container should not be aware of the existence of the system.
- $r3$ - Scalability: The system should be able to scale when the number of container increases.
- $r4$ - Deployability: The system should be readily deployable in a production environment with minimal effort.

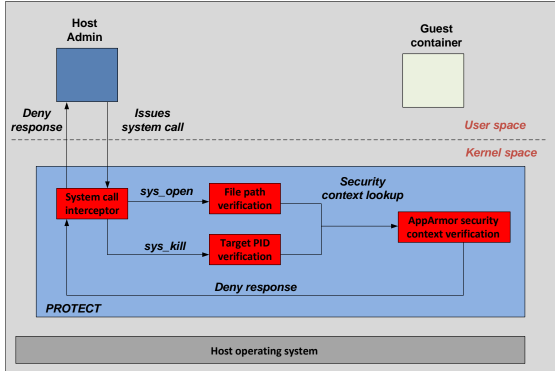

Figure 1: *PROTECT* Architecture

The implementation of our proposed approach as a kernel module on the host provides a single point of control to monitor host attempts to access guest containers while ensuring minimal performance overhead, satisfying $r1$ (efficiency) as well as $r3$ (scalability).

Moreover, having a kernel-based container access control ensures a solution that does not require changes to the Linux Containers (LXC) source code, hence being completely transparent to the containers ($r2$).

Our idea is similar to but distinctively different from the hypercall interception approach used in hypervisor-based virtualization security solutions [17][18]. One of the drawbacks of hypercall interception is that it requires the hypervisor source code to be recompiled for every hypercall modification which is infeasible in a multi-host virtualization environment. By implementing our proposed approach as a Loadable Kernel Module (LKM), we can easily deploy it without code modification ($r4$).

B. Implementation

Our proposed approach is composed of four components which are namely the *system call interceptor*, *file path verification*, *target process verification*, and *AppArmor security context verification* as shown in Figure 1.

1) System call interceptor: The system call interceptor is responsible for monitoring any attempts by the host to open files as well as to terminate a running process are monitored, by intercepting the `sys_open` and `sys_kill` system calls respectively.

In order to intercept the `sys_open` and `sys_kill` system calls, the system call table `sys_call_table` entries containing the addresses of their respective functions are first located. Their original addresses in the structure are then replaced with the system call functions of our proposed approach.

2) File path verification: To determine if an opened file belongs to a guest container, the `sys_open` system call is intercepted and the opened filepath is extracted from it. For example, when a user opens a file `proc/111/root/home/ubuntu/helloworld.txt`, the underlying `libc` library triggers the `sys_open` system call by passing the function arguments as follows: `sys_open("proc/111/root/home/ubuntu/helloworld.txt", O_RDONLY)`. The first two sections (i.e., `proc` and `111`) are extracted from the function argument, before being concatenated to read the process's `current` file containing the AppArmor security context of the process. The host is denied access to the target file if it belongs to a guest container process.

3) Target PID verification: To prevent the host from altering the state of running guest container processes, our proposed approach monitors the host's attempts to trigger the `sys_kill` system call for process termination. For instance when the user issues the command `kill -9 111`, the command arguments are passed to the `sys_kill` as such: `sys_kill (111, -9)`. Once triggered, the target process ID (i.e., `111`) is extracted and is used to determine its AppArmor security context by reading the `current` file as previously indicated.

4) AppArmor security context verification: Operating on the *principle of least privilege*, AppArmor restricts processes' access to the host files and directories using security profiles. The defined security profiles contain "path entries" [19] specifying the directories which the processes are allowed access to, along with their access rights to them. For instance a process such as `ls` is allowed access to the host system directories having been assigned the `unconfined` security context, but a guest container is restricted from doing so by being assigned the `lxc_container_default` security context. Based on this observation, our proposed approach denies denying host access to guest container's resources if the AppArmor security context input from the previous two components is `lxc_container_default`.

5) Access control: If any malicious events are detected using the aforementioned approach, our proposed approach denies access to the guest container file as well as its ability to alter its running process state by issuing the `EPERM` error.

VI. Experimental Evaluation

A. Experiment Setup

We have evaluated our implementation on a server with an Intel Xeon quadcore processor at 2.33 GHz along with 8GBs of memory, with Linux kernel version 3.18.18 (64-bit). The LXC (Linux Containers) platform was installed on the server node, with an Ubuntu guest container running on top of it.

B. Evaluation Results

1) Measuring `sys_open` execution time: To conduct this experiment, we first created an Ubuntu guest container on the host, with `nano` running in it. We then created a

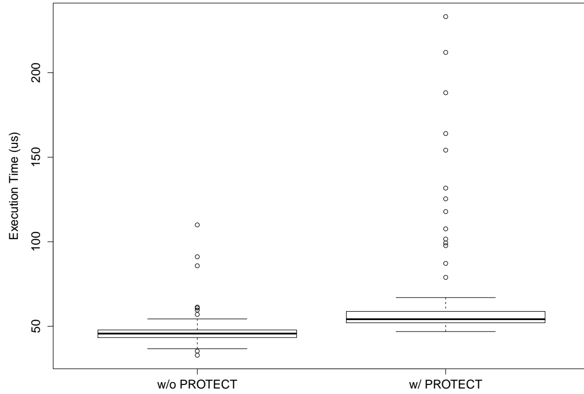

Figure 2: Comparison of execution time on *sys_open*

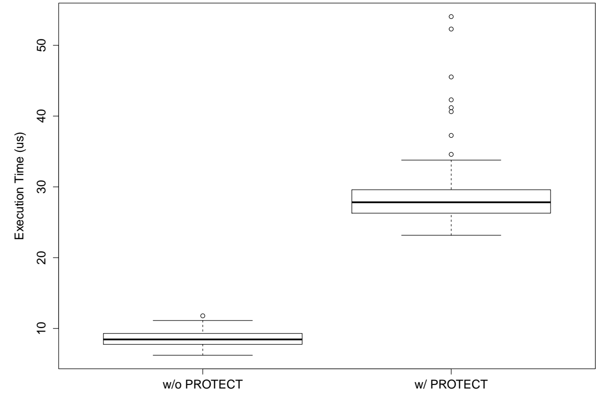

Figure 3: Comparison of execution time on *sys_kill*

small user space C program which uses the `sys_open` system call to open the `helloworld.txt` file located on the guest. We executed the program 100 times, each time opening the file 10 times both with the proposed approach running as well as without it. The time taken for both cases is shown in the boxplot in Figure 2.

As expected, the amount of time taken to open the targeted container file is on average higher when our approach is running with the host kernel. The median execution time with the presence and absence of it are 45.4 microseconds and 54.1 microseconds respectively, indicating an additional execution time of 8.7 microseconds on average. The increase in time is due to the context-switching performed by the proposed approach between the kernel and user spaces to access the container file's AppArmor profile.

On the other hand, we can also observe that there are eight outliers in the proposed approach's execution time. We believe that these sudden spikes in execution time are down to the effect of soft interrupts from other system operations and is more profound due to kernel-user space context-switching. However, given that we have only observed eight outliers out of a thousand points, we argue that these spikes will only happen rarely and will not bring a noticeable side effect to the system.

2) Measuring `sys_kill` execution time: In order to measure the execution time taken by the `sys_kill` system call, we developed another small user space C program which kills a running process using the `kill` function given its process id (PID). During the experiment, we used the program to terminate the `nano` process running in an Ubuntu guest container. The program was run 100 times both in the presence of the proposed approach as well as in its absence, and the execution times for both are shown accordingly in the boxplots in Figure 3.

At first glance, we can notice that there is a significant increase in execution time when the proposed approach is running within the host. More specifically, the median execution times are 8.4 microseconds and 27.8 microseconds with and without it respectively, representing an increase of 19.4 microseconds, or 230%. Similar to the case in monitoring `sys_open`, the overhead increase is due to our proposed approach having to perform context switching between the user space and kernel space in order to determine the target PID's security context before granting host access. Since the `sys_kill`'s routing is extremely simple and fast, the additional time for context switching becomes more profound.

Despite this, we note that `sys_kill` is often only activated on user demand meaning that the performance overhead associated with monitoring it will not affect the overall system performance as it is too small to be noticeable by users.

VII. Related Work

There are a few previous studies that have examined the performance of containers in various scenarios. [20] put VServer, OpenVZ, and LXC as well as Xen in a HPC (High Performance Computing) environment and found that that all container-based systems demonstrate a near-native performance of CPU, memory, disk and network. However, all types of containers tested have shown poorer isolation as compared with Xen.

While [20][7][6] all reported reduced levels of isolation by containers as compared with hypervisors, and hence are more vulnerable to malicious events, little has been done to address this issue. Both [11] and [12] reported the incidents in which malicious containers can gain *root* privilege due to imperfect process isolation. The current industry solution has been to mainly leverage the existing techniques we presented in Section IV[21][22]. However such techniques only provide better isolation between containers and do not efficiently prevent a compromised host from performing

malicious actions on the containers as we demonstrated in Section III. Our proposed approach addresses this type of vulnerability by intercepting and analysing malicious system calls in the kernel space.

VIII. Conclusion

While significant effort has been expanded in isolating hosts and containers, i.e., preventing containers' process gaining access to the host, limited work has been done preventing users who have access to a physical host gaining direct access into containers. In this paper, we presented two flawed namespace vulnerabilities and demonstrated that they can be easily exploited to gain direct access to containers' internal running processes.

In order to fully protect containers from malicious host activities, we presented a new access control solution which features the use of system call interception while leveraging the existing AppArmor MAC solution.

We have also extensively evaluated our approach under different scenarios. Our experimental results have revealed it was able to regulate illegal host access to guest containers while incurring negligible performance overhead.

Acknowledgment

The second author would like to acknowledge the support provided to him by the UK Engineering and Physical Sciences Research Council (EPSRC) grants EP/P004407/1 and EP/P004024/1.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 199–212.

[3] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: security and functionality in a commodity hypervisor," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 189–202.

[4] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "Hypersentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 38–49.

[5] "Linux containers." [Online]. Available: https://linuxcontainers.org/lxc/introduction/

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *technology*, vol. 28, p. 32, 2014.

[7] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3. ACM, 2007, pp. 275–287.

[8] R. Rosen, "Linux containers and the future cloud," *Linux J*, vol. 240, 2014.

[9] C. Babcock, "Docker: Less controversy, more container adoption in 2015," 2015. [Online]. Available: http://www.informationweek.com/cloud/platform-as-a-service/docker-less-controversy-more-container-adoption-in-2015/d/d-id/1318771

[10] G. Calarco and M. Casoni, "On the effectiveness of linux containers for network virtualization," *Simulation Modelling Practice and Theory*, vol. 31, pp. 169–185, 2013.

[11] N. Sapple, "Linux local privilege escalation via suid /proc/pid/mem write," 2012. [Online]. Available: http://blog.zx2c4.com/749

[12] S. Krahmer, "shocker: docker poc vmm-container breakout," 2014. [Online]. Available: http://stealth.openwall.net/xSports/shocker.c

[13] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," DTIC Document, Tech. Rep., 1973.

[14] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *Communications Magazine, IEEE*, vol. 32, no. 9, pp. 40–48, 1994.

[15] J. P. Anderson, "Computer security technology planning study. volume 2," DTIC Document, Tech. Rep., 1972.

[16] Ú. Erlingsson, "The inlined reference monitor approach to security policy enforcement," Cornell University, Tech. Rep., 2003.

[17] C. Li, A. Raghunathan, and N. Jha, "A trusted virtual machine in an untrusted management environment," *Services Computing, IEEE Transactions on*, vol. 5, no. 4, pp. 472–483, Fourth 2012.

[18] C. H. H. Le, "Protecting xen hypercalls: Intrusion detection/prevention in a virtualization environment," 2009.

[19] M. Bauer, "Paranoid penguin: an introduction to novell apparmor," *Linux Journal*, vol. 2006, no. 148, p. 13, 2006.

[20] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Feb 2013, pp. 233–240.

[21] "Docker security," 2015. [Online]. Available: https://docs.docker.com/articles/security/

[22] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, "Security of os-level virtualization technologies," in *Secure IT Systems*. Springer, 2014, pp. 77–93.