Undermining Isolation through Covert Channels in the Fiasco.OC Microkernel

Michael Peter
Technische Universitaet Berlin
Security in Telecommunications
Email: peter@sec.t-labs.tu-berlin.de

Matthias Petschick
Technische Universitaet Berlin
Security in Telecommunications
Email: matthias@sec.t-labs.tu-berlin.de

Julian Vetter
Technische Universitaet Berlin
Security in Telecommunications
Email: julian@sec.t-labs.tu-berlin.de

Jan Nordholz
Technische Universitaet Berlin
Security in Telecommunications
Email: jnordholz@sec.t-labs.tu-berlin.de

Janis Danisevskis
Technische Universitaet Berlin
Security in Telecommunications
Email: janis@sec.t-labs.tu-berlin.de

Jean-Pierre Seifert
Technische Universitaet Berlin
Security in Telecommunications
Email: Jean-Pierre.Seifert@telekom.de

***Abstract*—In the new age of cyberwars, system designers have come to recognize the merits of building critical systems on top of small kernels for their ability to provide strong isolation at system level. This is due to the fact that enforceable isolation is the prerequisite for any reasonable security policy. Towards this goal we examine some internals of Fiasco.OC, a microkernel of the prominent L4 family. Despite its recent success in certain high-security projects for governmental use, we prove that Fiasco.OC is not suited to ensure strict isolation between components meant to be separated.**

Unfortunately, in addition to the construction of system-wide denial of service attacks, our identified weaknesses of Fiasco.OC also allow covert channels across security perimeters with high bandwidth. We verified our results in a strong affirmative way through many practical experiments. Indeed, for all potential use cases of Fiasco.OC we implemented a full-fledged system on its respective archetypical hardware: Desktop server/workstation on AMD64 x86 CPU, Tablet on Intel Atom CPU, Smartphone on ARM Cortex A9 CPU. The measured peak channel capacities ranging from ∼13500 bits/s (Cortex-A9 device) to ∼30500 bits/s (desktop system) lay bare the feeble meaningfulness of Fiasco.OC's isolation guarantee. This proves that Fiasco.OC cannot be used as a separation kernel within high-security areas.

I. Introduction

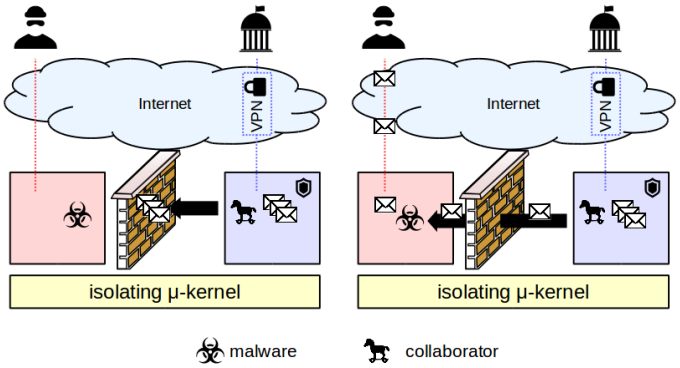
Over the past years, electronic devices have found their way into virtually every aspect of our daily lives, changing the way we communicate, work, and spend our leisure time. Even the latest wave – the triumph of smartphones and tablets – could qualify as a technological revolution. On the downside, it is obvious that radical changes of that magnitude involve many security risks. Already a severe threat before, malware has now a good chance to become one of the principal roadblocks for further innovations. These concerns are further fueled by the trend towards a very small number of commodity systems with ubiquitous network connectivity. Here, the former provides an enlarged attack surface while the latter exposes the respective systems to remote adversaries. As if the threat from criminal circles was not bad enough, the situation has been further worsened by targeted attacks — also known as cyberwar. Long on financial, technical, and even social resources and expertise, the entities behind these attacks were long thought to not have difficulties overcoming standard security measures, cf. [1]. Recent reports on highly advanced attacks corroborate this assumption [2]–[4].

To counter such attacks, efforts have been launched to improve the security of existing operating systems, with SELinux being a prominent example, cf. [5]. Not only has that proven a lengthy process, but it has also raised questions regarding further increases in system complexity and limited backward compatibility with existing software. Consequently, completely rebuilding the system architecture from ground up offers a worthwhile alternative path, which was sometimes followed. But often one considered other tasks, cf. [6]. A more rigorous approach – building systems on small secure kernels – has been long acknowledged both in industry, security agencies, cf. [7], and of course in academia [8], [9], yet adopting them was limited by insufficient resources and lacking performance of considered systems. Over the recent years, the huge performance increase has largely rendered these concerns moot. Moreover, a very strong commercial interest in small kernels has just recently started. To name only a few publicly known efforts we refer the interested reader to [10]–[14].

Owing to their architectural elegance and small size, microkernels are often automatically embraced as "secure", yet this blind trust is often too optimistic. Indeed, in this paper, we will show that Fiasco.OC, a popular third-generation microkernel of the venerable L4-family, does not deliver on that very promise. This is particularly interesting as Fiasco.OC is used in multiple projects for its assumed ability to enforce strict isolation, among them secure laptops [14]–[16] and secure mobile devices [17], [18], some of them highly praised [19], [20]. We would like to emphasize that Fiasco.OC, unlike its closed-source competitors, encourages scrutiny by third parties as its source code is publicly available [21].

A. Attack Model

Before we proceed, we would like to formulate the assumptions regarding the underlying system architecture and the assumed capabilities of hypothetical adversaries. We envision a system (see Figure 1) with (at least) two compartments where the communication between them is understood to strictly



(a) Effective isolation (b) Ineffective isolation

Fig. 1: Attack scenario using covert channels

An effectively isolating microkernel can prevent data from being passed between compartments, even if both of them have already been compromised by adversary (case (a)). If the microkernel is ineffective in enforcing this isolation, data may be first passed between compartments and then leaked out to a third party in violation of a security policy prohibiting this (case (b)).

obey a security policy. In particular, the policy may state that no communication between any two compartments shall be possible. An organization may require that assets can only be sent through protected links and can only processed in compartments that have no access to unprotected networks. Under these provisions, the confidentiality of data should be preserved as long as the isolation between compartments is upheld.

As for the attacker, we consider highly determined adversaries with superior technical and organizational resources who have managed to place malware into all compartments. To achieve that, the adversaries may either directly attack Internet-facing compartments or draw on insider support in the targeted organizations to sneak in malware (Trojan horse). The attacker's goal is it to leak (potentially classified or even highly classified) data from the isolated compartment to a compartment with access to the Internet. From there, he eventually will be able to exfiltrate data to a place of his choice.

Regarding the system, we assume that the microkernel has no low-level implementation bugs and the system's configuration is sound, that is, there are no (authorized) direct communication channels between isolated compartments and the sum of all assigned kernel quotas is less than the amount of kernel memory available in the system.

B. Contributions

Following the classical research directions on covert channels, cf. [22], we identify so far unknown weaknesses in Fiasco.OC's kernel memory subsystem. These substantially undermine all efforts to isolate subsystems in security critical systems built on top of Fiasco.OC. In the rest of the paper, we develop real-world covert channels based on those weaknesses found in Fiasco.OC's kernel memory management. Following [23] we measured the capacity of the respective channels for meaningful system configurations to gain a better understanding of their applicability and severity.

C. Outline

The remainder of the present paper is structured as follows. After some required background informations in Section II, we will describe the identified weaknesses of Fiasco.OC's memory subsystem in Section III with an emphasis on educational clarity. This is followed by a description in Section IV as to how the described attack principle can be turned into working covert channels. A special real-world attack scenario involving a paravirtualized Linux will be presented in Section V before we present our experiments undertaken to test the viability of the approach in general and the capacity of the individual channels in particular in Section VI. The paper concludes with a related work section and some final observations.

II. Background

This section serves to recall some covert channel facts and also to give some educational background on Fiasco.OC. Especially, its subtle kernel memory management details are highly complex yet vital to understanding the findings presented in this paper.

A. Covert channel background

The issue of information leakage over covert channels came first to public attention when Lampson described the problem in 1973 [24]. A widely used description of covert channel is that used in [22]:

> A *covert channel* is a path of communication that was not designed to be used for communication.

Since then, covert channels have been a frequent topic of scientific research, cf. [5], [10], [22], [23], [25], [26].

In 1987, Millen [23] came up with a theoretical approach to estimate the capacity of covert channels. The US Department of Defence (DoD) acknowledged the threat and extended their Rainbow Series in 1993 with a classification scheme for covert channels [25].

Shared resources, such as caches [27], CPU [28], network subsystems [29], [30], or memory management [31], [32], are not only prevalent in all modern systems, but also run the risk of being the conduit for covert channel communication. Hardware-based covert channels exploit the fact that almost all processor resources are shared among processes in a very tightly coupled way. These shared resources can be used to construct covert communication. Osvik et al. showed that observing cache behaviour can be used to attack particular implementations of the AES encryption standard [33]. On the IA-64 architecture, the *spontaneous deferral* feature can be abused to facilitate information leakage [34]. However, not only shared processor resources are prone to information leakage, covert channels can also appear due to deficiencies in operating systems. This is particularly notable as advanced system architectures employ hypervisors or other small isolation kernels with the goal of enforcing strict security policies. The *memory deduplication* feature in modern VMs is exploitable [31], [32], such that it allows to create very high-bandwidth channels. Lalande et al. [35] prove that it is possible to form covert channels with various Linux subsystems, such as shared settings, system logs, thread enumeration, processor statistics,

processor frequency, only to name a few. They use these to leak private information in Android applications. Timing-based covert channels use varying timing events in a system to encode information [29]. Cabuk et al. [36] propose two statistical methods to detect timing variance in IP traffic to construct a covert channel.

B. Fiasco.OC Background

Fiasco.OC [21], a member of the venerable L4 family developed at the TU Dresden (Germany), has many attractive features. Besides its relatively small size – between 20 kSLOC and 35 kSLOC depending on the configuration – it supports the construction of secure systems by offering a security model based on capabilities [37]. Apart from its own merits, Fiasco.OC is distinguished from other microkernels by its accompanying user-level framework L4Re [38], which provides both libraries and system components aiding in the construction of a highly compartmentalized system.

Long before virtualization came out of its mainframe niche, L4Linux [39], a port of Linux onto Fiasco, which eventually evolved into Fiasco.OC, demonstrated that the encapsulation of a whole operating system was possible on commodity hardware without devastating performance. Since then, this useful ability has been extended [40] and complemented with support for hardware-assisted virtualization [41], [42].

Fiasco currently runs only on x86 and ARM-based platforms, however the case can be made that these two are the most relevant platforms nowadays. Its support for the rapidly evolving ARM ecosystem opens up countless opportunities. Unlike other microkernels with L4 heritage, such as OKL4 or seL4 [43], for which the source code is not available, Fiasco is distributed under the GNU General Public License (GPL) v2. Nevertheless, Fiasco.OC is used in a number of academic and commercial projects [14], many of which target security critical environments.

C. Memory Management

In order to gain a better understanding of the context, it is useful to revisit the relevant aspects of Fiasco.OC's design and implementation with respect to memory management in more detail. As with many other L4-like kernels, in Fiasco.OC, a noteworthy dichotomy between user-memory and kernel-memory management exists. Like the first L4 kernel, cf. [9], Fiasco.OC seeks restrict its functionality to mechanisms, upon which policies can be implemented by user-level servers depending on the specific needs of the applications at hand. L4 pioneered this principle in that it succeeded in moving the management of user-level memory completely out of the kernel. To that end, all L4-related kernel provide three mechanisms: first, page faults are exported to user-level, usually by having the kernel synthesize a message on behalf of the faulting thread. Second, a mechanism whereby the right to access page can be delegated (L4 terminolgy: *map*) between tasks and, finally, a mechanisms to revert that sharing (*unmap*). Since pages can be recursively mapped, the kernel needs to track this operation, otherwise the unmap might not completely revoke all derived mappings. The kernel data structure used for this purpose is usually called *mapdb*, an abbreviation for *mapping database*.

The situation is quite different for kernel memory, for which Fiasco.OC provides no direct management mechanism. When, in the course of a syscall, kernel memory is requested or released, the kernel turns to its internal allocators, each of which implements its own allocation strategy. As kernel memory is limited, a mechanism is required to prevent users from monopolizing this resource. Towards this goal, Fiasco.OC uses a quota mechanism. Each task is associated with a quota object, which represents the amount of kernel memory that is available for all activities in that task[1]. Whenever a user activity, e.g. a syscall, prompts the kernel to create a kernel object, the kernel first checks whether the active quota covers the requested amount of memory. If this is not the case, the syscall fails. For each initial (user-level) subsystem[2] the amount of available kernel memory is specified in the startup script. It is the sole responsibility of the system integrator to specify quota values whose sum is not larger than the amount of kernel memory available for the system at hand.

Intuitively, properly set quotas should ensure that each subsystem can use the share of kernel memory allocated for it regardless of the activity of other subsystems. However, due to fragmentation, it may happen that the allocators cannot find a contiguous memory range large enough to accommodate the requested object. We will show that the combination of Fiasco.OC's design and implementation gives an attacker to opportunity to bring about fragmentation on purpose, which, in turn, allows him to tie down kernel memory far beyond what his quota should allow, effectively rendering the quota mechanism useless. The implications are twofold: not only can an attacker deprive subsystems of resources, the tight resource conditions also allows him to establish a high-bandwidth covert channel with a conspirator who resides in a subsystem with which no communication is allowed per system policy.

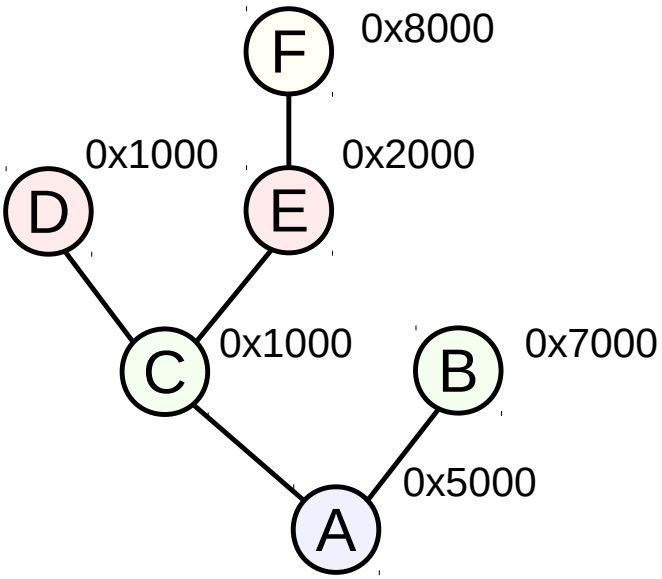
D. Fiasco.OC Implementation

1) Allocators: At the lowest level, kernel memory is managed by a buddy allocator. Since the size of kernel objects varies markedly, ranging from 32 bytes to 16 kB, and is not in all cases a power of two, allocating them directly from the buddy allocator would cause fragmentation over time. Thus, to limit the risk of fragmentation and boost locality of allocations, many objects are not directly allocated from the buddy allocator but are managed through slab allocators, which in turn are supplied with memory by the buddy allocator. Slab allocators, eighteen in total, accommodate only objects of the same type. On system startup, a certain amount of physical memory, ranging from 8% to 16% of the system's memory, depending on the amount memory in the system, is reserved for kernel use and fed into the buddy allocator.

Of the ten slab allocators for mapping trees (described below), eight are vulnerable to user-induced object placement. The two smallest are impervious to this attack as there is no way to shrink a mapping tree so small that it would be moved out into a allocator for smaller mapping trees. not size falls below the threshold below which they would be moved out of

[1] Quota objects can be shared among multiple tasks.

[2] A user-level subsystem, a group of collaborating tasks, is not to be confused with a kernel subsystem, such as the (kernel) memory subsystem of Fiasco.OC. As it is widely used in the community, we refrain from introducing a new term.



id	depth	address
A	1	0x5000
C	2	0x1000
D	3	0x1000
E	3	0x2000
F	4	0x8000
B	2	0x7000

Fig. 2: Mapping tree layout

the second smallest allocator, making the two smallest mapping tree allocators impervious to placement manipulations. That leaves the other eight mapping tree allocators as targets. Of the remaining slab allocators, only three are of interest: *ipc_gate, irq, factory*. The other allocators hold objects that have associated objects, for which an attacker cannot easily influence the place. For example, a task object has always an associated pagetable, which is allocated directly from the buddy allocator.

2) Hierarchical Address Spaces: Fiasco.OC's implementation of the mapdb seeks to minimize its size. Each physical page that is used as user memory is tracked through compact representation of a tree — the mapping tree — in a depth-first pre-order encoding. Each mapping can be represented by two machine words holding a pointer to its task, the virtual address of the mapping and the depth in the mapping tree[3]. Figure 2 illustrates the principle.

While this representation saves space compared to other implemntations using pointer-linked data structures, it brings about an object that potentially grows and shrinks considerably, depending on the number of mappings of that page.

If the number of mappings exceeds the size identifier of the mapping tree, the kernel allocates a bigger tree, into which it moves the existing mappings along with the new

[3]Since a page address is always aligned to the page size, the lower bits of the mapping address can be used for the depth in the mapping tree.

```
for each slab allocators do
  stride = sizeof(slab)/ sizeof(object)
  while not quota_exhausted do
    if (counter mod stride) == 0 do
      allocate permanent kernel object
    else
      allocate temporary kernel object
    fi
    increment counter
  done
  free all temporary kernel objects
done
```

Listing 1: Memory depletion

one. Conversely, when a thread revokes mappings, the kernel invalidates tree entries. Shrinking the tree, which involves moving it into a smaller data structure, takes place when the number of active entries is smaller than a quarter of the tree's capacity.

The combination of object whose size can be easily manipulated from user-level prompting the allocation of ever larger containers (mapping trees) and the known allocation strategy within slabs allows it to mount an attack on Fiasco.OC's kernel memory subsystem.

III. Unintended Channels

In this section we present so far undocumented issues with Fiasco's kernel memory management, which can be exploited to open up unintended communication channels. They were not anticipated by the designers and hence cannot be controlled by existing mechanisms. As a result, no security policy can be enforced on them.

A. Allocation Channels

As described in section II-C, an agent's kernel memory consumption is to be controlled by a quota mechanism. While the quota accounts for objects created on behalf of an agent, it does not capture the unused space in the slabs. It can be assumed that object creation is random enough that over time all slabs are evenly filled and that configuring a system with only half of its memory made available by quotas properly addresses the issue of fragmentation. Yet, a malicious agent is capable of causing a situation where this empty space accounts for more than 50% by deliberately choosing the order in which objects are created and destroyed. The algorithm to achieve that end is shown in Listing 1, with Figure 3 providing a graphical illustration of the process. To illustrate the point, both the agent's quota and the amount of used kernel quota are assumed to be zero. To accommodate the first object, the kernel allocates a new slab which is then used for subsequent allocations of that type (1). The process repeats (2). It is crucial which objects are released, the figure showing two possibilities. If the two remaining objects reside in the same slab, the second slab is not needed anymore and its memory can be reclaimed by the system allocator. In case the objects are allocated in different slabs, both slabs have to be kept. If repeated, an agent can cause the system to enter a state where it is filled with

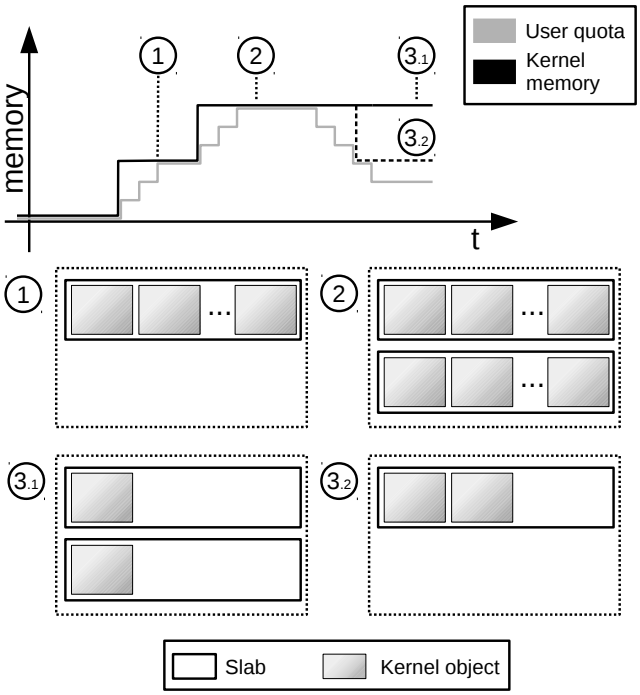


Fig. 3: Object placement in slabs

Depending on the order in which objects are created and destroyed, the number of slabs used to accommodate them can vary.

scores of slabs with only one object. While allocation requests for objects whose slab caches have nearly-empty slabs can be served easily, no new slabs can be allocated. An attacker would deliberately skip a frequently used object type during the fillup-attack, leaving the system with the few slabs for this type of object. This limited capacity is then used up later and no new slabs can be allocated.

It bears mentioning that the quota ballooning attack can not only be used for setting up communication channels but also for starving a system of any free memory in the buddy allocator. In addition to an initial sweep, a malicious agent would run periodically to gobble up memory that has been released by other parts of the system in the meantime. For systems where some components continuously allocate and release pagetables from the buddy allocator as to repopulate their address spaces – L4Linux is the archetypal example – ballooning the quota can be used to make these components seize up.

If objects could be shifted between slabs, a system could move objects such that only few slabs are fully populated, emptying others in the process. The memory of the newly empty slabs could then be reclaimed and returned to the underlying system allocator. Yet, Fiasco does not possess such an ability!

The amount of memory that can be tied down depends on four factors: the number of vulnerable slab caches, the number of objects held in their individual slabs, the order in which slabs are attacked, and, for objects with variable size such as mapping trees, on the minimal size required for the object to stay in a certain slab. In our experiments, we could tie down six times the amount of the assigned quota. In a system with two subsystem where the available kernel memory is equally divided, a factor of two is enough to starve the system completely.

B. Mapping-tree Channels

In addition to the allocation channels described in the previous section, there is another implementation artifact that can be misused as a communication channel. As described in section II-D, Fiasco tracks user-level pages in tree-like data structures, so called mapping trees. Available in various sizes, the mapping tree data structure grows and shrinks as the page it tracks is mapped into and unmapped from address spaces. During an unmap, the kernel uses it to find the part of the derivation tree that is below the unmapped page, if any, and unmaps it as well. Although the mapping tree structure is dynamically resizable, Fiasco imposes an upper bound, thus limiting the number of mappings that can exist of a physical page in the system. At first glance, this might not seem to be an issue because isolated subsystems are not meant to share pages to begin with. However, L4Re, the user-level OS framework running on top of Fiasco.OC, provides a number of services, among them a runtime loader, which is not unlike the loader for binaries linked against dynamic libraries on Linux (ld.so). Experimentally, we found that 18 pages are shared this way among all subsystems that are started through the regular L4Re startup procedure.

IV. Channel Construction

In this section, we explain how to construct a high bandwidth covert channel. We also devise more sophisticated channels to work around problems impeding stability and transmission rates, which allow us to establish a reliable and fast covert channel even in difficult conditions. Three of the proposed channels rely on the fact that a malicious thread is able to use up more kernel memory than its quota allows. The remaining channel exploits a limit in a shared data structure. All of them share a similar principle for sending and receiving data, as shown in Listing 2 and Listing 3. Data bits are expressed as x_y, x being the bit value and y being the base.

A. Page Table Channel

The Page Table Channel (PTC) requires an initial preparation as described in Section III-A. Following this, the channel can be modulated in two ways: for sending a 1_2, the sender allocates page tables from the kernel allocator, for transferring a 0_2, it waits for one interval. To facilitate page table allocation, a helper task is created by the sender and pages are mapped into its address space. These pages are placed in such a way that each page requires the allocation of a new page table. The receiver can detect the amount of free memory in the kernel allocator by performing the same steps as the sender. The number of page tables available to the receiver is inversely proportional to the number of page tables held by the sender. This knowledge can be used to distinguish between the transmission of a 1_2 and a 0_2. At the end of every interval, the sender has to release the page tables it holds. Unmapping the pages from the helper task is not sufficient because Fiasco.OC does not release page tables during the lifetime of a task.

Instead, the helper task has to be destroyed and recreated. The implications of this will be discussed in detail in our evaluation in Section VI-A1.

B. Slab Channel

The slab channel (SC) uses contention for object slots in slabs to transfer data. For this purpose, we set up the channel as described in section III and subsequently perform channel specific preparations. One slab is selected to hold mapping trees for the transmission. This slab is filled with mapping trees until only one empty slot remains, which is used for the actual transmission. Figure 4 shows the principle. To transfer a 1_2, the sender needs to fill this last slot. Assuming a slab of size 4KB for the transmission, it causes a mapping tree residing in a slab of 2KB to grow, until the it exceeds its maximum size and the kernel moves it into a bigger slab – the one intended for transmission. The receiver can determine which bit was sent by performing the same operation as the sender. If this fails, the receiver interprets this as a 1_2 and as 0_2 otherwise.

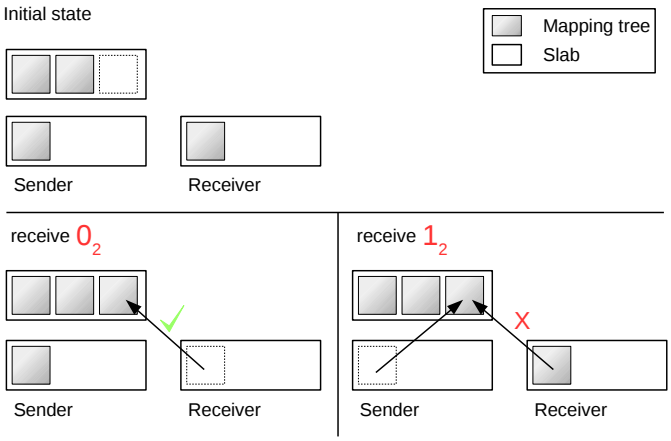


Fig. 4: Transmission in the slab channel

The sender fills or leaves empty the last spot in a slab; the receiver reads the value be trying to move an object into that spot and checking the return code indicating the success or failure of the operation.

```
for each bit do
  while not clock_tick mod x do
    sleep
  done
  if bit == 1 then
    channel_specific_map
    sleep
    channel_specific_unmap
  else
    sleep_1_tick
  fi
done
```

Listing 2: Clock-synchronized sender

```
while receiving do
  while clock_tick mod x do
    sleep
  done
  if channel_specific_map then
    add_0_to_bits
  else
    add_1_to_bits
  fi
  channel_specific_unmap
done
```

Listing 3: Clock-synchronized receiver

C. Mapping Tree Channel

The Mapping Tree Channel (MTC) exploits the fact that applications started by L4Re share 18 read-only pages. A size constraint imposed by the maximal size of the mapping tree – the data structure whereby an individual page is tracked – limits the number of times a page can be mapped to 2047. When this maximum is reached, any attempt to create new mappings of the page will fail. In the most basic form, the communication partners agree on one shared page, which they then use as a conduit. In the preparatory phase, the mapping tree of the chosen page will be filled such that only room for a single entry remains. Creating these mappings is possible because the shared pages are regular and are not subject to mapping restrictions.[4] To transfer a 1_2, the sender creates a mapping of the chosen shared page, maxing out its mapping count. The receiver, concurrently trying to create a mapping, fails as result. Conversely, a 0_2 is indicated by no mapping created by the sender so that the receiver's operation succeeds. Compared to the other two channels, the MPC incurs low overhead. Unlike the SC, an MPC transmission requires at most three operations. In contrast to the PTC, no costly task destruction is necessary.

D. Channel Optimizations

Since the step rate of clock-synchronized channels is limited by Fiasco.OC's 1000Hz timer resolution, the only path to higher bandwidths is to increase the number of bits transmitted per step. To that end, we devised two methods that allow for increased channel bandwidth at the cost of higher CPU utilization.

1) Multi-channel transmission: The bandwidth can be trivially boosted if the number of sub channels is increased. The underlying assumption here is that several of them are available. Fiasco.OC maintains multiple slabs, each of which can constitute a channel. An attacker can now choose between using a slab for tying down kernel memory or use it as a communication channel. Likewise, the existence of multiple shared pages can be exploited.

2) Multi-bit transmission: A different approach is to transmit multiple bits per channel and step. For that, a channel needs

[4]There are regions in Fiasco.OC tasks, such as the `user-level TCB` (UTCB) that, while being accessible, cannot be mapped.

to be capable of holding 2^n states, n being the number of bits [5]. Channel levels can be realized by occupancy levels, assuming the underlying channel resource can be allocated in increments. The number of operations to bring up and sense these levels grows exponentially with the number of bits. For that reason, multi-channel schemes, where the number of operations grows linearly with the number of bits, might be preferable. However, the number of available sub channels is limited.

E. Transmission Modes

1) Clock-synchronized transmission: Under clock synchronization, two agents make use of a shared clock to synchronize their execution. Briefly, the sender and receiver share a notion of points in time where certain actions have to be completed. It is the responsibility of either party to make sure that its activity (writing to the channel or reading from it) is completed before the next point is reached as there are no additional synchronization mechanisms whereby the other party could detect that its peer's activity was not finished.

Fiasco provides a sleep mechanism with a 1ms wake-up granularity. Moreover, every task can map a special page, the *kernel info page* (KIP), through which Fiasco exposes the current system time, again with a 1ms granularity. This global clock is very helpful for scenarios with long synchronization periods that consist of multiple 1ms ticks.

On x86 platforms the *time stamp counter* (TSC), a high-precision clock, is available by default, which can be used to detect points in time *within* a time slice. If the sender and receiver run on different processors, this mechanism can be used to define synchronization points that are not aligned with the 1ms time slices.

2) Self-synchronizing transmission: On systems under heavy load, there is no guarantee that conspiring threads are executed frequently enough as they compete with other threads for execution time. Whenever an interval is missed by either party, bits are lost and the transmission becomes desynchronized. To alleviate this problem, we can either incorporate error correction into the channel – at the cost of the bit-rate – or we can design our channel to be independent of a common clock source.

Under the self-synchronizing regime, sender and receiver do not observe a shared clock. Instead, they dedicate some of the available data channels to synchronization, effectively turning them into spinlocks. Using this mechanism, we can ensure that sender and receiver can indicate to each other whether the other party is ready to write to or from the channel. This process is illustrated in Figure 5.

One drawback of self-synchronization is that at least two data channels have to be set aside for lock operations, reducing the number of the channels for data transmission. Especially in setups where data channels are rare, this can be a serious issue. We take a look at the achievable bit rates in Section VI-B.

V. L4Linux Scenarios

L4Linux [39] is a port of the Linux kernel onto L4 kernels, with the latest version being widely used in combination with

[5]The levels can be spread over multiple channels, so as to be more flexible regarding the levels required per channel.

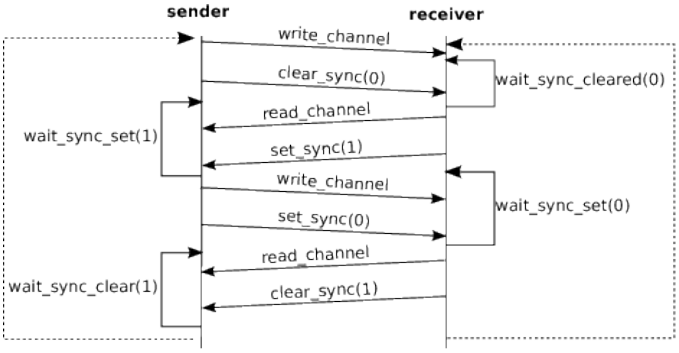


Fig. 5: Self-synchronizing transmission

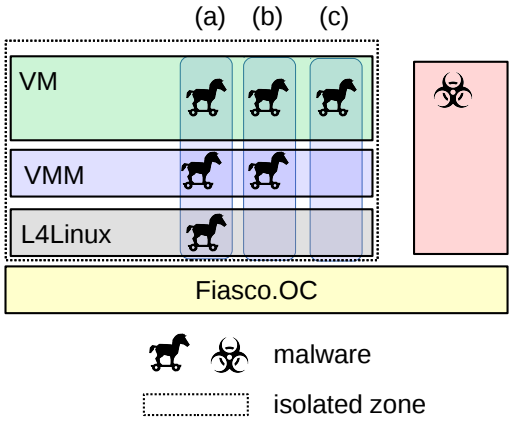


Fig. 6: Threats in L4Linux-based scenarios

Configurations of a virtualization stack with malware present in the system where no data can be leaked. The presence of covert channels ((b),(c)) require that more components be trustworthy so as to prevent information leaking out of the isolated zone.

Fiasco.OC. L4Linux does not only allow running unmodified Linux applications, it even supports the reuse of virtualization solutions like KVM [41]. The combination of mature Linux virtualization stacks, the strong isolation afforded by microkernels, which also reliably encapsulates VMs, and the excellent performance of hardware-supported VMs have led to the use of this architecture in production systems [14]. The requirement of direct access to the Fiasco.OC kernel API required a Linux-based agent to first gain control of the L4Linux kernel [6]. We will now show that even an application lacking this ability can use the covert channel, albeit with lower bandwidth. This has profound implications on the security situation as shown in Figure 6, as it requires more components to be trustworthy. A component is trustworthy if a desired behavior – no information leakage in our case – is only achieved if the component works as intended. Without covert channels, only the microkernel would have to be trustworthy (Figure 6.a)[7]. In systems with covert channels that are not accessible to Linux applications information cannot be leaked as long as L4Linux works as expected (Figure 6.b). If Linux applications, in the case at hand the user-level VMM, have access to the

[6]Only the task for the L4Linux kernel expose the Fiasco.API, the tasks used for running Linux applications do not.

[7]under the assumption of a sound system configuration, see Section I-A

covert channels, then they also need to be trustworthy (Figure 6.c). The changes from 6.a over 6.b to 6.c increase the attack surface exposed to a VM, which is in all cases assumed to be untrusted.

We have designed both a Linux-based sender and receiver. In either case, we use a clock-synchronized (Section IV-E1) page table channel (Section IV-A). The following sections describe the general principle of the sender and receiver, while we evaluate our implementation in Section VI-D.

A. L4Linux sender

The initial preparation is performed by the agent on the L4 side, as explained in Section III-A. Similarly to the regular PTC sender, the L4Linux sender creates a helper task by forking its own process, into which it maps memory in such a way that every mapping requires the allocation of a new page table. Since Linux processes are implemented by L4 tasks, a page table request in Linux results also in a page table allocation in Fiasco.OC. Thereby, the sender can control the number of kernel pages tied up in Fiasco.OC by the number of page tables it allocates in L4Linux. The receiver, an agent with access to the Fiasco.OC API, is identical to the one described in Section IV-A.

B. L4Linux receiver

As in the previous section, the channel setup is left to the L4 agent. The L4Linux receiver process is more complex because L4Linux hides Fiasco.OC syscalls that failed from Linux applications. Thus, the Linux-based agent has to deduce the system state, constructed by the sender, through variances in the timing behaviour of suitably chosen operations. Specifically, the receiver forks two Linux processes that run sequentially, synchronized over sockets by the main process. Both of them allocate page tables by the method described in the previous section. The first process is still in existence but halted when the second process runs. The allocations of the first process never fail, the purpose of its existence is a suitable target for resource reclamation on the part of L4Linux.

If the sender, which is identical to the one described in Section IV-A, did not deplete the kernel allocator, enough resources are available in Fiasco.OC to supply all allocations of the second process. In the other case, there is not enough memory for the second Linux process to finish, triggering an L4Linux resource reclamation cycle wherein the first process is destroyed.[8] As a result, the time it takes for the second process to complete its operations either encompasses the time for a task destruction (1_2) or not (0_2). Conveniently, owing to the long RCU cycle involved in a task destruction, the difference is large enough to be reliably measured. Figure 7 illustrates this principle in detail.

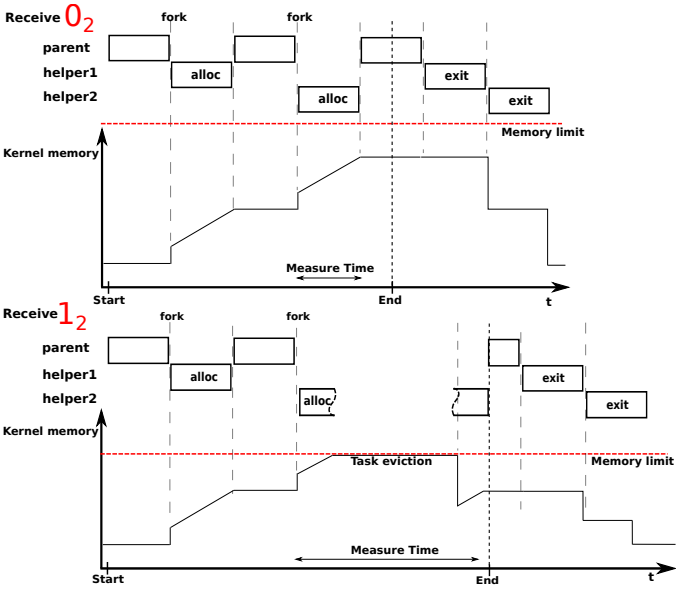


Fig. 7: Operation of L4Linux-borne agents

In case an allocation can only satisfied if the resources of task are reclaimed, then the time needed for the task destruction are visible to the allocating agent.

VI. Evaluation

To evaluate the feasibility of our approach and to measure the achievable bandwidth of the various channel configurations, we ran a number of a experiments on three different hardware platforms:

- PandaBoard [44], ARM Cortex, A9 1.0Ghz, 2 cores.
- MinnowBoard [45], Intel Atom E640, 1.0Ghz, 1 core, 2 hyperthreads.
- Desktop-grade PC, AMD Athlon X2, 2.4Ghz, 2 cores.

All three platforms are equipped with 1GB of RAM. The PandaBoard comes with an OMAP4430 SoC (System-on-a-Chip), which features two ARM Cortex A9 cores, which are widely used in smartphones and tablets. The Atom is pitched by Intel at the mobile market, among others. Our rationale to include a third, more powerful platform was based to have a substitute for future, faster mobile processors.

With the exception of one setup, we used Fiasco.OC/L4Re [21], [38] version r54 (both) and L4Linux [46] version r39. For the experiment involving L4Linux on ARM, we used Fiasco.OC/L4Re version r63 and L4Linux version r46.

In our measurement setups, we used two agents (sender and receiver) implemented as native L4Re applications, with the system setup such that they did not have direct communication channels between them. In the setups involving Linux, one agent was implemented as a Linux application. Unless stated otherwise, the sender transmits packets of four bytes. The first byte holds a continuously incremented counter value followed by three bytes with a CRC checksum. The receiver feeds the received bits into a queue and checks whether the latest 32 bits contain a valid CRC checksum. All transmission rates reported in this section reflect the number of bits transmitted per time interval, including the checksum bits. The "*channel states*" row in our tables lists the number of states n a channel needs to support in order to transmit $\log_2 n$ bits per interval. We want to point out that using a CRC checksum across all measurements was only used to be able to detect transmission errors, it is no indication of whether or not a channel is potentially

[8] Only the L4 task belonging to the Linux process is destroyed, not the Linux process itself. When L4Linux schedules it again, a new L4 task will be created for it.

lossy. Specifically, clock-synchronized channels are vulnerable to transmission errors depending on the system's load situation whereas self-synchronized channels transmit all data error-free regardless of other activities.

A. Clock-synchronized transmission

In this section, we evaluate all transmission methods that belong to the category of clock-synchronized channels under various aspects. We start with the basic capacities we observed, investigate the effects of using multiple processors and analyze the effects of individual improvements, such as transmission with multiple channels or multiple bits at a time. Finally, we take a look at the CPU utilization of our transmissions and what conclusions we can draw from these numbers in regards to channel speed, reliability, stealth and scalability.

1) Base Channel Capacities: Our first setup implements the PTC, as introduced in Section IV-A as our most basic transmission method. We soon noticed during our experiments that enabling SMP support results in lower bit rates when compared to uni-processor setups. This counter-intuitive observation can be explained by taking a closer look at the time required to perform the individual operations for sending and receiving, as implemented by the PTC. On SMP enabled setups, the destruction of the helper task takes significantly longer, for which the reason can be found in Fiasco.OC internals. It employs a *read-copy-update* (RCU) [47], a synchronization method incurring low overhead while also being simple. The downside of the approach are additional latencies for individual operations because an operation can only terminate after a grace period has elapsed. In Fiasco.OC, this grace period is 3 ms long. On MP systems, this affects clock-synchronized transmission methods that involve frequent object destructions. The PTC in particular suffers from this circumstance and achieves on uniprocessor system more than twice the bit compared to SMP systems, as shown in Table I. With a duration of two ticks for a read and write operation, SMP systems require at least five to six ticks for reliable transmission due to the delay, with the AMD system never needing more than five. As indicated by the slightly lower bit rate for the MinnowBoard, the object destruction on this platform occasionally requires an additional tick. In contrast, both platforms yield the same bit rate on uniprocessor setups, transmitting at a constant 500 bits/s rate. The SC (Section IV-B) operates in principle similar to the PTC but does not require a potentially costly object destruction. The effects of this improvement are reflected by the measurement results in Table II. The transmission rate is solely limited by the amount of data that can be written to and read from the channel within a transmission interval, which depends on the time required to create and delete mappings.

The last column shows our final optimization of this transmission method, which increases the frequency by replacing the KIP clock with the high resolution TSC for synchronization purposes. This allows us to increase the frequency and fit the write and read operations for one bit into 1000 ms, yielding 1000 bits/s on the desktop system and on the MinnowBoard.

2) Multi-channel/multi-bit: The amount of data that can be transmitted per time interval determines the maximum bit rate, a major limiting factor against which we introduced multi-bit transmission in Section IV-D2. This enables us to transmit multiple bits in every transmission interval, increasing the PTC bit rate for an unmodified frequency on all platforms, both in SMP and in UP setups. The exact numbers are presented in Table I, reflecting that on a more powerful processor more bits can be encoded per interval. The delaying effect of the object destruction on SMP setups remains visible in the result.
Another channel type, the MTC (Section IV-C), employs multiple sub channels. This optimization along with its distinguished reliance on a different Fiasco.OC mechanism for transmission makes it the fastest of our clocked channels. Table III shows a maximum bit rate of 4000 bits/s on the x86 platforms and 2000 bits/s on ARM. While both the MinnowBoard and the AMD system manage a reliable transmission every 2 ticks, we had to lower the frequency to every 4th tick on the PandaBoard. During our investigation of this, we discovered that the time required to send and receive a specific bit pattern is affected by what was sent before. The next section covers this effect in more detail.

TABLE I: PTC results

Platform		UP		MP	
AMD	channel states(#)	2	32	2	16
	period (clock ticks)	2	3	5	5
	capacity (bits/s)	500	1666	200	800
MinnowBoard	channel states(#)	2	4	2	16
	period (clock ticks)	2	2	6	6
	capacity (bits/s)	500	1000	167	667

TABLE II: SC results

Platform		UP	TSC + MP
AMD	channel states(#)	2	2
	period (clock ticks)	2	1
	capacity (bits/s)	500	1000
MinnowBoard	channel states(#)	2	2
	period (clock ticks)	2	1
	capacity (bits/s)	500	1000
PandaBoard	channel states(#)	2	
	period (clock ticks)	2	
	capacity (bits/s)	500	

TABLE III: MTC results

Platform		2 channels	8 channels
AMD	channel states(#)	2	2
	period (clock ticks)	2	2
	capacity (bits/s)	1000	4000
MinnowBoard	channel states(#)	2	2
	period (clock ticks)	2	2
	capacity (bits/s)	1000	4000
PandaBoard	channel states(#)	2	2
	period (clock ticks)	2	4
	capacity (bits/s)	1000	2000

3) CPU utilization: In clocked scenarios, the choice of an optimal transmission frequency is very vital. What is considered optimal heavily depends on the level of importance attributed to the properties speed, reliance and stealth. As our previous measurements showed, to prioritize either property, it is important to take factors limiting the number of bits that

can be transmitted per interval into account. Apart from delays imposed by the design, two major influencing factors are clock frequency and utilization of the system processor. These determine how fast mappings can be created and destroyed. In order to improve reliance, we need to always assume the worst case when estimating the number of bits that can be transmitted safely per interval. To prioritize stealth, it is critical to know the percentage of CPU utilization our transmission will incur. Last but not least, to achieve high capacities, we need to know the optimal ratio between bit rate and transmission frequency.

During our MTC (Section IV-C) experiments, we discovered that when comparing mapping and unmapping operations, the latter are by far the more expensive ones in terms of CPU utilization. We investigated this in detail by counting the CPU cycles spent while performing the individual steps of the MTC 2-channel scenario for sending and receiving bit patterns, as well as resetting the channel afterwards. Sending and receiving both require mapping operations whereas the reset step involves an unmap operation on the receiver side. Whenever a bit sent over a channel differs from its predecessor, the channel state needs to be adapted by the sender. This is included in the time recorded for the send operation. We limited our measurements to the x86 platforms, which lent themselves to this effort due to the presence of the TSC on x86.

With unmapping being more expensive than mapping, sending a bit pattern of 00_2 following a pattern of 11_2 is the most expensive pattern sequence since it requires both (sub-)channels to be reset by the sender. On the receiver side, the situation is slightly different. Since it resets the mappings created for sampling every interval, the number of unmap operations that must be performed is proportional to the number of mappings created during sampling. This is affected by the bits being sent as every 1_2 blocks the receiver from creating mappings on the respective channel. Hence, the most expensive pattern to receive is 00_2 as it allows the receiver to create the maximum number of mappings on both (sub-)channels. Table IV compares the percentage of CPU cycles spent on sending and receiving the worst case pattern or pattern sequence. We can use these values to tune our transmission parameters towards the three properties mentioned at the start of this section. It is apparent that there should be some room for additional bits in order to optimize our transmission for speed, as we stay well below 25% CPU utilization on both platforms. On AMD, our experiments use only a tiny fraction of the available CPU cycles, confirming that our configuration of the MTC establishes a particularly stealthy and reliable communication channel. Last but not least, it is obvious from the numbers that the MTC scales very well with processor performance as was expected.

TABLE IV: MTC worst case CPU utilization

Platform	**CPU Utilization**	
	Send	Receive
AMD	3.13%	5.65%
MinnowBoard	12.25%	17.54%

B. Self-synchronized transmission

We will now take a look at how well the self-synchronizing transmission method described in Section IV-E2 performs. All tests used maptree channels for both synchronization and data transfer.

The results (Table V) show that for all three systems, the channel capacity grows with the number of channels. The apparently sub-linear scaling can be accounted for if, instead of the channel capacity, the number of operations (including synchronization) is taken into account. Transmitting a bit requires two or three operations, depending on the value to be transmitted. The synchronization overhead for two transmission steps is five operations. So, going from one to two data channels increases the number of Kashyap Thimmarajuoperations between 44% (6.5/4.5) and 54% (8.5/5.5). The measured increased data capacities (54% (Pandaboard, 5408/3511), 56% (Minnowboard, 3840/2448), 48% (AMD, 14738/9957) are commensurate.

In contrast, the channel capacities scale much poorer with the number of (logical) cores used. We attribute this to resource conflicts in the memory subsystem (cache, memory bandwidth) as each operation has to scan through 16kB mapping tree. In the case of the Minnowboard, we do not see any scaling at all, which was to be expected as the used Atom E640 processor only features a single core with hyperthreading.

TABLE V: Throughput depending on the number of transmission channels and number of cores used

Nr. of channels (#)	**Throughput (bits/s)**		
	Panda-board	Minnow-board	AMD
1	3511	2448	9957
1+1	5605	2414	11940
2	5408	3840	14738
2+2	8782	3815	18592
4	7449	5368	19974
4+4	12166	5420	25492
8	9207	6697	24902
8+4	13569		
8+8		6881	30582
16	10457	7637	28416

Self-synchronized transmission with maptree data channels. A single number in the *# of channels* column indicates that a single pair (sender, receiver) was run on the same core/logical core with the given number of data channels. A number pair is used for two sender-receiver pairs running on different cores/logical cores. Since there are not enough data channels for *8+8* combination on the Pandaboard, we used a *8+4* setup as widest configuration there.

C. Impact of system load

Our previous experiments were run on systems without any additional activities. As real-world systems are not idle all the time, the question arises as to how load on the system impacts the throughput of the covert channels. To answer this question, we designed an experiment where we could dial a load and measure the channel throughput under the given circumstances. We used Fiasco with the fixed-priority scheduler, so that load could be easily generated by a highly-prioritized thread that

alternated between legs of busy looping and sleeping. We verified the correctness of this behavior by reading this thread's execution time, a figure provided by Fiasco.

As the results in Table VI show, the achievable throughput is directly proportional to the CPU time available to the conspiring agents. In keeping with the expectation for self-synchronized transfers, all data arrived ungarbled.

TABLE VI: Throughput under load

System load (%)	**Throughput** (bits/s)
95.2	213
90	415
84	757
67	1516
50	2274
34	3078
25	3493
20	3720
16.7	3900
10	4252
5	4450
0	4731

Self-synchronized transmission with four maptree data channels on the Pandaboard. Only one CPU is active.

D. L4Linux results

The L4Linux setup is based on a clock synchronized method, hence we need to synchronize to a common clock source. Since the frequency of the L4 KIP clock is well-known, it is possible to synchronize on a predefined, constant pattern sent by either side of the channel and calculating the clock offset based on that. To simplify our experiments, we made the KIP clock accessible to the L4Linux user space instead. The delay caused by task destructions is inherently longer on SMP systems, as explained in Section VI-A1, which is particularly beneficial to the L4Linux receiver. It measures this delay as part of its channel sensing routine. For both L4Linux sender and receiver, we implemented a regular user space program which performed the steps described in Section V. As our main objective was to prove the feasibility of a covert channel in L4Linux, we chose comparatively conservative parameters, as Table VII shows.

TABLE VII: L4Linux results

Platform		**Receive**	**Send**
AMD	channel states(#)	2	2
	period (clock ticks)	20	10
	capacity (bits/s)	50	100
MinnowBoard	channel states(#)	2	2
	period (clock ticks)	30	17
	capacity (bits/s)	33	58
PandaBoard	channel states(#)	2	2
	period (clock ticks)	30	30
	capacity (bits/s)	33	33

VII. Related Work

Ristenpart et al. [48] showed that constructing a cross-VM side-channel is possible in Amazon's EC2, prompting further research on virtual machine infrastructure. Wu et al. [49] use the EC2 as a host for covert communication. Exploiting peculiarities of the underlying x86 platform, they were able to achieve transmission rates of up to 100 bits/s. Xu et al. [27] also leverage cache crosstalk to construct a covert channel with channel capacities of up to 262 bits/s. Suzaki et al. [32] examined various virtual machines and discovered a side-channel in KVM based on the *memory deduplication* feature, which merges same-content memory pages, allowing for increased memory utilization. Based on these findings, Xiao et al. [31] constructed a covert channel with bandwidths of up to 1000 bits/s yet with a large memory footprint of $\sim$ 400MB. Lipinski et al. [50] drew on work done by Ristenpart et al. [48] and improved the method of hard disk contention achieving a 1000 times higher steganographic bandwidth compared to the 0.1 bits/s by Ristenpart et al. Okamura [28] et al. examined the load-based covert channel on the Xen hypervisor. Changes in execution time due to sharing of a physical CPU led to an information leakage. The constructed covert channel had bandwidth of $\sim$0.5 bits/s. Lin et al. [51] propose two covert communication channels based on the last PID and a temporary file in the Linux OS. They construct three channel mechanisms to counter mitigation techniques. The authors achieve bandwidths of up to 40 bits/s. In comparison, we achieved significantly higher bandwidths of up to 5000 bits/s, on a system that emphasizes isolation.

Murray et al. [52] showed that formal correctness proofs of a general purpose kernel (seL4) can be extended to cover information flow as well. The authors list a number of restriction, among them: support only for a single processor, no device interrupts, and a global static schedule. Although these assumptions might be common for deployed separation kernels (as the authors claim), they impede system construction compared to Fiasco. As such, the authors investigate a more specialized system.

VIII. Conclusions

It was shown that Fiasco.OC, a widely known and used third-generation microkernel, cannot deliver on isolation expectation in the face of a determined attacker. We have identified three shared facilities whose control mechanisms can be rendered ineffective (kernel allocator, object slabs) or, worse, who lack them at all (mapping trees). In our experiments, we showed the feasibility of using them as a means of communication and achieved maximal channel capacities of up to $\sim$30000 bits/s, outstripping previously found channels in VM environments by a fair margin. Our ongoing experiments indicate that the achieved channel capacities are only a provisional result and a more than twofold increase is within reach with further refinements. Moreover, the trend to faster processors with more cores suggests that the capacity of the described channel types will only grow in the future.

The identified channels raise concerns about Fiasco.OC's role in devices with security requirements. For example, the covert channel rates achieved on the Pandaboard (up to $\sim$ 13500 bits/s) lend plausibility to some disconcerting attack

scenarious against phones that offer encrypted VoIP calls. Specifically, a Trojan horse may eavesdrop on the raw audio data of an encrypted VoIP connection as they are processed unencrypted in a protected compartment[9]. The bandwidth of the covert channels needed to exfiltrate the call data into an unprotected compartment is sufficient for voice calls if compressed with modern algorithms, which need ($\sim$ 4000 bits/s).

Although an exhaustive list of design changes suitable to counter our attacks is beyond the scope of the paper, we would like to touch on the topic. As a communication step requires both conspirators to execute, it seems advisable to introduce a mechanism along the lines of work done by Wu et al. [53], whereby the switch rate between isolated domains can be controlled. We acknowledge that such a scheme may have a negative impact on throughput in scenarios where the activity pattern of subsystem is bursty. In the light of Linux-based systems, there cannot be any doubt about the need for the ability to encapsulate Linux instances. However, endowing Linux with the full microkernel API seems to be ill-advised as the offered feature set is not fully needed, yet grows the attack surface of the microkernel.

The inadequacy of Fiasco.OC's memory subsystem reinvigorates the debate about whether system design shall be driven by pragmatism or principle. Proponents of Fiasco's pragmatism often point to the wide range of functionality it provides. Indeed, the support of multi-processors, the ability to host Linux [46] on platforms without virtualization support, and the availability of a user-level framework [38] make for a system that lends itself to a wide range of applications. In contrast, the emergence of seL4 [43], [54], [55], the first general purpose kernel for which a formal correctness proof could be produced, brings within reach the prospect that systems can be constructed on error-free kernels. That said, it should be kept in mind that as of yet the seL4 ecosystem is in certain important aspects rather limited. For example, although multiprocessor support has been considered [56], a multiprocessor version of seL4 is not available. Moreover, the discussed clustered multi-kernel (CMK) model raises questions as to the implications on the user-level programming model. As case in point, CMK was considered for Fiasco as well, but then dismissed over concerns that the programming model would diverge to much from the prevelant processor agnostic models as found on Linux and Windows. In a similar vein, the virtual machine monitor shipping with seL4 [57] does not support the ARM architecture, which renders it unsuitable for mobile devices. In any event, it will be interesting to examine seL4 and watch its ecosystem evolve.

Small kernels have the tendency to inspire users confidence. Yet, as demonstrated this trust might be misguided. Instead, we encourage designers of security-conscious systems to scrutinize the merits of the OS kernels they use on reasonable grounds, not promises or perceptions.

IX. Acknowledgement

This research was supported by the Helmholtz Research School on Security Technologies.

[9]Unlike regular voice calls, VoIP calls are not handled by the baseband processor but by the application processor

References

[1] R. Clarke and R. Knake, *Cyber War: The Next Threat to National Security and What to Do About It*. Ecco, 2012.

[2] "Apt1: Exposing one of china's cyber espionage units," 2013. [Online]. Available: http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf

[3] "Apt28: A window into russia's cyber espionage operations?" 2014. [Online]. Available: http://www.fireeye.com/resources/pdfs/apt28.pdf

[4] (2014, November) Regin: Top-tier espionage tool enables stealthy surveillance. http://www.symantec.com/connect/blogs/regin-top-tier-espionage-tool-enables-stealthy-surveillance.

[5] T. Jaeger, *Operating System Security*, ser. Synthesis Lectures on Information Security, Privacy, and Trust. Morgan & Claypool Publishers, 2008.

[6] C. E. Irvine, T. D. Nguyen, D. J. Shifflett, T. E. Levin, J. Khosalim, C. Prince, P. C. Clark, and M. Gondree, "Mysea: The monterey security architecture," in *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*, ser. STC '09. New York, NY, USA: ACM, 2009, pp. 39–48. [Online]. Available: http://doi.acm.org/10.1145/1655108.1655115

[7] P. Karger, M. Zurko, D. Bonin, A. Mason, and C. Kahn, "A vmm security kernel for the vax architecture," in *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*, May 1990, pp. 2–19.

[8] J. Liedtke, "Improving ipc by kernel design," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 175–188, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/173668.168633

[9] J. Liedtke, "On micro-kernel construction," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 5, pp. 237–250, Dec. 1995. [Online]. Available: http://doi.acm.org/10.1145/224057.224075

[10] D. Kleidermacher and M. Kleidermacher, *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*. Elsevier Science, 2012. [Online]. Available: https://encrypted.google.com/books?id=75kh6iXNNZAC

[11] "Samsung and green hills mobile announce samsung knox hypervisor - powered by integrity," http://www.ghs.com/news/20140227_rsa_samsung_knox.html. [Online]. Available: http://www.ghs.com/news/20140227_rsa_samsung_knox.html

[12] "Green hills software signs teaming agreement with HP to offer secure android smartphones and tablets to uk public sector," http://www.ghs.com/news/20131218_hp_teaming_integrity_multivisor.html.

[13] "Viasat and green hills software team to deliver secure android smartphone," http://www.ghs.com/news/20131030_ARMTechCon_ViaSat.html.

[14] "Genua cyber-top," https://www.genua.de/loesungen/security-laptop-cyber-top.html.

[15] "genua presents cyber-top security laptop l4re inside," http://www.kernkonzept.com.

[16] "L4re - isolation und schutz in mikrokernbasierten systemen," https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Veranstaltungen/ITSiKongress/2013/Michael_Hohmuth_14052013.pdf?__blob=publicationFile.

[17] (2014) Simko - sichere mobile kommunikation. http://security.t-systems.de/loesungen/mobile-simko.

[18] (2013, September) L4re-based simko 3 security smartphone wins bsi approval for classified data. http://www.kernkonzept.com.

[19] (2014) http://www.horst-goertz.de/warum-ein-preis-fuer-deutsche-it-sicherheit/was-wird-ausgezeichnet/preistraeger-2014/preistraeger-2014-3-preis-mikrokernel. http://www.horst-goertz.de/warum-ein-preis-fuer-deutsche-it-sicherheit/was-wird-ausgezeichnet/nominierte-2014.

[20] (2013, September) Telekom's high-security cell phone obtains federal office for information security bsi approval. ww.telekom.com/media/enterprise-solutions/200664.

[21] (2014, May) Fiasco.oc website. http://os.inf.tu-dresden.de/fiasco/.

[22] M. A. Bishop, *Computer Security. Art and Science*. Addison-Wesley Professional, 2002.

[23] J. K. Millen, "Covert channel capacity." in *IEEE Symposium on Security and Privacy*, 1987.

[24] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, pp. 613–615, Oct. 1973. [Online]. Available: http://doi.acm.org/10.1145/362375.362389

[25] *A Guide to Understanding Covert Channel Analysis of Trusted Systems (Light Pink Book)*, Computer Security Center (CSC), Department of Defense (DoD), Nov. 1993.

[26] Z. Wang and R. B. Lee, "Capacity estimation of non-synchronous covert channels," *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, vol. 2, pp. 170–176, 2005.

[27] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of l2 cache covert channels in virtualized environments," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '11. New York, NY, USA: ACM, 2011, pp. 29–40.

[28] K. Okamura and Y. Oyama, "Load-based covert channels between xen virtual machines," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10. New York, NY, USA: ACM, 2010, pp. 173–180.

[29] S. Sellke, C.-C. Wang, S. Bagchi, and N. Shroff, "Tcp/ip timing channels: Theory to implementation," in *INFOCOM 2009, IEEE*, April 2009, pp. 2204–2212.

[30] M. Hussain and M. Hussain, "A high bandwidth covert channel in network protocol," in *Information and Communication Technologies (ICICT), 2011 International Conference on*, July 2011, pp. 1–6.

[31] J. Xiao, Z. Xu, H. Huang, and H. Wang, "Security implications of memory deduplication in a virtualized environment," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–12.

[32] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:6.

[33] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, ser. CT-RSA'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 1–20.

[34] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 2006, pp. 473–482.

[35] J.-F. Lalande and S. Wendzel, "Hiding privacy leaks in android applications using low-attention raising covert channels," in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, Sept 2013, pp. 701–710.

[36] S. Cabuk, C. E. Brodley, and C. Shields, "IP covert timing channels: design and detection," in *ACM Conference on Computer and Communications Security*, 2004, pp. 178–187.

[37] A. Lackorzynski and A. Warg, "Taming subsystems: Capabilities as universal resource access control in l4," in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, ser. IIES '09. New York, NY, USA: ACM, 2009, pp. 25–30.

[38] (2014, May) L4re home page. http://os.inf.tu-dresden.de/L4Re.

[39] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg, "The performance of μ-kernel-based systems," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 66–77, Oct. 1997. [Online]. Available: http://doi.acm.org/10.1145/269005.266660

[40] A. Lackorzynski, A. Warg, and M. Peter, "Generic virtualization with virtual processors," in *Proceedings of the Twelfth Real-Time Linux Workshop*, 2010.

[41] M. Peter, H. Schild, A. Lackorzynski, and A. Warg, "Virtual machines jailed: Virtualization in systems with small trusted computing bases," in *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, ser. VDTS '09. New York, NY, USA: ACM, 2009, pp. 18–23. [Online]. Available: http://doi.acm.org/10.1145/1518684.1518688

[42] S. Liebergeld, M. P. Peter, and A. Lackorzynski, "Towards modular security-conscious virtual machines," in *Proceedings of the Twelfth Real-Time Linux Workshop*, 2010.

[43] K. Elphinstone and G. Heiser, "From l3 to sel4 what have we learnt in 20 years of l4 microkernels?" in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 133–150. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522720

[44] (2014, May) Technical features – Pandaboard. http://pandaboard.org/content/platform.

[45] (2014, May) Technical features – Minnowboard. http://www.minnowboard.org/technical-features/.

[46] (2014, May) L4linux home page. http://l4linux.org.

[47] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni, "Read-copy update," in *In Ottawa Linux Symposium*, 2001, pp. 338–367.

[48] 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*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687

[49] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9.

[50] B. Lipinski, W. Mazurczyk, and K. Szczypiorski, "Improving hard disk contention-based covert channel in cloud computing environment," *arXiv preprint arXiv:1402.0239*, 2014.

[51] Y. Lin, L. Ding, J. Wu, Y. Xie, and Y. Wang, "Robust and efficient covert channel communications in operating systems: Design, implementation and evaluation," in *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*. IEEE, 2013, pp. 45–52.

[52] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "sel4: from general purpose to a proof of information flow enforcement," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 415–429.

[53] J. Wu, L. Ding, Y. Lin, N. Min-Allah, and Y. Wang, "Xenpump: a new method to mitigate timing channel in cloud computing," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 678–685.

[54] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220.

[55] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560537

[56] M. von Tessin, "The clustered multikernel: An approach to formal verification of multiprocessor operating-system kernels," Ph.D. dissertation, Ph. D. thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2013.

[57] (2014, July) Camkes vmm. http://github.com/seL4/camkes-vm-manifest.