The Last Mile

An Empirical Study of Timing Channels on seL4

David Cock, Qian Ge, Toby Murray, Gernot Heiser
NICTA and UNSW, Sydney, Australia
david.cock@nicta.com.au, qian.ge@nicta.com.au, toby.murray@nicta.com.au, gernot@nicta.com.au

ABSTRACT

Storage channels can be provably eliminated in well-designed, high-assurance kernels. Timing channels remain the last mile for confidentiality and are still beyond the reach of formal analysis, so must be dealt with empirically. We perform such an analysis, collecting a large data set (2,000 hours of observations) for two representative timing channels, the locally-exploitable cache channel and a remote exploit of OpenSSL execution timing, on the verified seL4 microkernel. We also evaluate the effectiveness, in bandwidth reduction, of a number of black-box mitigation techniques (cache colouring, instruction-based scheduling and deterministic delivery of server responses) across a number of hardware platforms. Our (somewhat unexpected) results show that while these defences were highly effective a few processor generations ago, the trend towards imprecise events in modern microarchitectures weakens the defences and introduces new channels. This demonstrates the necessity of careful empirical analysis of timing channels.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Information Flow Controls*

General Terms

Security, Measurement, Performance

Keywords

Confidentiality; covert channels; side channels; mitigation; microkernels; cache coloring; seL4

1. INTRODUCTION

Unanticipated information leaks are one of the oldest problems in computer security, with documented cases from as early as the 1940s [NSA, 1972]. Such leaks are traditionally classified as either storage or timing channels, depending on whether time is used to exploit them [Wray, 1991].

We can now prove the absence of storage channels in small and well-designed security-critical software, such as the seL4 microkernel [Klein et al., 2009, 2014], as demonstrated by the recent verification of intransitive noninterference for seL4 [Murray et al., 2013], in C. Notwithstanding some recent promising work, e.g. on proving upper bounds for and the absence of cache side channels in block cipher implementations [Doychev et al., 2013; Köpf et al., 2012] or proving the absence of cache leakage in an abstract hypervisor model [Barthe et al., 2012], proofs regarding timing channels in an operating system, even one as small as seL4, is beyond the reach of current approaches.

Therefore, for now such channels must be dealt with *empirically*. Their bandwidth can be measured by careful experiment, and the effectiveness of mitigations assessed according to how they reduce this. Such an empirical approach must be based on sound information theory, by accurately measuring and analysing the *channel matrix* [Shannon, 1948].

Figure 1 shows the empirical channel matrix (see Section 4.1) of the cache channel under seL4 on the Exynos4412 platform, which is summarised in Table 1. Each point $\{x, y\}$ gives the probability of an attacker making a particular observation (y: number of cache lines touched within a single timeslice) given the working set size (x: the number of cache lines evicted) of a sending program. Darker colours are higher probabilities, shown on the scale at right. The sender here might be an unwitting victim or might collaborate with the attacker. The clear correlation between the sender's working set size and the attacker's (most likely) observations, shown in the narrow band of non-zero probabilities, shows that the attacker can infer this size with high confidence. The bandwidth of the channel is calculated from this matrix (see Section 4.2), and in this case is 2400 bits/sec.

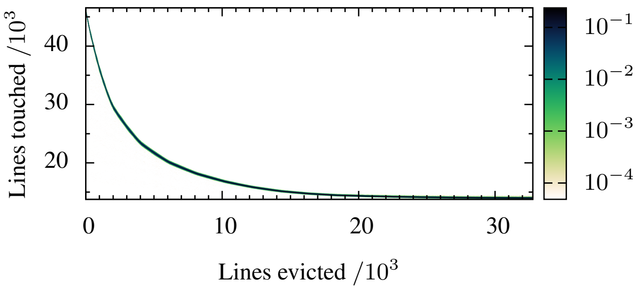

Figure 1: Exynos4412 cache channel matrix, no countermeasure. $N = 1000$, $B = 2400$ b/s. Colours indicate probabilities, further explanation in Section 4.1.

seL4 is a general-purpose operating system (OS) kernel, designed for security- as well as safety-critical use cases. Its notable features are comprehensive formal verification, with a complete proof chain from high-level security and safety properties to binary code [Klein et al., 2014], and performance at least as good as that of traditionally-engineered kernels [Elphinstone and Heiser, 2013].

These desirable properties come at a cost. Formal verification is expensive: a disincentive to modifying the system for a particular use case (although no worse than traditional assurance processes such as Common Criteria [NIST]). We thus look for ways to combat timing channels in seL4 without undermining its general-purpose nature.

We consider several timing channels that are relevant to an seL4-based system, and how they can be mitigated with minimal overhead, and minimal changes (hence re-verification) to the kernel. The need for low overhead rules out those that add noise (see Section 2). Furthermore, we only consider *black box* techniques, which require no insight into the internals of software running on seL4, as retrofitting security into complex software is generally impossible.

We do not yet aim for comprehensive coverage of timing channels related to seL4, but analyse representative examples in detail, to explore the limits of what we can achieve under the above constraints. A wider study of timing channels is planned for future work. Despite these limitations, we make several unexpected observations that generalise beyond seL4.

We look at one *local* vulnerability, the *cache-contention channel*, and two countermeasures, *instruction-based scheduling* [Stefan et al., 2013] and *cache colouring* [Liedtke et al., 1997]. We also examine a *remote* vulnerability, the distinguishing portion of the Lucky 13 attack of AlFardan and Paterson [2013] against DTLS in OpenSSL 1.0.1c.

The contribution of this paper is robust empirical evidence, at very high confidence, for the following claims:

- Black-box techniques, such as instruction-based scheduling and cache colouring, can be highly effective, but are less so on modern processors. For example, even with a partitioned L2 cache, flushing the L1 and TLB on a context switch, we still see a cache-channel bandwidth of 25b/s on a recent ARM processor (Exynos4412 see Section 5.4). This could (in theory), be exploited to leak a 1024 bit encryption key in a little over 40 seconds.
- On recent ARM processors, the instruction counter provides a timing channel not previously described, and with a bandwidth of 1100b/s, if exploited using the preemption tick as a clock. This channel can be closed by virtualising the counter.
- The "constant-time" Lucky 13 fix in OpenSSL 1.0.1e still exhibits a considerable side channel, at least on ARM.
- Operating-system techniques provide better mitigation of OpenSSL's Lucky 13 channel at lower performance penalty ($10\mu s$ vs. $60\mu s$ latency).

We conclude that timing channels, especially local channels, remain a real threat and are becoming more difficult to close. However, in the right environment, simple mitigations, carefully deployed on a well-designed kernel, can be effective. As hardware becomes more complex and opaque, any assurance case must be backed by solid, empirical analysis on the deployment platform.

2. TIMING CHANNELS: BACKGROUND

A *timing channel* transfers information in the (relative) timing of events. The OS does not usually control the timing of all events in the system. Timing channels thus often bypass operating system protections, and so pose a threat to confidentiality.

Despite decades of research, timing channels continue to plague mainstream systems [AlFardan and Paterson, 2013; Hund et al., 2013]. Historically, concern about timing channels was confined to high-assurance systems: certified separation kernels are required to limit their bandwidth [IAD] and the NSA provides guidance on how to avoid cache channels for systems built on secure real-time OSes [NSA]. While they remain a concern for modern high-assurance systems [Owen et al., 2011], timing channels are now recognised as a threat to co-tenant cloud computing [Ristenpart et al., 2009], in which mutually distrusting *tenants* pay for access to common computing infrastructure. Hence timing channel mitigation has again become a hot topic in computer security [Kim et al., 2012; Stefan et al., 2013; Zhang and Reiter, 2013].

One high-bandwidth timing channel is the *cache-contention channel*, which we cover in Section 5. In a cloud context, this channel has been exploited to learn high-value secrets like encryption keys [Zhang et al., 2012]. An otherwise isolated sender and receiver share one or more processor *caches*, which reduce access times to blocks of memory by keeping copies close to hand. A channel exists when the sender can influence which of the receiver's blocks are in the cache, since this affects the time it takes the receiver to access its memory. The event being observed here is the completion of a memory access by the receiver; the sender influences how long this event takes to arrive.

To measure the time between events, the receiver needs a *clock*: an independent event source. The observed channel is also a clock, each observation constituting an event. A timing channel exists between a sender and a receiver whenever the receiver has access to two clocks, and the sender controls their relative rates [Wray, 1991].

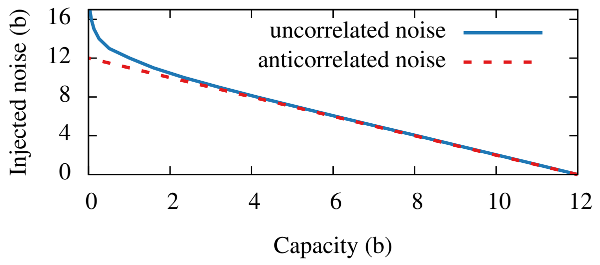

Figure 2: Correlated vs. anti-correlated noise against channel capacity.

Mitigation strategies, or *countermeasures*, are techniques that reduce information transmitted on a channel. Established strategies fall into three categories: (i) restricting the receiver to a single clock, (ii) limiting the sender's influence over the rate of the receiver's clocks (i.e. increasing determinism) and (iii) introducing noise into these clocks to make it harder to recover the signal being transmitted (as in fuzzy time [Hu, 1991]).

Introducing noise is inefficient if high security is needed: Figure 2 plots the level required to reduce the capacity of a 12-bit channel to any desired level, if that noise is either uncorrelated (and uniformly distributed), or perfectly anti-correlated with the signal (i.e. reducing it). While 12 b of anticorrelated noise closes the channel, the level of uncorrelated noise required increases asymptotically as we approach zero. Reducing the capacity by more than an order of magnitude requires huge amounts of noise, severely degrading overall system performance. The countermeasures we consider all build on this insight.

As a program can always observe its own progress, it always has access to one clock, its program counter (PC). Hence, restricting to a single clock requires denying any access to wall-clock time *and* ensuring that all observable events are synchronised to the PC, or ensuring that the PC is synchronised to wall-clock time.

Preemptive schedulers usually allocate processing time in fixed *time slices*. If the receiver can detect preemption events (e.g. via a helper thread), it obtains enough wall-clock time information to calibrate its PC clock, and thus time the channel events.

Instruction-based scheduling mitigates this channel by preempting not at fixed intervals, but after some fixed number of instructions. Stefan et al. [2013] explored this approach, and we cover it in Section 5.3.

Cache partitioning [Liedtke et al., 1997] is a well-known and recently-explored [Godfrey, 2013] countermeasure that ensures that the sender cannot influence which of the receiver's blocks are in the cache, and thus cannot alter the time taken for the receiver to access its memory. We cover this in Section 5.4.

We cannot deny wall-clock time to a *remote attacker*. We instead rely on making receiver-observable events deterministic. We examine this case using a remote client, the receiver, interacting with a server, who is the (unwitting) sender. Any variation in response time creates a channel. We mitigate this by enforcing a minimum bound on the server's response time; ensuring that responses are only released after some pre-determined interval, in order to implement a delay-based policy (e.g. [Askarov et al., 2010; Zhang et al., 2011]). This is the subject of Section 6.

3. THREATS & COUNTERMEASURES

As described, we explore mitigations against several different attacks, each with its own threat model. Across all scenarios we assume an attacker who is trying to learn some secret information. We make no assumptions about this secret—for instance, that it is selected uniformly at random. We also assume an attacker with arbitrarily high computational power. Our goal is to prevent the attacker from learning the sender's confidential information.

The mitigations have in common that they are *noiseless*, i.e. they attempt to reduce the signal on the channel rather than increasing the noise, aiming to minimise performance impact. They are also *black box* techniques, i.e. do not require modifications to or even an understanding of the internals of applications: all are implemented at the OS level.

3.1 Cache channel

As a *local vulnerability* we examine the *cache-contention* channel, which arises when sharing a memory subsystem between otherwise isolated domains. Here the receiver attempts to obtain secrets from a malicious sender partition. We assume a single-core, time-shared system, such as a multi-level secure (MLS) system or cross-domain solution. We assume further that the system has been configured appropriately so that no storage channels exist between the sender and receiver, and no devices are shared between them other than the CPU, bus and memory hierarchy (caches and main memory). The absence of storage channels implies that no region of physical memory is shared between the attacker and sender.

We assume a well-designed system with a minimal trusted computing base (TCB). The TCB will not intentionally leak secrets, but other components might. Under the classical distinction [Wray, 1991] between *covert channels* and *side channels*, in which the former involves intentional leakage while the latter is unintentional, we assume that trusted components leak secrets only over side channels, while untrusted ones also employ covert channels.

We use two countermeasures here: *Instruction-based scheduling* (IBS) and *cache colouring*, both introduced above. The former synchronises the clocks observable by the receiver, while the latter eliminates contention on the cache (i.e. the signal). In addition to the above assumptions, IBS requires that the receiver can be isolated from any notion of real time (as would be provided by physical devices or network connectivity). It is therefore only applicable in restricted circumstances.

3.2 Lucky Thirteen

As a *remote vulnerability*, in Section 6, we reproduce the distinguishing attack ("Lucky thirteen") of AlFardan and Paterson [2013] against DTLS, as implemented in OpenSSL version 1.0.1c, and demonstrate that the current version (1.0.1e) is still vulnerable.

We mitigate this channel, with better performance and lower overhead than the state of the art solution, using a black box technique: we employ real-time scheduling to precisely delay messages and thus hide timing variation.

4. METHODOLOGY

We work from a large corpus of statistical observations. From this we construct a *channel matrix*, and calculate summary measures, such as Shannon capacity, similarly to Gay et al. [2013]. Comparing the bare and the mitigated channels allows us to determine the effectiveness of a mitigation strategy.

We view the channel connecting a *sender* and a *receiver* as a pipe, into which the sender places *inputs*, drawn from some set I, and the receiver draws *outputs*, from some set O. For instance, in the cache-contention channel, the sender might touch some subset of its allocated memory, to ensure that a particular fraction of the cache is filled with its modified data; while the receiver touches as many lines as it can in some interval. The number of dirty lines that must be cleaned to RAM affects the receiver's progress. The input $i \in I$ is thus the number of lines touched by the sender, and the output $o \in O$ the number touched by the receiver, from which it attempts to infer i.

4.1 The Channel Matrix

The channel matrix captures the end-to-end behaviour of a channel. It has a row for each output $o \in O$ and a column for each input $i \in I$. The value at position $\{i, o\}$ gives the (conditional) probability of the receiver seeing output o (e.g. touching o lines) if the sender places input i into the channel (evicting i lines).

For example, Figure 1 is the channel matrix for the cache channel as measured on the Exynos4412 with no countermeasures. Each point gives a conditional probability, with darker colours for higher values, on a log scale. For instance, if the sender evicts 8,000 lines, the receiver will touch around 20,000. Here, the output clearly varies with the input (the more lines evicted, the fewer the receiver touches), and the figure intuitively captures this correlation and thus the existence of the channel.

Each channel matrix is built by testing all possible inputs, and observing for each a large number, N, of outputs (the *sample size*). Counting these gives a histogram that records for each output, o, the number of times, $n_{i,o}$, it was observed for each input, i. The estimated conditional probability of seeing o given i, the cell $\{i, o\}$, is thus $n_{i,o}/N$. Each column of the channel matrix in Figure 1, for instance, consists of 1000 observations (sample size $N = 1,000$).

We use a synthetic receiver to observe the channel. For the cache-contention channel depicted in Figure 1, the receiver uses the preemption tick to provide a regular sampling interval to measure the number of lines it managed to touch. We assume a malicious sender (see Section 3.1): we use a synthetic sender that varies its cache footprint according to i, the value to transmit.

For the remote channel (Section 6), for which we assume non-malicious (unintended) leakage, OpenSSL's vulnerable DTLS implementation forms the sender. The receiver in this case executes the distinguishing portion of the Lucky 13 attack [AlFardan and Paterson, 2013], which measures the response times for two different packets, M_0 or M_1. The set I consists of these two input packets.

Owing to the large number of input and output symbols (columns and rows), the channel matrices themselves are large—the analogous matrix to Figure 1 for the E6550 (see Table 1) would occupy 4.8 TiB if stored in full. We take advantage of the sparseness of the matrices—for any given input there are a only a small number of outputs that occur with non-zero probability—to compress them. Even so, the largest of our compressed matrices still occupies 380 MiB when using single-precision floating point to store its entries.

The channel matrix represented by Figure 1 has 32,768 columns (input symbols) and 45,000 rows (output symbols), for a total of 1.5 billion cells (conditional probabilities). This is much larger than those usually considered in the literature, and demonstrates that a numerical approach scales to realistic problem sizes.

4.2 Measures of Leakage

Given the channel matrix, we can calculate the *Shannon capacity* [Shannon, 1948], denoted C, a standard summary measure of capacity. We use a sparse matrix implementation of Yu's improved form [Yu, 2010] of the Arimoto-Blahut algorithm (ABA) [Arimoto, 1972; Blahut, 1972]. Multiplying by the sampling rate (333 Hz in most of our experiments, as explained shortly) gives the *channel bandwidth*, denoted B, in bits per second.

There are two additional quantities that we use where appropriate (e.g in Section 6): the *maximum vulnerability*, denoted $V_{\max}$, and the *min-leakage*, denoted $\mathcal{ML}$. $V_{\max}$ is the greatest likelihood (among all possible secrets), of an optimal (computationally unbounded) attacker correctly guessing the secret, given what it has observed. This is a safe upper bound on the vulnerability of the system. As we shall see in Section 6, the bound is tight, as it can be achieved by an attacker in sufficiently simple examples.

$\mathcal{ML}$ is a pessimistic analogue to the Shannon capacity. Whereas Shannon capacity can tell us the average amount of information leaked by a channel, min-leakage gives the worst case. It is the rate of change of the *min-entropy* (H_∞), given an observation. Thus $H_\infty(\text{final}) = H_\infty(\text{initial}) - \mathcal{ML}$. The min-entropy, in turn, is simply the (log) vulnerability of a distribution given no more leakage i.e. $H_\infty = -\log_2 \max_x P_x$. For further details see Köpf and Basin [2007]; Smith [2009].

4.3 Low-Capacity Channels

For a matrix such as Figure 1, the existence of a channel is obvious, and calculating its bandwidth is straightforward. In other cases, however, it is not. Figure 3 shows the same channel with a countermeasure applied (cache colouring). The figure shows plenty of noise but no apparent variation between columns, suggesting that different inputs are indistinguishable to the receiver. We nonetheless calculate a nontrivial bandwidth of 27 bits per second. We analyse this residual channel in Section 5.4.

A channel with a true bandwidth of zero may nonetheless appear to have a small nonzero bandwidth when we sample it: given a finite number of samples, the reconstructed output distributions for two different inputs will appear slightly different due to sampling error. This will make inputs appear distinguishable when in fact they are not. Analysis of a large number of synthetic matrices demonstrates that the effect of sampling error on small channels ($C \ll 0.1$ b) is to increase their apparent capacity (by making identical distributions appear to differ slightly) and on large channels ($C \gg 0.1$ b) is to decrease it (by making the output distributions appear to overlap more than they really do), although the latter effect is small.

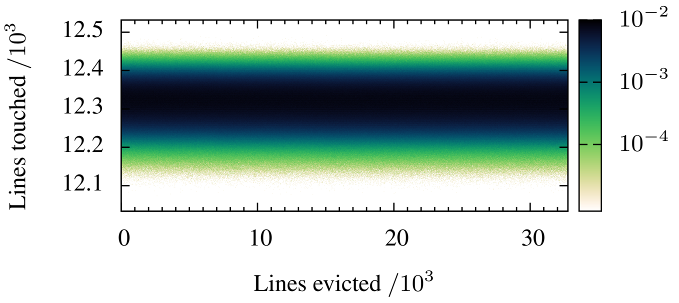

Figure 3: Exynos4412 cache channel with cache colouring. $N = 7200$, $B = 27$ b/s, $CI_0^{\max} = 15$ b/s.

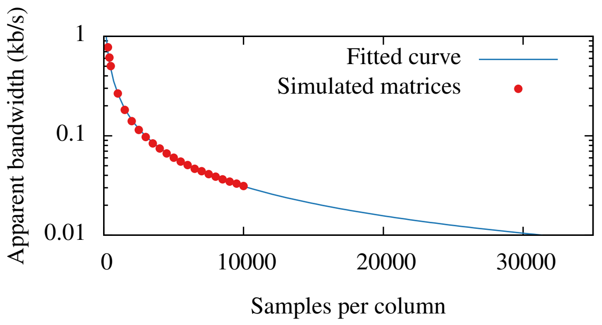

Figure 4: Apparent bandwidth against number of samples (N) for 0-bandwidth matrices derived from Figure 3, with least-squares fitted curve.

We use a simple statistical test to determine whether any apparent capacity (and hence bandwidth) results from sampling error. If the capacity is really zero, the output distribution (a vertical slice through the matrix) must be identical for all inputs (otherwise they would be distinguishable). In that case, the samples that made up each column must be drawn from the same distribution. We obtain this hypothetical distribution on outputs by averaging the channel matrix along its rows, resulting in an effective sample size of N times the number of columns. From this we sample 1,000 new matrices, each representing a true zero-capacity channel, and calculate their apparent capacities. The greatest bandwidth among these (multiplying by sample rate) we label $CI_0^{\max}$, or the greatest value in the 99.9% confidence interval for the apparent bandwidth of a zero-capacity channel. If the actual measured bandwidth lies above $CI_0^{\max}$, then there is only a one in 1,000 chance that this apparent capacity is the result of sampling error, and is strong evidence that there is a real underlying channel.[1]

For example, in Figure 3, $B = 27$ b/s, which is above $CI_0^{\max} = 15$ b/s, and thus there is a channel, despite the graph seemingly showing no horizontal variation.

Figure 4 shows how the number of samples required scales with the desired resolution. The points give the apparent bandwidth solely due to sampling error for matrices (derived from Figure 3, as already described), with a varying number of samples per column. The noise level drops with $1/x$—the curve is a least squares fit to $\frac{1}{ax+b}$. We see that in order to detect a channel of bandwidth 10 b/s, we would require 32,000 samples per column (for a threshold of detection where the noise is lower than the measured signal). Clearly, at some point the number of samples required will become

[1]If the apparent bandwidth is *below* $CI_0^{\max}$, the test is inconclusive: there is *no evidence* of a channel, but that is not evidence of no channel.

unfeasibly large, and a different technique will be required. The current approach does suffice however, for a number of small residual channels, as we shall shortly demonstrate.

4.4 Data collection

Collecting observations takes time. Each sample used to build Figure 3, for example, takes 3 ms. The 230×10^6 samples in this figure thus represent 200 hours of cumulative observation. In total, this project involved collecting 4.1 GiB of (compressed) sample data, across roughly 2,000 hours of observations over a 12 month period. A large number of samples was necessary to obtain the statistical power to detect the smallest of the channels that we report.

The software used to generate these matrices, and perform the necessary statistical analysis is available as open source.[2]

The local cache channel is highly sensitive to hardware properties (memory architecture and processor microarchitecture). We therefore evaluate it on a number of recent ARM and x86 platforms, Table 1 shows the platforms and their properties. In all cases, our experiments were run with the sender and receiver sharing the same CPU core.

5. LOCAL CACHE CHANNEL

5.1 Exploiting the channel

Caches are divided into *lines*, small, equally-sized blocks of a few dozen bytes in length. Each line holds the contents of an aligned memory block of the same size. Modern CPU caches are *set associative*, meaning that each memory block may reside in a fixed subset of cache lines, typically determined by a number of *index bits* taken from the block's address. The cache lines are thus partitioned into a number of identically-sized *sets*; the number of lines per set is the *associativity* of the cache. The address of a memory block determines the unique cache set in which it may reside, and thus which lines it may occupy. The subset of the cache in which a particular memory block can reside is its *cache colour*.

```
char A[L][L_SZ];

void sender(void) {
  int S;

  while(1) {
    for(i=0;i<S;i++) {
      A[i][0] ^= 1;
    }
  }
}
```

```
char B[L][L_SZ];
volatile int C;
void receiver(void) {
  while(1) {
    for(i=0;i<L;i++) {
      B[i][0] ^= 1;
      C++;
    }
  }
}

void measure(void) {
  int R, C1, C2;
  while(1) {
    C1=C;
    do { C2=C; }
      while(C1==C2);
    R=C2-C1;
  }
}
```

Figure 5: Preemption tick exploit code.

On loading a block into an already full set, some other entry must be evicted. Thus, if sender and receiver have access to (disjoint) blocks of memory that map to the same cache set, the sender can evict the receiver's blocks from the cache by touching its own (loading them into the cache). This is the basis of the cache-contention channel we analyse in this section.

Pseudocode to exploit this channel is given in Figure 5. Sender and receiver run alternately, sharing a core, with access to disjoint memory partitions. Arrays `A` and `B` each cover the entire last-level cache (L2). These arrays are allocated from physically contiguous memory, thus covering all cache sets. The receiver touches one word of each line in the cache, filling the cache with its own data. By measuring its rate of progress (via a helper thread, `measure()`), the receiver infers how many of its blocks were already cached. The sender communicates by evicting some number, `S`, of these (the channel input). The receiver sees the number of lines touched in a fixed interval, `R` (the channel output) which, as established, depends on `S`.

In this example we use the *preemption tick* to determine the measurement interval, as an example of a clock that is difficult to eliminate, although any regular event would do. Here, the simple round-robin scheduler of seL4 inadvertently provides a precise real-time clock.

As indicated, we consider two mitigations against this channel. *Cache colouring* eliminates the channel by partitioning the cache between sender and receiver, and requires no other restrictions on the system. *Instruction-based scheduling* prevents the use of the preemption tick as a clock by tying it to the receiver's progress, but to be useful requires that all other clocks have also been removed. It is, however, applicable to other channels (e.g. the bus), while cache colouring is specific to the cache channel.

5.2 Unmitigated channel

We first analyse the channel with no countermeasures, to establish a baseline. Figure 1 gives the results for the Exynos4412, obtained by taking $N = 1000$ samples for each value of S. All tested platforms produce very similar results.

As the sender evicts more of the receiver's lines from the cache, the number of cache lines that the receiver touches in each time slice decreases. This occurs as it takes longer for the receiver to touch evicted lines, and thus it touches fewer lines during each fixed time slice. The unmitigated channel's Shannon capacity, or the expected leakage *per observation*, is calculated from the channel matrix as explained in Section 4.2. The capacity of this channel is 7 bits.

We calculate bandwidth as follows: Each observation requires three time slices (of 1 ms each), one for each of the sender and the two receiver threads, or a total of 333 observations per second. As each leaks 7 bits, the bandwidth is approximately 2.3 kb/s.

5.3 Instruction-based scheduling

As described, IBS removes the preemption-tick clock from the receiver, assuming that all other time sources are already gone. On seL4 we could thwart this particular exploit by preventing the receiver from creating its helper thread: seL4's strong resource management model provides control over the kernel-scheduled threads a task can create. This would further restrict application, and does not apply to most operating systems.

To implement IBS, we modify seL4 to trigger preemptions each time some fixed number, K, of instructions has executed. This is easily achieved with the help of the *performance management unit* (PMU) available on modern CPUs, which can be configured to generate an exception after some number of events (here retired instructions).

Implementing this in seL4 on ARM requires changing only 18 lines of code, and x86 is similarly straightforward. Because this

[2]http://ssrg.nicta.com.au/software/TS/channel_tools/

Processor	iMX.31	E6550	DM3730	AM3358	iMX.6	Exynos4412
Manufacturer	Freescale	Intel	TI	TI	Freescale	Samsung
Architecture	ARMv6	x86-64	ARMv7	ARMv7	ARMv7	ARMv7
Core type	ARM1136JF-S	Conroe	Cortex A8	Cortex A8	Cortex A9	Cortex A9
Released	2005	2007	2010	2011	2011	2012
Cores	1	2	1	1	4	4
Clock rate	532 MHz	2.33 GHz	1 GHz	720 MHz	1 GHz	1.4 GHz
Timeslice	1 ms	2 ms	1 ms	1 ms	1 ms	1 ms
RAM	128 MiB	1024 MiB	512 MiB	256 MiB	1024 MiB	1024 MiB
L1 D-cache						
size	16 KiB	32 KiB	32 KiB	32 KiB	32 KiB	32 KiB
index	virtual	physical	virtual	virtual	virtual	virtual
tag	physical	physical	physical	physical	physical	physical
line size	32 B	64 B	64 B	64 B	32 B	32 B
lines	512	512	512	512	1024	1024
associativity	4	8	4	4	4	4
sets	128	64	128	128	256	256
L2 cache						
size	128 KiB	4096 KiB	256 KiB	256 KiB	1024 KiB	1024 KiB
line size	32 B	64 B	64 B	64 B	32 B	32 B
lines	4096	65536	4096	4096	32768	32768
associativity	8	16	8	8	16	16
sets	512	4096	512	512	2048	2048
colours	4	64	8	8	16	16

Table 1: Experimental platforms.

change is small and localised, our previous experience [Klein et al., 2014] with re-verification of code changes for seL4 suggests that it should be straightforward to verify.

Effectiveness of instruction-based scheduling

We re-run the experiments used to generate Figure 1 with instruction-based scheduling enabled. Under IBS, timeslices are no longer constant, although in practice the variation is small. To compare with our other results, we normalise the bandwidth of the IBS channel to a sampling rate of 333Hz. This could be achieved, for example on the DM3730 (1GHz clock), by setting $K = 10^6$ (assuming 1 instruction per cycle). In practice, once sender and receiver start contending, and hence stalling more often, the real timeslice will grow (as instructions take longer on average to execute). The effect is to *reduce* the available bandwidth, and hence normalised results are a safe worst-case estimate.

The results from using IBS (for compatibility with existing data, we ran with $K = 10^5$ rather than $K = 10^6$) for all platforms are summarised in Table 2. For the simplest (and oldest) core, the ARM1136-based iMX.31, the mitigation is perfect (down to the limit of our statistical precision), showing a 20,000-fold reduction in capacity, from 1,400 b/s down to 0.1 b/s. However, for the more recent platforms, significant channels remain. On the Exynos, for example, the channel bandwidth is reduced by a factor of 87, still leaving a remaining channel of 27 b/s. While the channel matrix is now a very narrow band, it still has horizontal structure, as shown (greatly magnified) in Figure 6.

As we move to more complex cores the results get steadily worse, until we reach the Cortex A9-based Exynos4412 and the Conroe-based E6550, which show a reduction factor of only 6 and 154 respectively. Looking at the channel matrix for the Exynos4412 in Figure 6 suggests an explanation. As we preempt after 100,000 instructions, and each receiver loop iteration (and hence line touched) takes exactly 10 instructions, we expect to see precisely 10,000 lines touched per preemption. On the simpler cores, that is exactly what we see, but in Figure 6 the number

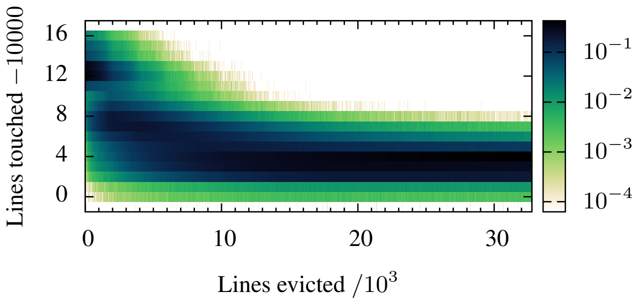

Figure 6: Exynos4412 cache channel with IBS. $N = 1000$, $B = 400$ b/s, $CI_0^{\max} = 1.2$ b/s.

clearly varies according to the level of contention.[3]

Most of the variation here is due to delaying the PMU interrupt. The interrupt arrives 12 iterations (120 cycles) late without contention, dropping to 8 (80 cycles) when the sender fills the L1 cache (512 lines, thus contending in the L2), and further to 40 cyc once we pass the size of a single L2 cache way (2048 lines), and start to see self-conflict misses. It seems that the core delays the exception until a break in the instruction stream, e.g. a stall due to a cache miss, and thus the overshoot drops as the rate of misses increases (and thus the likelihood of a stall shortly after the exception is raised).

An approach that we are yet to try, is to configure the PMU to interrupt a little earlier than when K instructions have been executed, and then single-stepping the processor using hardware breakpoints until precisely the Kth instruction. This strategy has previously proved effective in the context of execution replay [Dunlap, 2006].

5.4 Cache colouring

Unlike IBS, cache colouring does not require denying the re-

[3]The variation in Figure 6 is on the order of 10 cycles, and is expressed as an offset from the expected value of 10,000. Figure 8 (top) is presented similarly.

	Baseline		Instruction-Based Scheduling				Cache Colouring			
Platform	B (b/s)	N	B (b/s)	N	Factor	$CI_0^{\max}$	B (b/s)	N	Factor	$CI_0^{\max}$
iMX.31	1,400	7,000	0.1	10,000	20200	0.02	7.1	63,972	200	3.8
AM3358	1,600	6,000	0.6	10,000	2700	0.32	5.0	49,600	330	2.7
DM3730	1,800	8,000	0.5	10,000	3500	0.26	1.7	63,200	1000	0.9
iMX.6	2,100	800	-	-	-	-	12	11,400	180	6.3
Exynos4412	2,400	1,000	400	1,000	5.9	1.2	27	7,200	87	15
Exynos4412(TLB flush)	2,400	1,000	-	-	-	-	25	7,200	94	13
E6550(2ms TS)	1,500	1,000	9.5	600	150	11	76	4,836	19	42
E6550(improved)	3,000	800	-	-	-	-	120	7,500	26	62

Table 2: Mitigation effectiveness against the pre-emption clock across platforms. 1ms timeslice unless noted.

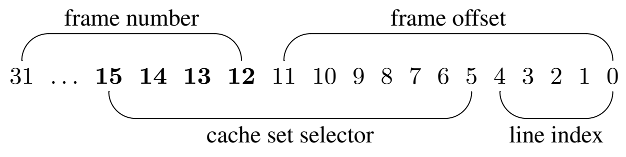

Figure 7: Cache colouring on the Exynos4412, showing *colour bits* 15–12, where frame number and cache set selector overlap.

ceiver wall-clock time. Colouring partitions the cache between sender and receiver, preventing contention. This is achieved by colouring all physical memory, and allocating disjoint colours to different partitions. Colouring of memory happens at page granularity, as this is the OS-level allocation unit.

Figure 7 illustrates colouring on the Exynos4412. Its 32 B cache lines are indexed by the 5 least significant bits (4–0) of the physical address (PA), while the next 11 bits (15–5), the *cache set selector*, are used to select one of 2048 16-way associative sets. A 4 kiB frame is identified by the top 20 bits (31–12) of the PA: its *frame number*. Note that the last four bits of the frame number (15–12, highlighted) overlap with the top bits of the cache selector—the sets covered by a frame depend on its location. Two frames whose addresses differ in any of these *colour bits* will never collide. We thus divide memory into a number (here $2^4 = 16$) of *coloured pools*, and assign partitions to separate pools.

We partition not just user code and data, but also the kernel heap (using seL4's allocation model [Klein et al., 2014]). We also replicate the kernel's code in each partition. An improved version (see below) also colours the kernel stack; colouring kernel global data is future work, as it is only small, and not under user control.

The L1 caches of our platforms either have only one colour (cache size divided by associativity does not exceed page size), and therefore cannot be partitioned by colouring, or are indexed by virtual address (which is outside of OS control). Therefore the L1 caches must be flushed on a partition switch to prevent a timing channel.

Cache Colouring Effectiveness

Table 2 summarises the results, and Figure 3 shows the channel matrix for the Exynos4412, those for other platforms are similar (see Cock [2014]). For the simpler cores (iMX.31, AM3358 & DM3730) we see a great reduction in bandwidth (factor 200–1000). Yet, every result here fails the statistical test introduced in Section 4: The bandwidth we see has less than a one in 1000 chance of being produced for a channel of a true bandwidth of zero.

The results are particularly unimpressive on the more complex Exynos4412 and E6550 (factor 20–90). We see a similar pattern as for IBS: mitigation is effective on older, simpler cores (although

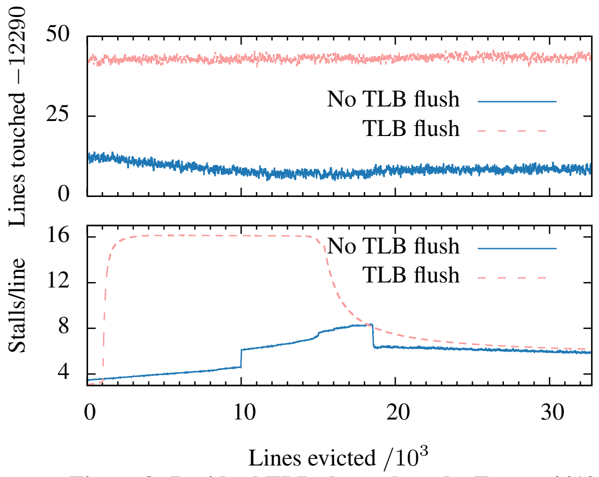

Figure 8: Residual TLB channel on the Exynos4412.

less so than IBS), and becomes steadily worse as the cores become more complex, although becoming more effective than IBS on the more recent chips.

Figure 8 provides an explanation for the Exynos4412. In the top plot, we see the column average, shown in blue (lower line), of Figure 3. This is the *expected value* of the channel output (number of lines touched), for each input (number evicted). Here we see a small, but clear variation, on the order of 5 parts in 10,000, depending on the eviction rate. The cause is shown by the corresponding blue (lower) curve in the bottom plot: as the rate of stalls due to TLB misses increases, the rate of progress of the receiver decreases. The sender and receiver are competing in the TLB, which is not partitioned by cache colouring (the first-level TLBs on this chip are fully associative, and thus cannot be coloured). Flushing the TLB on a context switch eliminates the variation, as shown by the red curve in the top plot, at the cost of an increased miss rate (bottom plot).

This effect occurs on all ARM platforms tested, and explains some of the residual bandwidth. The result of flushing the TLB on each context switch is given in Table 2, where the residual bandwidth has dropped from 27 to 25 bits per second. There is clearly still some interference effect, as the bandwidth is still higher than the confidence threshold of 13 b/s. We have not yet managed to identify this further source of contention, although we have ruled out contention in the branch predictors, which are reset by the full L1 cache flush between partitions. While it is disappointing that we have not yet managed to completely close the channel, the empirical approach ensures that we do not have a false sense of security.

On close inspection of the E6550 matrix of Figure 9, we find another small but detectable artefact at half the L2 cache size (32,768

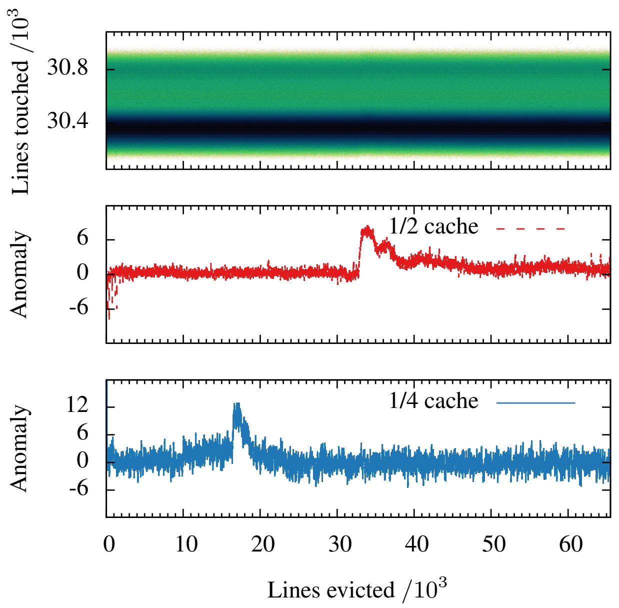

Figure 9: E6550 cache channel, cache colouring. $N = 4836$, $B = 76$ b/s, $CI_0^{\max} = 42$ b/s.

lines). The effect is clearer in the column averages, plotted in red.

This cannot be TLB contention, as the TLB on this chip is not tagged, and is thus automatically flushed on every context switch. Rather, it appears to be due to the sender triggering instruction-cache misses in the kernel. Note that, with the effective cache size halved by colouring, the observed effect coincides with the point at which the sender completely dirties the L2 cache. As the caches on this chip are inclusive, this also evicts kernel code for the sender's domain.

The correlation is confirmed by the blue curve, which plots the anomaly if the sender is given only 1/4 of the cache. As expected, the artefact moves to 16,384 lines. We eliminate this sharp artefact with an improved implementation: colouring the kernel stack, and flushing the L1 caches with coloured data arrays (see Section 5.6 for details on x86 L1-cache flushing).

The results are given in row 7 of Table 2: the observed bandwidth shifts from 76 to 120 bits per second. Taking into account that the new implementation has brought the timeslice length into line with other platforms (halving it from 2ms to 1ms), this represents an improvement of 30% (with a 1ms timeslice, the old implementation's bandwidth would double to 150 b/s). There is, however, clearly still a residual channel, as $CI_0^{\max}$ is only 62.

Overall, we see that while cache colouring remains broadly effective (if it can be implemented, and all residual channels are carefully eliminated), it is getting harder to implement on newer hardware. This is exactly the same trend that we see for IBS: Undocumented behaviour on complex CPUs is essential to both countermeasures, yet is becoming steadily harder to reverse-engineer, and sometimes renders implementation seemingly impossible.

5.5 Unexpected Channels

While analysing the results of the previous section, we discovered two interesting and unexpected results: First, a way to greatly increase the signal-to-noise ratio in a contention channel (thus boosting its usable capacity), and that the cycle counter provides an entirely new channel due to branch prediction.

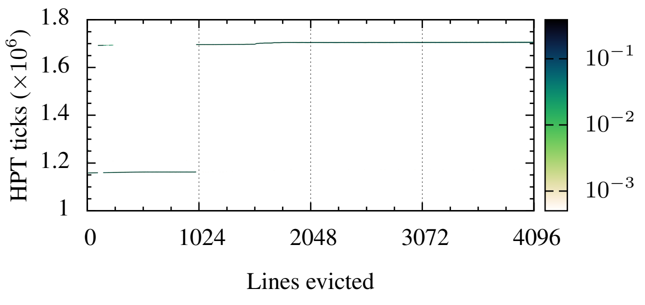

Figure 10: AM3358 cache channel, cache colouring, showing cycle counter variation. $N = 100$, $B = 2900$ b/s.

We instrument the `while` loop of the `measure` thread in Figure 5 to show the number of cycles spent executing the busy loop and the two background threads, on the AM3358. Given that the residual channel on this platform is small (5.0 b/s), we expect to see no strong correlation with the sender's eviction rate. Yet Figure 10 shows a strong effect, with a measured bandwidth of 2,900 b/s.

The cause is suggested by the fact that the variation is of almost exactly 532,000 cycles, or one 1 ms preemption period at 532 MHz. We see a slight variation in where the preemption point falls, leading to the inner loop either terminating immediately (if the thread is preempted, and the shared counter `C` updated, *after* it is read), or running for a full timeslice if the preemption falls elsewhere. A variation of a few cycles (maybe just one) is magnified enormously, *under the sender's control.*

The effect is to eliminate the noise in the receiver's measurement, allowing the full capacity to be realised. It is important to note that the number of discrete input and output symbols (the amount of underlying variation that the sender controls) still places an upper bound on channel capacity. This example reinforces the point we made in Section 2 that *adding noise is less effective than limiting the underlying signal*—Figure 10 shows that the noise can be eliminated by unexpected means, but the exploitable variation remains.

The Cycle Counter Channel

Some (barely visible) artefacts in Figure 10 point to a previously unreported channel. Zooming into the first few iterations (Figure 11) shows a weak effect at 7 evictions, where the previously constant value splits in three, and a stronger one at 10, where it settles on the lowest of the three. Each of these drops is almost exactly 53,200 lines, or 10% of a timeslice.

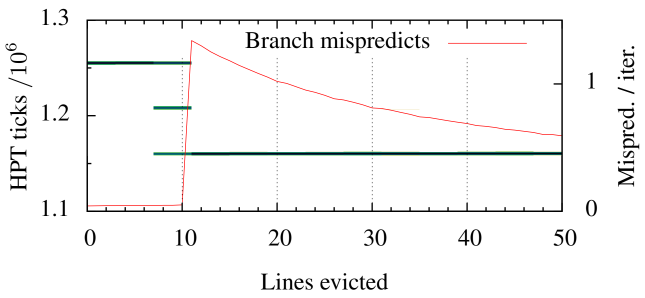

Figure 11: AM3358 cache channel, cache colouring. Cycle counter effect of mispredicts. $N = 997$, $B = 1100$ b/s.

The number of branch mispredicts per iteration (red line) provides an explanation. Assuming that the sender and receiver touch lines at roughly the same rate (we only recorded the receiver's rate), we see roughly one branch misprediction per loop iteration, once the loop is longer than 10 iterations, and 7 in rare cases.

The precise cause of these mispredicts is unclear (the loop should be correctly predicted for 9/10 iterations), but the correlation with the cycle counter is clear, with a small drop beginning at 7 iterations (exaggerated by the log scale for probability), and finishing at 10. It appears that a branch mispredict leads to the counter missing a single cycle. Note that the wall-clock time between samples is unaffected, only the cycle counter's value varies, giving an exploitable bandwidth of 1,100 bits per second.

We conclude that the reported value of the cycle counter is imprecise on ARM, and *the imprecision is correlated with branch mispredicts*. The cycle counter is globally visible, unless virtualised, thus giving rise to a timing channel not previously reported. This channel is distinct from traditional ones involving branch prediction, where contention in the branch target buffer (BTB) leads to variations in runtime [Aciiçmez et al., 2006]. The obvious defence is virtualising the cycle counter, which is possible on both x86 and ARM.

5.6 Cost of countermeasures

IBS can be implemented without run-time overhead, it simply replaces the timer with the PMU as an interrupt source. (It reduces fairness somewhat, as memory hogs now get longer time slices.)

Cache colouring has two costs: flushing the L1 caches and TLB on a partition switch, and reducing the effective cache size. The latter cost depends on the size of the working set of the application code: it is worst if the working set just fits into the cache, and negligible if the working set is less than half the cache size. The isolation provided by colouring can also occasionally increase performance [Tam et al., 2007].

The direct cost of an L1 flush is low on ARM (single instruction) and expensive on x86 (due to lack of support for a selective L1 cache flush, requiring the kernel to replace any useful data by traversing large arrays and jump tables). The indirect cost (of a cold cache) can be expected to be low: The flush is only needed at a partition switch, which only occurs at the end of a time slice (of 1 ms or $\approx$ 1M cycles). The DM3730's 512-line cache, with a miss latency of 12 cycles[4] takes roughly 6,000 cycles to refill, or 0.6% of a timeslice, which constitutes an upper bound on the indirect cost. In most cases this cost will be much smaller, as the L1 cache is normally cold after a partition switch even without flushing (given that other process have been executing for at least 1 M cycles).

The TLB flush also has low direct cost, and it is likely to be cold after a partition switch, resulting in low indirect cost. The full cost of a TLB refill on the DM3730 is $\approx$ 64 entries $\times$ 50 cycles $=$ 3200 cycles.

6. REMOTE TIMING SIDE-CHANNELS

Both countermeasures evaluated so far address *local* channels due to shared hardware. In this case we have some control over the attacker, either over its resources (as in cache colouring), or its access to time (as in IBS). *Remote* channels require a different approach, as the attacker is effectively unrestrained. In particular, a remote attacker must be assumed to have an accurate clock.

6.1 OpenSSL vulnerability

We first demonstrate the ease with which remote side-channel attacks can be carried out at essentially unlimited distance, and then present the *scheduled delivery* countermeasure, which uses a monitor to hide response-time variations.

As a realistic vulnerability, we replicate the strongest form of the Lucky 13 attack of AlFardan and Paterson [2013]—the distinguishing attack against Datagram TLS (DTLS) [Modadugu and Rescorla, 2004], with sequence number checking disabled. By successfully addressing this we also address its weaker forms, in particular plaintext recovery, which use the same mechanism.

The attack uses the fact that TLS first calculates the MAC (message authentication code, or digest), and then encrypts it. This allows intercepted packets to be submitted to a server, which will then decrypt and begin to process them *before* their authenticity is established. We exploit the non-constant execution time of the MAC check itself by manipulating the padding in the packet. Ultimately, we construct two packets: M_0 and M_1, that take a different length of time to process, before being rejected (the MAC of the manipulated packet will fail), where the time depends on the (encrypted) contents. We *distinguish* two encrypted packets by intercepting them, and forwarding them to the server.

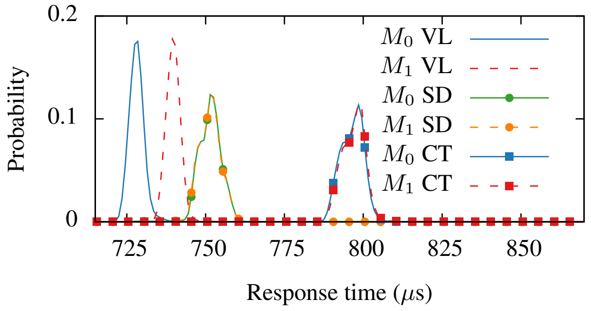

Figure 12: Response times for M_0 and M_1. Shows peaks for OpenSSL 1.0.1c (VL), scheduled delay (SD) and 1.0.1e (CT), demonstrating reduced latency. 10^6 samples, binned at 1μs.

Version	Hops	D (km)	$V_{\max}$	$\mathcal{ML}$ (b)	RTT$\pm\sigma$ (ms)
1.0.1c	1	0	1.00	0.99	0.73 ± 0.01
	3	0	0.60	0.11	1.2 ± 0.2
	4	4	0.77	0.57	1.30 ± 0.06
	13	12,000	0.63	0.21	180 ± 30
1.0.1e	1	0	0.62	0.07	0.80 ± 0.005
1.0.1c-sd	1	0	0.57	0.03	0.75 ± 0.005

Table 3: Vulnerability against network distance (Hops) and physical distance (D), for DTLS distinguishing attack.

Figure 12 shows the measured response times for the two packets, as measured from an adjacent machine (no switch). In the terminology of Section 4, the packets M_0 and M_1 are the two (secret) channel inputs, and the response time is its output. Each pair of peaks for M_0 and M_1 (labelled VL, SD and CT respectively) in Figure 12 forms a channel matrix with just two columns (M_0 and M_1), built by taking 10^6 observations for each input.

The leftmost (VL) peaks are the response times for the vulnerable implementation of OpenSSL 1.0.1c, on the AM3358. Times are measured, as in the original attack, by sending a modified packet immediately followed by a valid packet (also captured from the wire), and taking the response time of the second. This avoids the problem that DTLS does not acknowledge invalid packets. The victim executes an echo server, over TLS. As line 1 of Table 3 shows, these peaks are trivially distinguishable, allowing the at-

[4]http://www.7-cpu.com/cpu/Cortex-A8.html

tacker to correctly guess which packet was sent with near certainty ($V_{\max} = 100\%$). This is a leak of 0.99 b of min-entropy.

The rightmost (CT) peaks give the round-trip for the constant-time implementation of OpenSSL 1.0.1e, which substantially reduces the vulnerability—the curves are *almost* identical. However, they still differ measurably, as row 5 of Table 3 shows—the two can still be distinguished with $V_{\max} = 62\%$ probability, while to be completely secure, we should only be able to guess with 50% probability. This emphasises the difficulty of producing portable cross-platform constant-time code, and indicates that the production version of OpenSSL (as of writing) is still vulnerable.

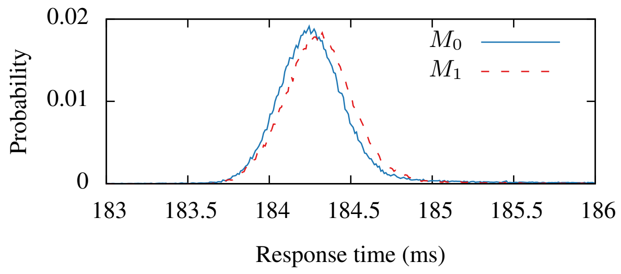

Figure 13: Response times for OpenSSL 1.0.1c, intercontinental distance. 10^5 samples, $10\mu s$ bins.

As rows 2–4 of Table 3 show, while distinguishability (and hence vulnerability) drops with increasing network distance, it does so very slowly. The attack is still easily feasible at the greatest separation that we could achieve—launching the attack from an Amazon EC2 instance in Oregon, USA, against our target machine in our laboratory in Sydney. Despite many routing hops, firewalls, and great physical distance, the attacker still guesses correctly with 62% probability given only one observation. We thus conclude that *network distance provides extremely poor protection against timing channels*, and that variation of only $10\mu s$ is easily distinguishable at any point on the internet with good connectivity. Figure 13 shows the two distributions. This supports the mathematical analysis summarised in Figure 2: adding noise (in this case network jitter), provides poor protection against timing channels. We now show that our preferred approach, reducing variance, is much more effective, as it removes the vulnerability at the source.

6.2 Scheduled Delivery

Our countermeasure to this channel is a system-level black-box approach that avoids the difficulty of producing portable constant-time algorithms. We build on the fast context-switches and well-understood temporal behaviour of seL4 [Blackham et al., 2011], to impose precise delays on communication. Recent work [Askarov et al., 2010] suggests *policies* for automatically setting such delays. We provide an efficient *mechanism*.

The effect of a manually-tuned delay is shown by the central (SD) pair of peaks in Figure 12. As rows 5 and 6 of Table 3 show, we achieve a lower vulnerability than the constant-time implementation of OpenSSL (57% distinguishability or 0.03 b leak of min entropy vs. 62% and 0.07 b). Despite this, our countermeasure reduces latency by 6% compared to CD. The better matching between the curves occurs as nothing in our implementation is data-dependent, and the only intrinsic penalty is the cost of blocking and restarting the server, which from the figure is $\approx 10\mu s$. We have not yet managed to find the cause of the small remaining variation in the response time. For a detailed analysis of this approach, see Cock [2014].

6.3 Cost

Figure 14 shows the overhead of SD. Each curve plots CPU load against packet ingress rate, up to the point at which which packet loss begins (single CPU). For the unmodified OpenSSL 1.0.1c (blue), load increases linearly, with loss beginning with the onset of saturation at 3000 packets per second. The constant-time OpenSSL 1.0.1e (red) shows a 10% CPU overhead, consistent with the increased latency observed in Figure 12, and correspondingly earlier saturation, at 2800 p/s. The extra cycles are wasted ensuring that execution time is always worst case.

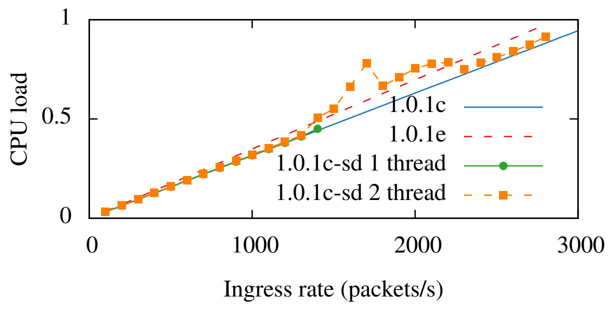

Figure 14: Performance and overhead of scheduled delivery, OpenSSL 1.0.1c, and 1.0.1e (constant-time).

The next curve (green), for a single-threaded server under SD, is close to that for the vulnerable version, with only 1.7% overhead. This is the benefit of not wasting time in a constant-time implementation. Instead of busy-waiting, we idle by entering a (low-power) sleep state, with obvious advantages for mobile devices.

This curve also demonstrates the downside: packet loss begins at 1400 p/s. Packets arriving while sleeping are dropped, limiting throughput. This is an extreme case, however, as the echo server does no work at all, so all CPU time is spent in OpenSSL itself. In any non-trivial system, the server will work while the packet handler sleeps, with no throughput loss once OpenSSL is less than half the load.

The orange curve shows that slack could be re-used to run a second server thread. This is not secure (as it transforms latency variation into throughput variation), but demonstrates that we need not suffer a throughput overhead, given a non-trivial application. Except the excursion between 1300 and 2200 p/s, due to our simplistic prototype ports of lwIP [Dunkels, 2001] and OpenSSL, we regain peak throughput of 2800 p/s, still with better overhead than constant-time.

7. DISCUSSION

Our results highlight the importance of a systematic empirical approach to timing channels. It is far too easy to overlook potential channels, and without establishing sound bounds on bandwidth, one could easily be fooled into a false sense of security.

Our work demonstrates some such pitfalls. For instance, one would expect cache colouring to be an effective countermeasure, even against an attacker with access to an accurate measure of wall-clock time. Specifically (excepting frequency scaling) we expected the cycle counter to be such a timing source. However, we found the cycle counter not only to be inaccurate on modern processors, but in fact influenced by cache misses, and thus creating a timing channel of its own!

In fact, *none* of the examined countermeasures were perfect: IBS, cache colouring, constant-time implementations and scheduled delivery all leave residual channels, a depressing realisation.

However, we note that our local exploits were performed under the most pessimistic assumption of a malicious agent exploiting a covert channel. The (generally small) remaining channels may provide sufficient protection in a side-channel scenario, such as a co-hosted cloud environment. However, we cannot say this with certainty.

We must also recognise the limitations of our approach: the precision of our estimates is always limited by the quantity of data available. Given a finite number of observations, we can only rule out channels down to a certain bandwidth; there is always the possibility of a residual channel hiding below the limit of our statistical precision. For example, even for our best result, IBS on the iMX.31 showing a bandwidth of essentially zero after 10,000 samples per column, the confidence interval $CI_0^{\max}$ extends to 0.1 b/s, meaning that there could be a channel of lower bandwidth that we simply cannot resolve. If a single column were to deviate with a probability of, say 10^{-6}, we would only expect to see it in one of 100 experiments.

Between the release of the iMX.31 in 2005, and the Exynos4412 in 2012, IBS has gone from an essentially perfect countermeasure, to a highly ineffective one. This highlights the strong effect that subtle (and generally undocumented) hardware effects have on countermeasures, and the value of careful empirical evaluation.

8. RELATED WORK

Our empirical approach to timing channels is similar to that of Gay et al. [2013] who measured *interrupt-related covert channels* (IRCCs) by experimentally determining the channel's Shannon capacity. However, they make the assumption that the channel output follows a binomial distribution, in order to compute Shannon capacity in a closed form, and so avoid working with very large channel matrices as we do (see Section 4). They *empirically* measure only the unmitigated channel bandwidth; however, earlier work of Mantel and Sudbrock [2007] involved a *theoretical* comparison of IRCC mitigation techniques under an information theoretic channel model.

Cache colouring was originally developed to assist real-time systems to partition the L2 cache into a number of non-overlapping domains [Liedtke et al., 1997]; its potential utility as a cache-channel mitigation technique is therefore obvious. Various hardware mechanisms for cache partitioning have been proposed [Jaleel et al., 2012], although none are available in the platforms we analyse.

This has recently been analysed by Godfrey [2013] (on the Xen hypervisor) using an actual side-channel attack, while we build a synthetic covert-channel attack, and measure the bandwidth reduction. Unlike Godfrey, we partition kernel as well as user memory.

STEALTHMEM [Kim et al., 2012] is a recent system that generalises the idea of cache partitioning, offering a limited amount of *stealth* memory, rather than partitioning the complete cache. While this leads to potentially less performance impact, it requires modifying applications and is only applicable to trusted entities (senders), while we treat applications as black boxes.

IBS works by correlating clocks, and so is related to deterministic execution techniques, originally used to debug systems [Aviram et al., 2010a,b; Bergan et al., 2010; Ford, 2012], although without requiring full determinism. It was implemented in the Hails web application framework [Stefan et al., 2013], specifically to address timing channels. Martin et al. [2012] propose modifying the CPU to add noise to the timestamp counter (RDTSC), which we argue is inefficient for high-security applications.

Scheduled delivery considers only the *external* behaviour of a component—its response time—which it delays to reduce leakage. Askarov et al. [2010]; Zhang et al. [2011] present an *adaptive* delay policy to counter remote timing side-channels. We present an efficient mechanism to implement such a policy.

9. CONCLUSIONS

We have examined representative locally- and remotely-exploitable timing channels on the verified seL4 microkernel, and suitable mitigation strategies. We find that instruction-based scheduling and cache colouring (against cache contention), and scheduled delivery against remote attacks, are easy to implement in seL4, without impacting its generality. The exception is the L1 cache flush needed for cache colouring, which x86 doesn't allowing, requiring expensive explicit cache trashing.

While these mechanisms are effective on older processors, performance optimisation in newer processors not only introduces imprecision in hardware-generated events, which manifests as non-determinism, but the degree of imprecision is frequently affected by user-controlled events, cache misses or branch mis-predicts, which introduces new channels. Thus more effort is required to treat the cache channel, forcing the OS developer to play catch-up with the processor manufacturers.

For remotely-exploitable channels we find that, at least for the Lucky-13 attack on OpenSSL, OS-level black-box approaches are more effective, and come with less latency penalty, than the official "constant-time" mitigation.

In summary: Closing timing channels remains difficult, even for a small high-assurance system like seL4. Unexpected results show the importance of a systematic experimental approach to determining channel bandwidth, to avoid a false sense of security.

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *2007 CT-RSA*, pages 225–242, 2006.

Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symp. Security & Privacy*, pages 526–540, San Francisco, CA, May 2013.

Suguru Arimoto. An algorithm for computing the capacity of arbitrary discrete memoryless channels. *Trans. Inform. Theory*, 18 (1):14–20, 1972.

Aslan Askarov, Andrew C. Myers, and Danfeng Zhang. Predictive black-box mitigation of timing channels. In *17th CCS*, pages 520–538, Chicago, Illinois, USA, 2010.

Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *2010 CCSW*, pages 103–108, Chicago, Illinois, USA, 2010a.

Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *9th OSDI*, Vancouver, BC, 2010b.

Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *25th CSF*, pages 186–197, 2012.

Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *9th OSDI*, pages 1–16, Vancouver, BC, Canada, 2010.

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *32nd RTSS*, pages 339–348, Vienna, Austria, Nov 2011.

Richard E. Blahut. Computation of channel capacity and rate-distortion functions. *Trans. Inform. Theory*, 18:460–473, 1972.

David Cock. *Leakage in Trustworthy Systems*. PhD thesis, University of New South Wales, 2014.

Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security*, 2013.

Adam Dunkels. Minimal TCP/IP implementation with proxy support. Technical Report T2001-20, SICS, 26, 2001. http://www.sics.se/~adam/thesis.pdf.

George Washington Dunlap, III. *Execution replay for intrusion analysis*. PhD thesis, University of Michigan, 2006.

Kevin Elphinstone and Gernot Heiser. From L3 to seL4 – what have we learnt in 20 years of L4 microkernels? In *SOSP*, pages 133–150, Farmington, PA, USA, Nov 2013.

Bryan Ford. Plugging side-channel leaks with timing information flow control. In *4th HotCloud*, pages 1–5, Boston, MA, 2012.

Richard Gay, Heiko Mantel, and Henning Sudbrock. Empirical bandwidth analysis of interrupt-related covert channels. In *2nd QASA*, London, Sep 2013.

Michael Godfrey. On the prevention of cache-based side-channel attacks in a cloud environment. Master's thesis, Queen's University, Ontario, Canada, Sep 2013.

Wei-Ming Hu. Reducing timing channels with fuzzy time. In *1991 Comp. Soc. Symp. Research Security & Privacy*, pages 8–20, 1991.

Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symp. Security & Privacy*, pages 191–205, San Francisco, CA, May 2013.

IAD. *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*. Information Assurance Directorate, Jun 2007. Version 1.03. http://www.niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03/.

Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. CRUISE: Cache replacement and utility-aware scheduling. In *17th ASPLOS*, pages 249–260, London, England, UK, 2012. URL http://doi.acm.org/10.1145/2150976.2151003.

Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security*, pages 189–204, Bellevue, WA, USA, Aug 2012.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, Feb 2014.

Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *14th CCS*, pages 286–296, Alexandria, Virginia, USA, 2007.

Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *24th CAV*, pages 564–580, 2012.

Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *3rd RTAS*, Montreal, Canada, Jun 1997.

Heiko Mantel and Henning Sudbrock. Comparing countermeasures against interrupt-related covert channels in an information-theoretic framework. In *20th CSF*, pages 326–340, 2007.

Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *ISCA*, pages 118–129, Portland, Oregon, USA, Jun 2012. URL http://doi.acm.org/10.1145/2366231.2337173.

Nagendra Modadugu and Eric Rescorla. The design and implementation of datagram TLS. In *NDSS*, 2004.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symp. Security & Privacy*, pages 415–429, San Francisco, CA, May 2013.

NSA. *Information Assurance Guidance for Systems Based on a Security Real-Time Operating System*. National Security Agency, Dec 2005. SSE-100-1, http://www.nsa.gov/ia/_files/sse-100-1.pdf.

NSA. TEMPEST: A signal problem. *Cryptologic Spectrum*, 2 (3), 1972. Available at: http://www.nsa.gov/public_info/_files/cryptologic_spectrum/tempest.pdf.

Chris Owen, Duncan Grove, Tristan Newby, Alex Murray, Chris North, and Michael Pope. PRISM: Program replication and integration for seamless MILS. In *IEEE Symp. Security & Privacy*, pages 281–296, May 2011.

Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *16th CCS*, pages 199–212, Chicago, IL, USA, 2009.

Claude E. Shannon. A mathematical theory of communication. *The Bell Syst. Techn. J.*, 1948. Reprinted in SIGMOBILE Mobile Computing and Communications Review, 5(1):3–55, 2001.

Geoffrey Smith. On the foundations of quantitative information flow. In *12th FOSSACS*, pages 288–302, York, UK, 2009.

Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *18th ESORICS*, pages 718–735, Egham, UK, Sep 2013.

David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared L2 caches on multicore systems in software. In *3rd WS Interaction between Operat. Syst. & Comp. Arch.*, San Diego, CA, USA, Jun 2007.

NIST. *Common Criteria for IT Security Evaluation*. US National Institute of Standards, 1999. ISO Standard 15408. http://csrc.nist.gov/cc/.

John C. Wray. An analysis of covert timing channels. In *1991 Comp. Soc. Symp. Research Security & Privacy*, pages 2–7, May 1991.

Yaming Yu. Squeezing the Arimoto-Blahut algorithm for faster convergence. *Trans. Inform. Theory*, 56(7):3149–3157, 2010.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *18th CCS*, pages 563–574, Chicago, IL, USA, 2011.

Yinqian Zhang and Michael K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th CCS*, pages 305–316, Raleigh, NC, USA, 2012.