Leakage in Trustworthy Systems

David Cock

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Engineering

Faculty of Engineering

August 2014

Originality Statement

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed .....

Date .....

Copyright Statement

Authenticity Statement

'I certify that the Library deposit digital copy is a direct equivalent of the final officially approved version of my thesis. No emendation of content has occurred and if there are any minor variations in formatting, they are the result of the conversion to digital format.'

Signed .....

Date .....

Abstract

This dissertation presents a survey of the theoretical and practical techniques necessary to provably eliminate side-channel leakage through known mechanisms in component-based secure systems.

We cover the state of the art in leakage measures, including both Shannon and min entropy, concluding that Shannon entropy models the observed behaviour of our example systems closely, and can be used to give a safe bound on vulnerability in practical scenarios.

We comprehensively analyse several channel-mitigation strategies: cache colouring and instruction-based scheduling, showing that effectiveness and ease of implementation depend strongly on subtle hardware features. We also demonstrate that real-time scheduling can be employed to effectively mitigate remote channels at minimal cost.

Finally, we demonstrate that we can reason formally (and mechanically) about probabilistic non-functional properties, by formalising the probabilistic language pGCL in the Isabelle/HOL theorem prover, and using it to verify an implementation of lattice scheduling, a well-known cache-channel countermeasure. We prove that a correspondence exists between standard vulnerability bounds, in a channel-centric view, and the refinement lattice on programs in pGCL, used to model a guessing attack on a vulnerable system—a process-centric view.

For Fiona, Lillian and Frederick; and my parents.

Contents

List of Figures

List of Tables

List of Publications

- Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, Portland, OR, USA, September 2006. ACM. doi:10.1145/1159842.1159850

- David Cock. Bitfields and tagged unions in C: Verification through automatic generation. In Bernhard Beckert and Gerwin Klein, editors, *Proceedings of the 5th International Verification Workshop*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, August 2008

- David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Mu noz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 167–182, Montreal, Canada, August 2008. Springer. doi:10.1007/978-3-540-71067-7_16

- Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515, Munich, Germany, August 2009. Springer. doi:10.1007/978-3-642-03359-9_34

- 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 *Proceedings of the*

22nd ACM Symposium on Operating Systems Principles, pages 207–220, Big Sky, MT, USA, October 2009. ACM. doi:10.1145/1629575.1629596

- Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, June 2010. doi:10.1145/1743546.1743574

- David Cock. Lyrebird – assigning meanings to machines. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Proceedings of the 5th Systems Software Verification*, pages 1–9, Vancouver, Canada, October 2010. USENIX

- David Cock. Exploitation as an inference problem. In *Proceedings of the 4th ACM Workshop on Artificial Intelligence and Security*, pages 105–106, Chicago, IL, USA, October 2011. ACM. doi:10.1145/2046684.2046702

- David Cock. Verifying probabilistic correctness in Isabelle with pGCL. In *Proceedings of the 7th Systems Software Verification*, pages 1–10, Sydney, Australia, November 2012. Electronic Proceedings in Theoretical Computer Science. doi:10.4204/EPTCS.102.15

- David Cock. Practical probability: Applying pGCL to lattice scheduling. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, pages 1–16, Rennes, France, July 2013. Springer. doi:10.1007/978-3-642-39634-2_23

- David Cock. From probabilistic operational semantics to information theory; side channels in pGCL with isabelle. In *Proceedings of the 5th International Conference on Interactive Theorem Proving*, pages 1–15, Vienna, Austria, July 2014. Springer. doi:10.1007/978-3-319-08970-6_12

- David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile; an empirical study of timing channels on sel4. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 1–12, Scottsdale, USA, November 2014. ACM. doi:10.1145/2660267.2660294. (to appear)

1 Introduction

In this work we investigate the problem of side and covert channels in component-based secure systems. We approach the problem from two directions, *implementation* and *verification*, and this dissertation is therefore divided into three parts: Chapter 2 sets out the problem of information leakage in detail, surveys historical work, and establishes both a threat model: the guessing attack, and measures of vulnerability: Shannon capacity and min leakage; Chapter 3 covers the implementation and evaluation of practical countermeasures and the thorough empirical analysis of real channels; Finally, Chapter 4 through Chapter 6 present our contribution to verification, demonstrating that we can extend existing large-scale refinement proofs to tackle the sorts of probabilistic properties that arise in the study of side and covert channels.

This work is an offshoot of the L4.verified project [Klein et al., 2009, 2014], that established the functional correctness of the seL4 microkernel [Derrin et al., 2006; Elkaduwe et al., 2008], with a fully machine-checked proof. In addition to contributing generally to the study of systems-level approaches to information leakage, we lay the groundwork for the rigorous treatment of these channels in seL4. Our experimental work is thus mostly carried out on this system, although our results are more widely applicable. We also present methods to incorporate reasoning about highly hardware-specific, and often probabilistic, behaviour that are compatible with our refinement-driven verification methodology.

The two parts are presented independently, as far as possible. A reader who is only interested in the practical measurement and mitigation of channels can get a complete picture of the relevant work from Chapter 2 and Chapter 3, and can safely skip the remainder. Likewise, a reader interested only in verification could safely skip Chapter 3, and proceed directly to Chapter 4.

The practical approach provides context for the verification approach, but the two nevertheless stand alone.

The contents of the chapters are as follows:

- Chapter 2 presents the foundational results that motivate both the implementation and verification parts. In it, we introduce the specific problem of timing channels, and our threat model: adaptive *guessing attacks* that exploit the extrinsic leakage of a system. We survey the literature on leakage models, settling on *Shannon capacity* as our usual measure of choice for attacks involving large numbers of guesses, while *min leakage* provides tighter security bounds for smaller numbers. We contribute a proof of the maximum divergence of one-guess vulnerability given entropy, justifying that Shannon entropy *can* be used to give safe (if somewhat pessimistic) bounds on one-guess vulnerability.

- In Chapter 3, we take the groundwork of Chapter 2, and apply it practically. We demonstrate that existing techniques (cache colouring and instruction-based scheduling) can be implemented efficiently in seL4, to mitigate local channels with varying degrees of success. We also propose a novel mechanism (scheduled delivery) to tackle remote channels. In order to evaluate these countermeasures, we thoroughly analyse several real channels (cache contention, bus contention, and the lucky thirteen attack on DTLS), and demonstrate that these channels cannot be fully understood without careful empirical analysis—undocumented hardware behaviour has a dramatic effect on leakage.

- In Chapter 4 we present the formal machinery to verify probabilistic security properties on realistic systems. We develop our formalisation of pGCL—a language that rigorously fuses probability with nondeterminism and classical refinement. We demonstrate the effectiveness of our proof mechanisation in Isabelle/HOL, the same system used for L4.verified.

- We take this formalisation and, in Chapter 5, attack a large example: the verification of a well known countermeasure against cache channels—lattice scheduling. We demonstrate that, in addition to showing a classical trace-based noninfluence property, we can prove probabilistic asymptotic fairness, and incorporate the entire L4.verified proof stack. This

result demonstrates that appropriately stated probabilistic results can be composed with classical refinement, and integrated with large existing proof artefacts.

- Finally, Chapter 6 links us back to the foundational theoretical results of Chapter 2, by showing that we can recover standard information-theoretic models from a concrete pGCL model of a guessing attack, with a fully machine-checked proof. We demonstrate the link between a program-oriented and a channel-oriented view of leakage. This is reinforced by the connection we derive between the lattice of channel bounds, and the refinement lattice in pGCL. The focus in this final chapter on the *kind* of results can be attacked with the verification approach established in the preceding chapters.

2 Covert and Side Channels

This chapter expands on work first presented in the following paper:
David Cock. Exploitation as an inference problem. In *Proceedings of the 4th ACM Workshop on Artificial Intelligence and Security*, pages 105–106, Chicago, IL, USA, October 2011. ACM. doi:10.1145/2046684.2046702

We are concerned with the leakage of sensitive information from a system, for example a password or an encryption key, through unexpected channels—those that were not anticipated in the system's specification. Our particular focus is on predicting and rectifying such leaks in real systems software, without assuming the ability to modify it. We assume that such imperfections are inevitable, and consider what approaches we, as system implementors can take to efficiently rectify them.

We begin by surveying the historical literature, to place this work in context. Relevant contemporary work is summarised chapter-by-chapter. We next establish our threat model (the *guessing attack*), and show that, despite its problems, Shannon entropy can be used as a safe measure of vulnerability, with appropriate corrections. We motivate each step of our theoretical development by appealing to the features of a very simple side channel, due to a common optimisation in the implementation of the C `strcmp` routine.

The message of this chapter is that the guessing attack is a sound and broadly applicable threat model, and that Shannon entropy can be used to give a safe bound on vulnerability, allowing us to use standard results in information theory.

To make our case, we build a leakage measure from first principles, gradually adding complexity (moving from a uniform distribution of secrets in Section 2.3, to nonuniform distributions in Section 2.4, and finally imperfect observations in Section 2.6), justifying at each point that the model matches what we see in our example system (the `strcmp` channel of Section 2.2). We show that in capturing the observed behaviour of this system, we are lead naturally to Shannon entropy (and hence channel capacity) as a measure. We show that despite its known limitations, we can nevertheless construct a safe measure based on entropy, in Sections 2.5 & 2.7.

2.1 Background

Communication channels that bypass a system's information flow policy have historically been divided between *side channels* and *covert channels* [Wray, 1991]. The distinction is rather arbitrary, and mostly depends on the context in which the channel is encountered. We emphasise the commonality between the two, and focus on restricting the common, underlying mechanisms. We further argue that channels that exploit explicit system behaviour (even if not intended for communication) are effectively dealt with by existing techniques, and thus restrict our attention to channels that lie outside these mechanisms.

Previous attempts have been made to produce general-purpose systems which either limit or eliminate side and covert channels, motivated in particular by the US DoD Trusted Computer System Evaluation Criteria (the orange book standards), which suggested a maximum bandwidth of either 1b/s for general systems, with the ability to audit (detect) channels of more than 0.1b/s [DoD, §8.0, page 80].

Digital Equipment Corporation's VAX/VMM (virtual machine monitor) [Karger et al., 1991] was designed as a high-security general-purpose system, and was intended to be to be certified to TCSEC level A1 (the highest). The design was inspired by the earlier KVM/370 [Schaefer et al., 1977] project to retrofit a secure virtualisation layer to IBM's existing VM/370 mainframe operating system. While the project did not achieve its goals, it pioneered several mitigation techniques, namely fuzzy time [Hu, 1991, 1992b] and lattice scheduling [Hu, 1992a], both of which we analyse later.

While other work has focussed on producing provably secure hardware [Greve and Wilding, 2002], we are attempting to build secure general-purpose

systems on commodity hardware.

Side channels are those in which supposedly hidden information, for example an encryption key, is *accidentally* signalled on a public medium. Here, the concern is whether a carefully-designed, and trusted, system might reveal a secret through unexpected means, for example its power consumption [Kocher et al., 1999], execution time [Wray, 1991; Brumley and Boneh, 2003], or even (acoustic) noise output [Backes et al., 2010].

Interest in side channels arose in connection with cryptography and sensitive (particularly military) communications. The deliberate interception of radio signals, including inadvertent emanations from field telephones, dates back at least to 1915 [Cryptome, 2002], however in these cases no particular effort was made to hide the sensitive signal—these were not *protected* systems. Side channels in the form we usually consider: unanticipated emanations from a protected system, were recognised in cipher machines at least as early as the 1940s, as detailed in documents recently declassified by the United States' National Security Agency (NSA) [NSA, 1972].

An example from this document is the discovery that a particular encryption device (the 131-B2) produced visible spikes on the display of an oscilloscope on the opposite side of the room, and that by analysing these, the unencrypted plaintext could be recovered. This is a device that was designed to protect sensitive data (although only limited detail is available in the redacted public document), indeed specifically to allow it to be sent over unsecured radio channels, which was nevertheless compromised by additional, unintentional radiation. In this example, the secret (the plaintext) is correlated with a publicly observable variable (the radiation).

In this work, we consider *timing channels*, where the observed variable is the execution time of a program, or the relative arrival times of a series of messages. While the physical details are different, when viewed as abstract communication channels these examples are equivalent.

Covert channels are those in which information is *deliberately* leaked by a compromised insider, the "trojan horse". Interest in covert channels arose with the advent of multiprogrammed systems and utility computing [Lampson, 1973], where a third-party component (for example, a library routine) might be used to process sensitive data. In this scenario, the owner of the data (the

client) wants to be assured that the library routine cannot leak that data to an external accomplice, no matter how hard it tries. Interest in covert channels (and locally-exploitable side channels) is re-emerging with the adoption of cloud computing [Ristenpart et al., 2009], which is beginning to fulfil the promise of utility computing, but is naturally subject to the same risks.

It is important to note that the set of leakage mechanisms available to the Trojan are exactly the same as those used by a side channel. The only difference is intent. As it is being exploited deliberately, a covert channel may well be used more efficiently, relative to its theoretical capacity, than the equivalent side channel, but the two are otherwise equivalent. In particular, a bound on the ability of the channel to transfer information provides an upper bound on the bandwidth achievable in either case. Therefore, for an identified leakage mechanism, we focus on minimising this *channel capacity*, recognising that doing so prevents its exploitation in either case.

Storage channels form part of another, orthogonal, scheme for classifying leakage channels [Lampson, 1973; Lipner, 1975; Kemmerer, 1983, 2002]. Here, the distinction is between channels that exploit the ability of (for example) a trojan horse to store information in an unexpected location, to be read back later by its accomplice. These channels range from the relatively straightforward (e.g. a hidden CPU register [Sibert et al., 1995]), through the subtle (e.g. disk arm positioning [Karger and Wray, 1991]), to the fiendish (e.g. modulating processor temperature by loading the CPU [Murdoch, 2006]). A storage channel may be exploited as either a covert or a side channel.

Timing channels are the complement of storage channels, in the classical taxonomy. A timing channel carries information in the relative timing of events, even if the values delivered are constant. This division assumes a model of what the receiver sees: it observes the system at a series of instants, at each of which it may inspect that part of the state that is visible to it. If the sender can affect the state visible to the receiver at any instant, this is a storage channel, while if it can affect the timing of these instants (but not necessarily any observable value) we have a timing channel.

Wray [1991] argued convincingly that this distinction is arbitrary, giving examples of channels that can be classified as either storage or timing channels, depending on how they are exploited. Both the disk-arm and processor-

temperature channels are examples. While both "store" a real value (the arm's current position relative to the spindle, and the processor's current temperature), the most practical (but not necessarily the only) way of measuring both is to use time: for the disk arm, as noted by Wray, measuring the time to seek to a known block reveals the arm's starting position, while measuring either execution speed (for a processor employing thermal throttling), or clock skew [Murdoch, 2006] gives a rough estimate of temperature.

We instead classify channels in a way directly relevant to our work—by the techniques used to analyse them:

Functional leakage is our term for any leak demonstrated by the *functional specification* of the system. This definition is motivated by our experience with seL4 [Klein et al., 2009]. The seL4 microkernel is modelled as a transition system, with the kernel mapping the combined machine and kernel state on entry, to a modified state on exit. Any leak via kernel mechanisms (for example in the observed scheduling order or failure to clear memory or registers when switching between domains), is visible in this model. The absence of such channels has been formally established for Murray et al. [2013], with a fully mechanised proof. This is roughly equivalent to a storage channel.

There may, of course, be channels that exist in the implementation which are not captured by the specification, for example a hidden register that we have failed to model. We would still consider this to be functional leakage, and treat this as a specification bug (either of the kernel or, more likely, of the hardware). Work is ongoing to discover and fix such bugs.

Once we consider the specification to be both complete and correct, we define any leakage that it implies to be *intrinsic* i.e. the leakage that is a unavoidable, in correct operation. If the system is intended to be noninterference-secure [Goguen and Meseguer, 1982] (a very strong notion of isolation), then this intrinsic leakage must be zero. We rate an implementation by how close it comes to the intrinsic leakage of its specification: the additional leakage due to implementation artefacts should be as close to zero as possible.

Non-functional leakage is the complement of functional leakage—anything not expressed by the specification. An example, in seL4, is timing: the specification states only what changes the kernel makes to the state, but says nothing about how long it takes to make them. This captures, by definition,

```
1  int
2  strcmp (const char *p1, const char *p2) {
3    register const unsigned char *s1 =
4      (const unsigned char *) p1;
5    register const unsigned char *s2 =
6      (const unsigned char *) p2;
7    unsigned reg_char c1, c2;
8
9    do {
10     c1 = (unsigned char) *s1++;
11     c2 = (unsigned char) *s2++;
12     if (c1 == '\0') return c1 - c2;
13   } while (c1 == c2);
14
15   return c1 - c2;
16 }
```

Figure 2.1: `strcmp`, from GNU glibc 2.9 (edited for readability).

all remaining sources of leakage, but our classification is context dependent. Time *could* be incorporated into the specification, as could any number of physical variables (temperature, for example). There are two reasons to avoid this: First, the specification would be more complex, and we have no good reason to suppose that we could complete a formal proof equivalent to that for the functional specification (even if we had a specification of the hardware's behaviour); Second, the question, by its very nature, is open ended. The history of research into side channels in particular is one of the progressive discovery of more and more leaks not covered by existing models. There must therefore be a category for vulnerabilities that we haven't found yet.

Any classification of channels is likely to be to some extent arbitrary, and moreover to reflect the bias of the classifier. We settle on the functional/non-functional distinction as it matches our verification approach, although we shortly refine it into the notions of *intrinsic* and *extrinsic* leakage, reflecting the distinction between leakage implied by a specification, and that added by an implementation.

2.2 The `strcmp` Channel

To illustrate the problem, we analyse a timing channel in common implementations of the C library's `strcmp` (string compare) routine. We use this example to illustrate our notions of vulnerability and channel capacity, and also to motivate our threat model: the *guessing attack*.

Figure 2.1 is typical, from GNU glibc version 2.9 (edited for readability). The comparison is performed by the loop at lines 9–13. This steps through the strings in parallel (lines 10 & 11), until either the first ends (line 12, strings are null terminated), or a mismatch is found (line 13). If the second string is shorter, the loop terminates when its null terminator fails to match the corresponding character in the first. The number of iterations, and thus execution time, increases with the number of successful matches: the *common prefix length* of the two strings.

An appropriate specification, expressed in Higher-Order Logic[1] (HOL), is as a function from a pair of strings to a truth value, abstracting from the details of the implementation:

$$\begin{aligned} &\text{strcmp} :: \mathit{string} \times \mathit{string} \to \mathit{bool} \\ &\text{strcmp}\ (s_1, s_2) \equiv (s_1 = s_2) \end{aligned} \tag{2.1}$$

If an attacker observes the execution time sufficiently closely, then he or she learns much more than this simple specification indicates. The implementation leaks more than the specification allows (the *intrinsic* leakage)—the non-functional leakage in this example is substantial. Defining and measuring this occupies the remainder of the chapter.

A Guessing Attack on `strcmp`

Imagine that our attacker is trying to guess a password by repeatedly invoking a password checker that uses `strcmp`. Usually, a guessing attack is expensive: exponential in the length of the password. We will see, however, that with the addition of side-channel leakage, this becomes linear.

We begin by calculating the vulnerability given only the intrinsic leakage implied by Equation 2.1.

[1] HOL is a formal logic for mathematical specification and proof. We make heavy use of HOL in later chapters, where we mechanically verify leakage properties. For the moment, it is sufficient to consider this specification as an implementation in a functional language, such as Haskell or ML.

2.3 Leakage with a Uniform Prior

Assume, for now, that all secrets are equally likely (perhaps generated uniformly at random), and that the attacker knows the precise execution time. This is the simplest case, and we relax both assumptions shortly.

Suppose that the attacker supplies a string of its choice (a guess), which the system compares against its secret, answering yes (for a match) or no (for a mismatch). Let the secret consist of n characters, drawn from an alphabet of m symbols, giving m^n possibilities. Assume that this is known to the attacker. If the secret is chosen uniformly, the attacker has no reason to suspect that one secret is more likely than another, and may as well guess in any order. After each guess, either the secret is found (the system answered yes), or the attacker eliminates a possibility.

As all remaining secrets are equally likely, the chance of guessing correctly at each step is one over the number of possibilities remaining: The chance that the *next* guess is correct, having made k incorrect guesses, is

$$V_0(k) = \frac{1}{m^n - k} \tag{2.2}$$

The subscript 0 indicates that this is the probability of compromise given no additional information. This gives an obvious, fundamental measure of vulnerability:

Definition 1 (One-Guess Vulnerability)**:** For a system subject to guessing, the one-guess vulnerability, V_0, is the chance that an optimal (computationally unbounded) attacker can compromise it in a single attempt, given no extra information.

This has the benefit of being succinct, easily stated, and obviously applicable. The downside, as we'll see, is that it is generally impractical to calculate. We thus focus on efficiently approximating, or bounding, V_0.

Any definition of vulnerability is, implicitly or explicitly, made with reference to a *threat model*—ours is the *guessing attack*. This may appear limiting, but it's not. The class of systems vulnerable to guessing is very broad: We can include any system that authenticates using a secret. We can also include most applications of cryptography. A protocol for authenticity (e.g. a digital signature) is subject to guessing in exactly the same manner as a password (keep guessing until the signature matches). A guessing attack against a confidentiality protocol (e.g. message encryption) is also simple, if the ciphertext

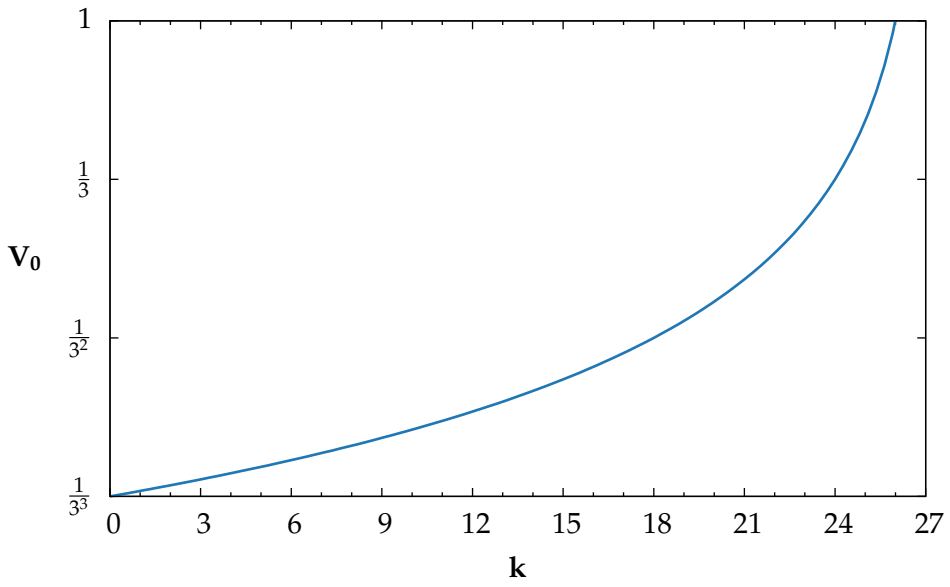

Figure 2.2: Vulnerability (V_0) vs. number of guesses (k), for `strcmp` with no side channel, where $m = 3$, $n = 3$.

is known. The attacker can conduct an *offline* guessing attack (in contrast to the *online* attack against the password checker), by trying keys one at a time, until the message is decrypted. The attacker has a nonzero (though vanishingly small) chance of guessing correctly on the first attempt. We do need to assume that the attacker is able to recognise valid plaintext. Thus, one-time-pad encryption is *not* vulnerable to guessing [Shannon, 1948].

Returning to our example, Figure 2.2 plots V_0 given only intrinsic leakage, for $m = 3$ and $n = 3$: secrets of length 3 drawn from an alphabet of 3 symbols. We see that vulnerability initially rises very slowly (the vertical scale is logarithmic, to better discriminate smaller values), and reaches 1 (compromise) after 26 guesses. At this point, the attacker knows that there is only one possible secret, of the initial $3^3 = 27$.

How does adding a side channel affect V_0? That is, how does the leakage added by the implementation of Figure 2.1 compare to the intrinsic leakage of Figure 2.2? Clearly, $V_0(0)$ will not change, as leakage is only observed after guessing. We only deviate after the first guess, and if the attacker is optimal, vulnerability only increases (on average) given more information. Note that the curves must meet again after $m^n - 1$ guesses, as with only one secret

remaining in either case, the attacker will guess correctly with probability 1.

How does the curve behave between $k = 0$ and $k = m^n - 1$? Recall that the we leak the execution time of the `strcmp` routine, which exposes the common prefix length of the guess and the secret. Take the place of the attacker. Knowing the prefix length, we attack the positions individually: To find the first, we generate m strings of length n, differing in the first position, and pass these to the system. $m - 1$ of these will fail to match in the first position, and thus the loop will execute once (as it is post-checked). One, however, will match, and we will see *at least* 2 iterations. The string giving the longest runtime thus has the correct character in the first position (and possibly more). Repeat for the remaining characters: Once i positions are known, we find position i (strings are zero indexed), by setting $0 \dots i-1$ to their known values, and varying i, looking again for the greatest response time.

The Vulnerability Model

Given this strategy, we can calculate $V_0(k)$. Every incorrect guess now eliminates not just one, but a set of secrets: when $0 \dots i-1$ are known (thus we're guessing position i), if we see only $i+1$ iterations, our guess for i was wrong. This eliminates not only our guess, but any that match it in its first i positions (mismatches in $0 \dots i-1$ have already been ruled out, by induction). If we see $i+2$ or more iterations however, we have the correct character in position i and eliminate all other possibilities.

With leakage, plotting V_0 against time is much more complicated. Figure 2.3 shows all possible traces for $m = 3, n = 3$ (in red), with the number of guesses along the horizontal axis, and the current value of V_0 on the vertical. A single trace begins at $\{0, 1/27\}$, at the bottom-left, and on each guess moves one step right and some number upward (as it may guess several positions at once). The shaded circles show the likelihood of reaching a state, normalised by column. Thus, a circle at $\{k, v\}$ gives $P(v|k)$: the conditional probability of seeing vulnerability v after k guesses. The vertical axis is logarithmic, as is the vertex shading. The no-leakage trace is included (in blue), for comparison.

The increase in vulnerability is obvious: all traces reach $V_0 = 1$ by the 6^{th} guess, whereas the original attack took until the 26^{th}. However, this only compares the worst-case for each attack. How do they compare on average?

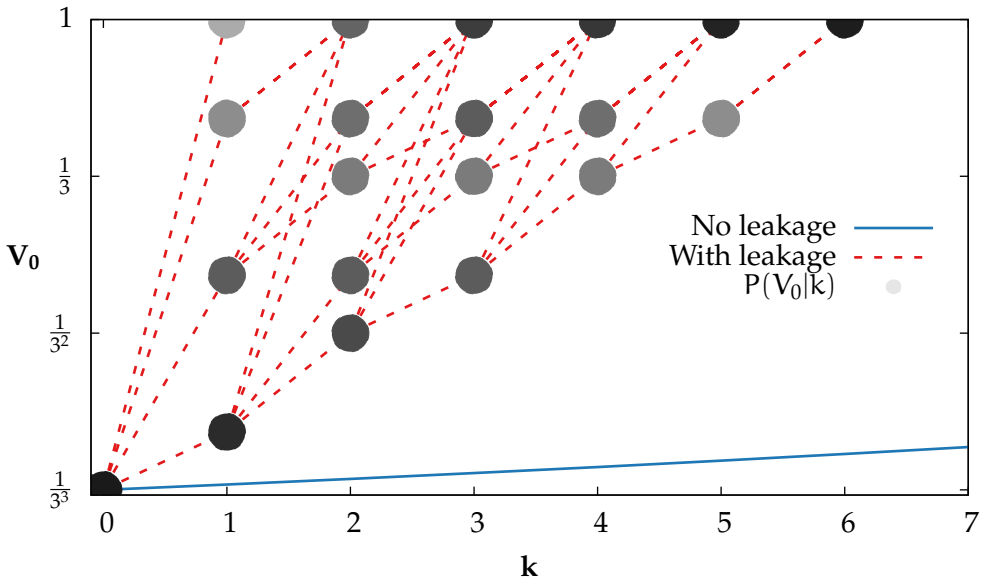

Figure 2.3: `strcmp` attack traces with leakage for $m = 3$, $n = 3$, showing vulnerability (V_0) vs. number of guesses (k), and $\log_2 P(V_0|k)$ at each vertex. Includes the no-leakage trace in blue for comparison

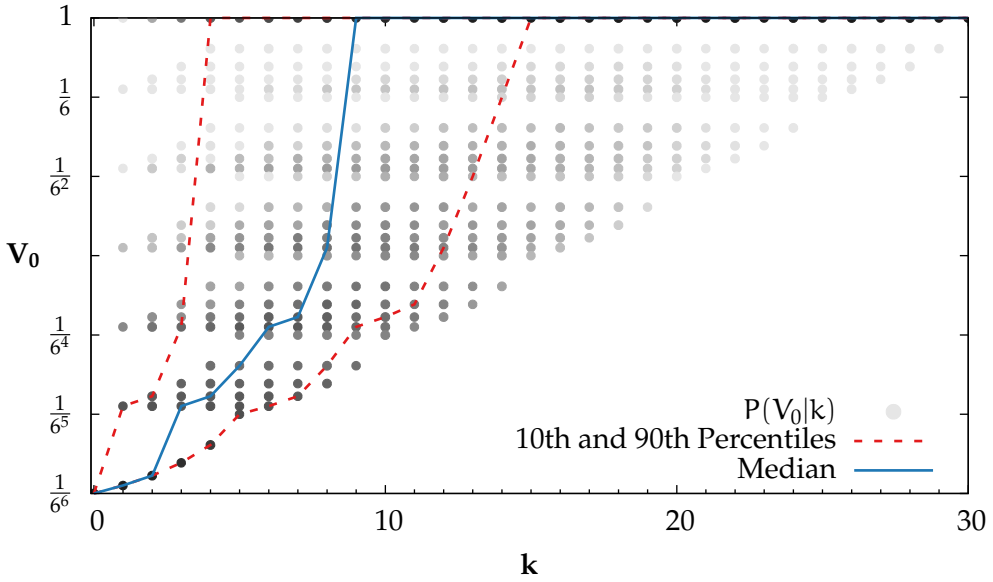

Figure 2.4: `strcmp` attack traces for $m = 6$, $n = 6$, showing vulnerability (V_0) vs. number of guesses (k), and $\log_2 P(V_0|k)$ at each vertex. Includes the 10th, 50th and 90th percentiles for $P(V_0|k)$.

Increasing to $m = 6$ and $n = 6$ gives Figure 2.4. Here we drop the traces, leaving only vertices, for clarity. They follow the same pattern as in Figure 2.3. We also drop the no-leakage curve, as it never rises high enough to be visible: it now takes until $k = 6^6 - 1 = 46,655$ to reach $V_0 = 1$ without leakage, with all leakage traces terminating by $k = 30$.

Including the median (blue) and the 10^{th} and 90^{th} percentiles gives a feeling for the distribution of traces: 90% lie above the 10^{th} percentile (the rightmost red curve) at any given k. Note that a trace may cross a percentile, and that the percentile itself does not (necessarily) represent a trace. We see that 90% of all traces find the secret in no more than 15 guesses, with half taking less than 9. The distribution is thus clustered to the left, with a long right-hand tail. The worst-case length (the defender's best case) is thus an overestimate and a poor guide to likely vulnerability. The best-case is likewise a dramatic underestimate: 1 guess.

How can we quantify this increase in vulnerability? Figure 2.4 contains all the information we need, certainly, but we want a summary measure. There are two reasons: Firstly, the full distribution is a clumsy way to describe the system; Secondly, and more importantly, we can't generate these figures for anything more than toy examples.

As a measure, V_0 isn't good enough on its own as it only refers to the current state and doesn't take into account any future leakage. Consider Figure 2.4 again, and imagine that we are on a trace similar to the (blue) median. Suppose we reach $V_0 = 1/6^4$, after 6 guesses. Taking only V_0 into account (given our uniform search space), there are $6^4 = 1296$ equally-likely secrets which, given no extra information, would take on average 648 further guesses. From the figure however, we expect to get the secret correct in only 3 additional guesses! Using only V_0, we would assign this outcome a probability of only 3/1296. We thus need to take future leakage into account.

We must note that we are making a qualitative judgement regarding what we want in a security measure. Our strawman (the expected number of guesses remaining), exemplifies one particular style of measure: *expected time to compromise*. This example, where we ignore future interaction with the system, and let the attacker perform, say, an offline brute-force attack, is known as the *guessing entropy*[Massey, 1994], and is defined as:

$$\sum_i iP(x_i)$$

where the x_i are the list of possible secrets, arranged in order of decreasing likelihood.

These measures are widely used, and intuitively reasonable, in the field of cryptography, where the primary concern is often with a well-resourced attacker performing an offline cryptanalysis (essentially a sophisticated guessing attack) against captured ciphertext. Here, the argument is in terms of the cost of the attack, relative to the value of the secret, and is quantified by the *work factor* of a cryptosystem.

The difficulty in applying such a measure here is that it depends critically on the assumption of uniformity. For a cryptographic key or a nonce[2], a great deal of effort is made to ensure the uniformity and unpredictability of the secret, and thus this assumption holds. In the case of a password, or the leakage of sensitive control-flow information, this assumption *does not* hold. Passwords in particular are well recognised, ([Dell'Amico et al., 2010; Malone and Maher, 2012]), to be non-uniformly distributed, with some astonishingly common. In this case, knowing the expected time to compromise gives us little confidence in the security of the system.

In the extreme, if we have one secret of probability 0.99, and 10^{100} secrets of probability 10^{-102}, the expected time to compromise is roughly $0.99 \times 1 + 0.01 \times 10^{100}/2 \approx 10^{98}$, an enormous number. The probability of compromise *in a single guess* however, is 99%! We cannot assume uniformity on secrets, and we must judge a system like this to be insecure. We therefore use *expected chance of compromise*, rather than expected time to compromise. The one-guess vulnerability is a simple example of such a measure. The advantage is that we avoid the above problem; The disadvantage is these measures are often difficult to calculate, as already established for `strcmp`.

We now show that we can, with appropriate care, *bound* chance of compromise, given only an average security measure. This allows us to combine the simplicity of time-to-compromise, with the rigour of chance-of-compromise.

This problem, the disconnect between chance-of-compromise and time-to-compromise, was recognised by Smith [2009], specifically between the Shannon entropy and the min entropy. The Shannon entropy is a summary measure that we will introduce shortly, while the min entropy is a worst-case measure, and is equivalent to V_0. There has been a great deal of interest in

[2]A nonce is a randomly-selected single-use token. Its most important property is that it should be unpredictable to an attacker, and is thus usually selected at random.

min entropy as a vulnerability measure [Espinoza and Smith, 2013], leading to the recent work of Alvim et al. [2012], showing that when generalised (to *gain functions*), it subsumes most other measures. Our contribution is to show that while the worst case is indeed as established, this only occurs for pathological distributions, and that if care is taken, Shannon entropy *can* be used safely.

Multiple Guesses

To incorporate leakage across multiple steps, we extend our measure from one guess (and no observations) to many:

Definition 2 (k-Guess Vulnerability)**:** The system is compromised once $V_0 = 1$, i.e. the attacker knows the secret, and is certain to guess correctly on its next attempt. The k-guess vulnerability, or V_k, is the probability that the system will be compromised given *at most* k guesses. That is, it is the *probability* that $V_0 = 1$ after k guesses (and $k - 1$ observations, for $0 < k$).

For any k, V_{k+1} is at least as high as V_k: An extra guess gives the attacker another attempt, which succeeds with non-negative probability. We apply the measure by setting k appropriately: If we're analysing a password checker that allows only three tries before disabling logins, then setting k to 3 gives the probability of compromise. In a cryptographic setting, k might indicate the attacker's estimated computational power: If we're confident that the attacker can't process more that, say, 10^9 keys in a reasonable timeframe then V_{10^9} again gives the overall chance of compromise.

Figure 2.5 shows vulnerability against time (number of guesses) for the $m = 6, n = 6$ example. This plot is generated from the same data as Figure 2.4. V_k is plotted in orange, and rises to one by about the 10^{th} guess. Let $V_0(k)$ be the expected vulnerability after k guesses, *assuming that the secret hasn't been guessed yet*. From this, we can reconstruct V_k as:

$$V_k = V_0(0) + \big(1 - V_0(0)\big) \times \big[V_0(1) + \big(1 - V_0(1)\big) \times \dots\big]$$

That is, the probability that we guess by the k^{th} attempt is the probability that we guess on the first attempt, plus the probability that we *don't* multiplied by the probability that we guess correctly somewhere between attempt 1 and k, *given that we guessed wrong at* 0.

We can thus translate between the two measures, and will calculate using $V_0(k)$ from now on. We see the benefit by considering the remaining curves

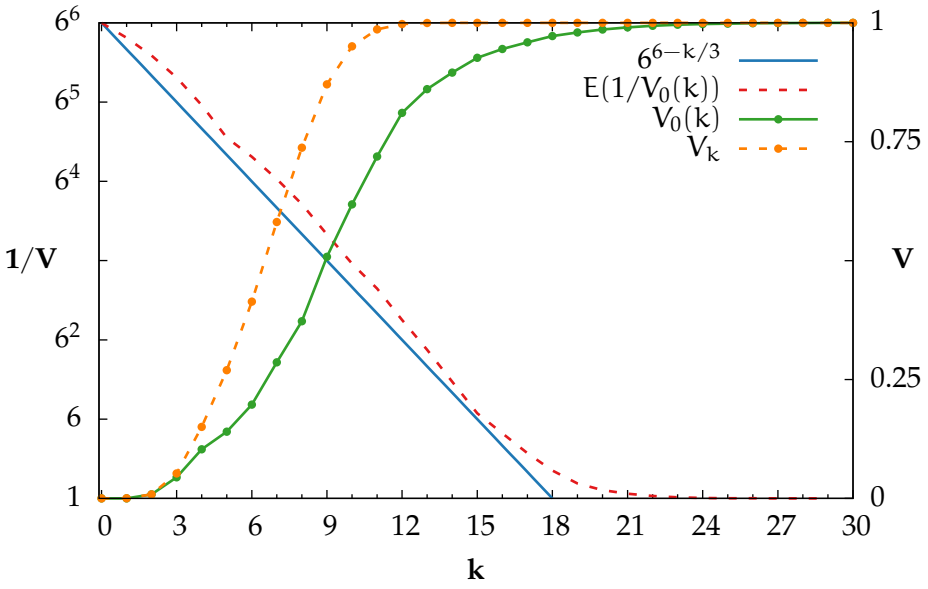

Figure 2.5: k-guess vulnerability (V_k) vs. k, for $m = 6, n = 6$, together with expected one-guess vulnerability ($V_0(k)$). Also shown are the expected uncertainty set size ($E(1/V_0(k))$), and the linear trend assuming 1 position is found every 3 guesses, showing roughly log-linear behaviour.

in Figure 2.4. $V_0(k)$ is plotted in green, and as we see it always lies below V_k: This is as we expect, as it considers only the probability of terminating on guess k, and ignores the probability of terminating earlier.

Recall from Equation 2.2 that V_0, the one-guess vulnerability, is inversely proportional to the number of possible secrets remaining, as they are uniformly distributed. Thus, the reciprocal of V_0 in any state is the size of this *uncertainty set*. We plot the expectation of this value in red, that is, the size of the *expected uncertainty set*. This shows how many secrets remain, on average, after k guesses. Rather than directly measuring *security*, as V_0 does, this quantifies the *amount of work remaining* for the attacker—a time-to-compromise measure. This curve shows us something interesting: Up to 15 guesses (at which point the system is almost totally compromised), the logarithm of the uncertainty-set size decreases more-or-less linearly (note that the left-hand vertical scale is logarithmic).

The blue line gives a lower bound on this size, and hence an upper bound on leakage. Here we see that the size of the uncertainty set decreases by a

factor of 6, approximately every 3 guesses. This is equivalent to saying that the attacker guesses one position per 3 attempts, which is what we expect given that there are 6 possibilities per position. This tells us that vulnerability is approximately *log-linear*: the ratio of the set sizes remains roughly constant for a given interval. This suggests a *logarithmic* definition of leakage:

Definition 3 (Uncertainty Set)**:** For a leakage profile in which the set of possible secrets is always uniformly likely, let S_u be this *uncertainty set*: the set of secrets of nonzero probability. The uncertainty set measure is then the size of this set:

$$V_u = \|S_u\| \tag{2.3}$$

Leakage is the *ratio* of vulnerability between two states:

$$L_u = \frac{\|S_u\|}{\|S'_u\|} \tag{2.4}$$

Note that, if the probability is uniform, $V_0 = 1/V_u$, and the two measures are equivalent.

The power of this definition is that we have a single summary measure both for vulnerability (V_u), and for leakage (L_u), that in our example allows a fair approximation by a simple linear model. We will see shortly that this measure also remains useful when we abandon first the assumption of uniformity (Section 2.4), and then of complete knowledge (Section 2.6). Generalising leads us eventually to Shannon entropy (Equation 2.10).

Limitations

Before we continue, we need to recognise the limitations of this model. In deriving this measure from V_u, we have thrown away information. Recall that we started by noting that $1/V_0 = V_u$, thanks to uniformity. From this, we hypothesised that $1/\mathrm{E}(V_u)$ might be a useful approximation to $V_n = \mathrm{E}(1/V_u)$. Note that, in general, $\mathrm{E}(1/X) \neq 1/\mathrm{E}(X)$: the expectation does not commute with the reciprocal. The two measures are plotted together in Figure 2.6. We see that, while our estimated vulnerability curve (red) has roughly the same shape as the true curve (green), it is offset to the right by about 12 guesses. That is, it predicts that a given degree of compromise will occur 12 guesses *later* than it really does, and thus underestimates vulnerability.

Why is this such an underestimate? We'd like to take advantage of the linear relationship we discovered in Figure 2.5, which would give us a single

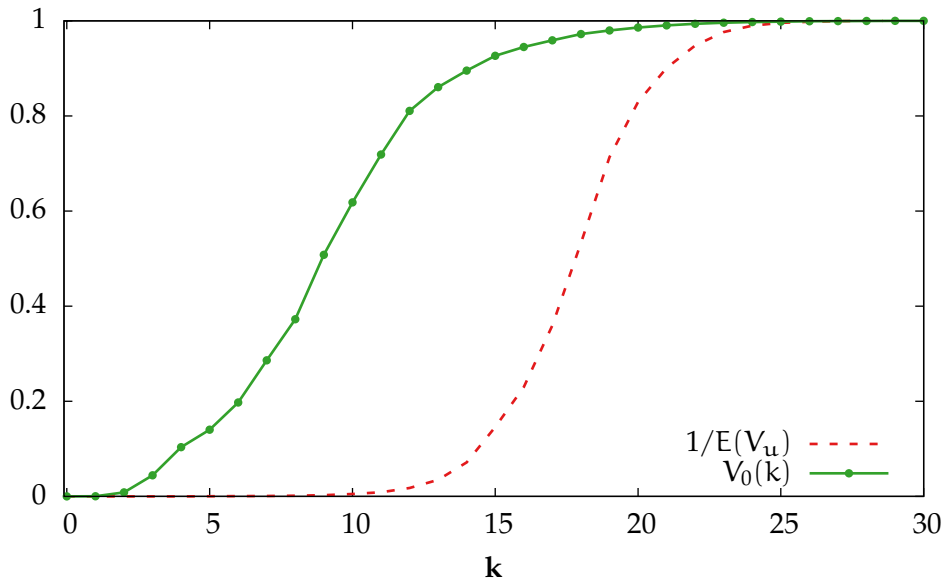

Figure 2.6: Expected vulnerability, and vulnerability estimated from expected uncertainty set size, showing the similar shape of the curves.

number (the slope), as the constant leakage rate of the system. The fact that the two curves have a similar shape is suggestive, and if we were to shift our estimate left by 12 guesses, we would have a reasonable approximation to the true vulnerability (for this system, at least). Is there any way in which this is theoretically justifiable? Luckily enough, there is.

As noted, a summary measure throws away information. In particular, there may be many distributions with a given $E(V_0)$, but for these, $1/E(1/V_0)$ will generally differ. Recall that in calculating $V_0(k)$, we are working with a distribution on the states of the system after k guesses. The only quantity that matters for the calculation of V_0 is the size of the uncertainty set, which ranges from 1 to m^n.

We can still bound the range of values that the true vulnerability, $E(V_0)$, may take. The question is: what is the largest possible value of $E(V_0)$ for any distribution for which $1/E(1/V_0))$ is known? The answer is relatively simple as, while the function $1/x$ is not linear, it is (anti-)monotonic.

Pessimistic Correction

Lemma 1 (Maximising the Reciprocal Expectation)**:** Let X be a random variable, ranging over the integers $[1, n]$. For any fixed y, over all distributions where $E(X) = y$, $E(1/X)$ is maximised when all probability mass is assigned either to 1 or to n.

Proof. See Appendix A □

Lemma 2 (Pessimistic Correction of Expected Uncertainty Set)**:** For a secret of length n, drawn uniformly from an alphabet of size m, the greatest possible expected vulnerability, $E(V_0)$, over all distributions where $E(1/V_0)$ is held constant, is:

$$1 - \frac{1}{m^n} - \frac{E(1/V_0)}{m^n} \tag{2.5}$$

Proof. First, we use the fact that $V_0 = 1/\|S_u\|$ to instead look for the maximal value of $E(1/\|S_u\|)$, where $E(\|S_u\|)$ is known. By Lemma 1, this must be attained by a distribution that assigns nonzero probabilities only to $\|S_u\| = 1$, and $\|S_u\| = m^n$. There is only one such distribution, satisfying the linear equations:

$$\begin{aligned} P(\|S_u\| = 1) + m^n P(\|S_u\| = m^n) &= E(\|S_u\|) \\ P(\|S_u\| = 1) + \quad P(\|S_u\| = m^n) &= 1 \end{aligned}$$

solving gives:

$$\begin{aligned} P(\|S_u\| = 1) &= \frac{m^n - E(\|S_u\|)}{m^n - 1} \\ P(\|S_u\| = m^n) &= \frac{E(\|S_u\|) - 1}{m^n - 1} \end{aligned}$$

whence,

$$\begin{aligned} E(1/\|S_u\|) &= P(\|S_u\| = 1) + \frac{P(\|S_u\| = m^n)}{m^n} \\ &= \frac{m^n - E(\|S_u\|)}{m^n - 1} + \frac{E(\|S_u\|) - 1}{m^n(m^n - 1)} \\ &= 1 - \frac{1}{m^n} - \frac{E(\|S_u\|)}{m^n} \end{aligned}$$

□

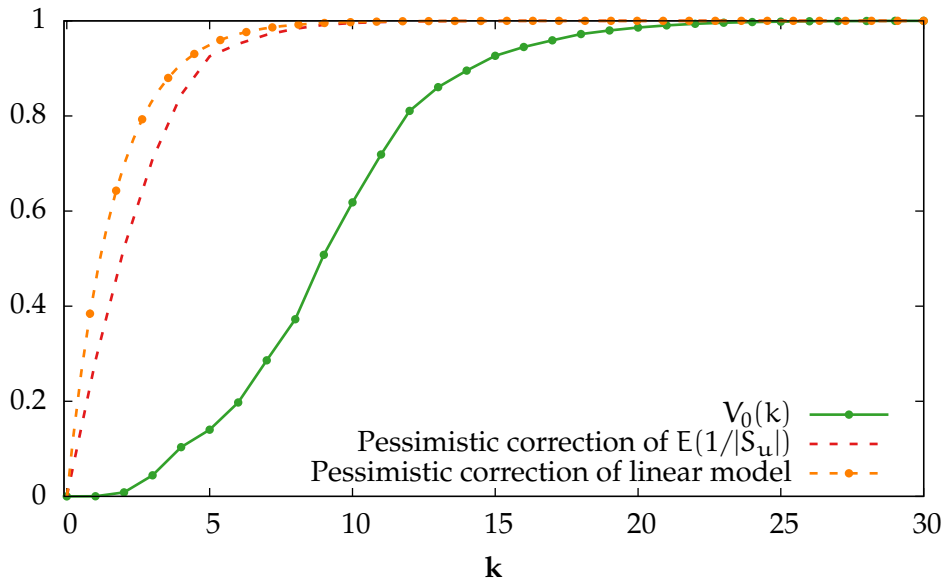

Figure 2.7: Pessimistic correction of the uncertainty-set measure.

We use Equation 2.5 to transform the expected uncertainty set size in Figure 2.5 to a safe upper bound on vulnerability, as plotted in Figure 2.7. We see that while our new bound is pessimistic, it is *safe*. Moreover, we can apply it to the linear model of Figure 2.5 (1 position every 3 guesses), to get a very similar, and also safe upper bound.

Summary We have established that in the case of a uniform distribution on secrets, leaking the common prefix length dramatically shortens a successful guessing attack against `strcmp` (Figure 2.2 vs. Figure 2.3). While the number of optimal attack traces is large, they are unevenly distributed, with short traces being most likely, and a long tail (Figure 2.4). While the expected vulnerability grows rapidly and non-linearly, we see that during most of the attack, the attacker's *uncertainty set* shrinks geometrically: $\log|S_u|$ is approximately linear, with a slope of $\log 6/3$, corresponding to our intuitive model of one position guessed on average, per 3 attempts.

From a log-linear model of leakage, we reconstruct a safe bound on expected one-guess vulnerability (V_0), by establishing how far the true set size can deviate from its expectation. We are thus able to use a model that matches the observed behaviour of the system (linear leakage) but that is nonetheless

safe. We model the system using a single parameter: the *leakage rate*, while retaining a safe bound on vulnerability.

2.4 Leakage with a Nonuniform Prior

We have so far assumed that secrets are chosen evenly at random—that the attacker's *prior distribution* is uniform. Imagine instead that the secrets are chosen according to some distribution P_S. How does this change our results? There are two reasons to relax this assumption: First, we may be dealing with secrets that really are non-uniform (passwords, or the output of a weak random number generator); Second, as the attacker makes observations, it will begin to consider certain secrets to be more likely than others (those that are more likely to produce the output that it's seen). Thus, after some number of observations, the attacker's *beliefs* form a non-uniform distribution, and we'd like to quantify how much more vulnerable the system has become.

Let us return first to the one-guess vulnerability, V_0 (Definition 1). Here there is little change. Where previously it didn't matter which secret (of those with nonzero probability) the attacker guessed, an optimal attacker will pick the most likely. Thus in general,

$$V_0 = \max_s P_S(s) \tag{2.6}$$

This formula is still valid in the uniform case, albeit trivial.

What about our uncertainty set measure? We'd like to retain the simple linear leakage model, but is this possible? First, we need to establish the optimal attack strategy. Under uniformity, all guesses were equally good, but that is no longer true. The attacker obviously maximises its chance of guessing correctly straight away, and hence V_0, if it chooses a secret that maximises $P_S(s)$ i.e. one of those that are most likely (there may be more than one). However, it is not obvious that doing so maximises its chances for a later guess, if the first one fails. We could imagine a situation where the top two secrets have almost, but not quite, identical probabilities. It might also be the case that if we guess the *less* likely of the two, the side-channel information will reveal the correct value to us, even if our guess was incorrect. In this contrived example, we would maximise V_0 by guessing the first secret (the most likely), but we would maximise V_n for any $1 \leqslant n$ by guessing the second. We could not do both. In this case, we could be optimal with respect to V_0 or

to V_n, but not both. We must therefore further analyse the leakage that our *strcmp* example actually produces, to establish the optimal attack.

Detailed Analysis of an Optimal Attack

We first note a standard result in probability:

Lemma 3: Any joint distribution $P(X_1, \ldots, X_n)$ over a set of n random variables (not necessarily independent), can be expressed in terms of incremental conditional distributions:

$$\begin{aligned} P(x_1, \ldots, x_n) &= P(x_n | x_{n-1}, \ldots, x_1) P(x_{n-1} | x_{n-2}, \ldots, x_1) \ldots P(x_1) \\ &= \prod_i P(x_i | x_{i-1}, \ldots, x_n) \end{aligned} \tag{2.7}$$

Proof. This is a standard result, and follows from the definition of conditional probability. □

Let s_i be the character in position i in the putative secret, s. The probability that the secret is in fact s is then the joint probability that all positions are correct, or $P(s_1, \ldots, s_n)$. By Lemma 3, we may rewrite this as the product of the conditional probabilities $P(s_k | s_{k-1}, \ldots, s_1)$—the probability that the k^{th} position is s_k, given that the first $k-1$ positions have values s_1 through s_{k-1}.

Consider an attack that has exposed $j-1$ positions: s_1 through s_{j-1} are now known. If the attacker wants to maximise its chance of a successful guess, to optimise V_0, it needs to choose the most likely values for the remaining positions, given what it now knows — it needs to choose s_j through s_n to maximise $P(s_n, \ldots, s_j | s_{j-1}, \ldots, s_1)$. A computationally unbounded attacker must be assumed to be capable of finding such an assignment.

Suppose, alternatively, that the attacker wants to maximise its chance of guessing correctly on the *next* try, assuming that the current guess is incorrect (a reasonable assumption in the early stages of an attack). In this case, the attacker is seeking to maximise the *leakage* due to the next guess, or $V_0(k+1) - V_0(k)$. Considering only position j, there are two possibilities, if we fail to guess correctly: either s_j was wrong, in which case we strike it from our list of possibilities and keep trying at position j, or it was correct, in which case we eliminate every other possibility and continue guessing at some later position (not necessarily $j+1$). What is the effect on V_0 in each of these cases?

To answer this, we recall another standard result, Bayes' theorem:

$$P(h|e) = \frac{P(e|h)P(h)}{P(e)} \tag{2.8}$$

This result is the foundation of statistical inference: it formalises the process of updating a statistical model, given new evidence. To see this, it is helpful to rewrite is in this form:

$$P(h|e) = \frac{P(e|h)}{P(e)} \times P(h) \tag{2.9}$$

This is a recipe for converting $P(h)$, the probability that hypothesis h holds given no additional evidence (the *prior probability of* h), into $P(h|e)$, the probability that h holds, once we have observed the evidence, e (the *posterior probability of* h). The important point is that *this is exactly what the attacker is doing*: It begins with some idea, or *belief*, of how likely each secret is (initially we assumed this to be the uniform distribution, which assumption we now relax), and given the evidence that it observes (the intrinsic leakage—the match/mismatch response, and the side-channel leakage—the common prefix length), updates its belief, making some secrets less likely, and others more. Equation 2.9 tells us how an optimal attacker does this: the prior probability of a secret, $P(s)$, is multiplied by the probability of seeing o, if the secret were s, or $P(o|s)$ (the *consequent* probability of o), divided by the probability of seeing o in any case, or $P(o)$. This factor is the *evidence ratio*.

We apply Equation 2.9 to calculate the posterior probabilities for a correct, and an incorrect guess at position j. Suppose first that our guess was wrong: that is, we observed a common prefix length (cpl) of $j-1$. The probability of observing this, if the j^{th} character really were s_j is zero:

$$P(\text{cpl} = j-1|s_j, \ldots, s_1) = 0$$

Thus, the evidence ratio is zero, and so is the posterior probability that s_j is correct, *knowing that the first* $j-1$ *characters are correct, but having observed a common prefix length of only* $j-1$:

$$P(s_j|\text{cpl} = j-1, s_{j-1}, \ldots, s_1) = 0$$

What about the other possibilities, the other characters that we might have guessed? For each of these, say s'_j, the probability of the evidence is one. That

is, if the correct value is s'_j but we guessed s_j, then the common prefix length *must* be $j-1$:

$$P(\text{cpl} = j-1 | s'_j, \ldots, s_1) = 1$$

This time, we need to evaluate the denominator—the prior probability of the evidence. This is the probability of seeing a prefix length of $j-1$, given that we guessed s_j, *without knowing whether or not our guess is correct*. Here we use the decomposition introduced in Lemma 3. Given that s_1 through s_{j-1} are correct, the probability that the prefix length is only $j-1$, is the probability that the position s_j is wrong, *given that the first* $j-1$ *positions are correct*:

$$P(\text{cpl} = j-1 | s_j, \ldots, s_1) = 1 - P(s_j | s_{j-1}, \ldots, s_1)$$

Thus, for any secret that matches s_1 through s_{j-1} and s_j, the posterior probability is zero–it's been ruled out (any mismatches between s_1 and s_{j-1} have already been ruled out by induction). Any secret that matches s_1 through s_{j-1} but doesn't match s_j has its probability increased by a factor of $1/(1 - P(s_j | s_{j-1}, \ldots, s_1))$. Note that any secret whose probability is already zero remains ruled out. The effect is thus to rule out a set of secrets, and uniformly scale up the probabilities of all the remaining possibilities, such that the sum is still one.

What if we guessed correctly? In this case the probability of every secret that *doesn't* match at s_j should go to zero, and we see that this is exactly what Bayes' rule gives us. The probability of seeing a prefix length of j or more, were the correct value anything other that s_j is zero, and thus:

$$P(\text{cpl} \geqslant j | s'_j, \ldots, s_1) = 0$$

From this we see that the posterior probability does indeed go to zero. On the contrary, if the correct value were s_j, we *must* see a prefix length of j or more:

$$P(\text{cpl} \geqslant j | s_j, \ldots, s_1) = 1$$

We must again consider the probability of the evidence, which is now the probability that our guess is *correct*, or:

$$P(\text{cpl} \geqslant j | s_j, \ldots, s_1) = P(s_j | s_{j-1}, \ldots, s_1)$$

and thus

$$P(s_j | \text{cpl} \geqslant j, s_{j-1}, \ldots, s_1) = \frac{1}{P(s_j | s_{j-1}, \ldots, s_1)} \times P(s_j | s_{j-1}, \ldots, s_1) = 1$$

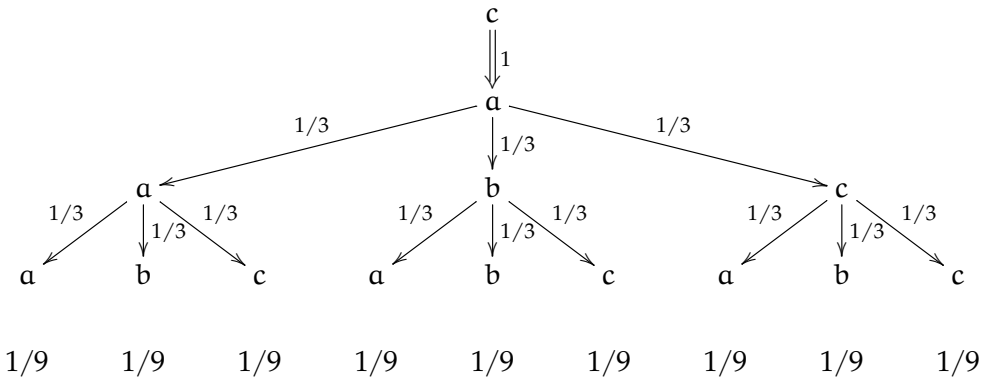

Figure 2.8: An attack state ($m = 3, n = 4$) represented as a tree. Every path from the root is a possible secret. The prefix 'ca' has been established. The edge from s_{j-1} to s_j is labelled with $P(s_j|s_{j-1}, \ldots, s_1)$. The leaves are labelled with the probability of the path that leads to them, or $P(s_j, \ldots, s_1)$

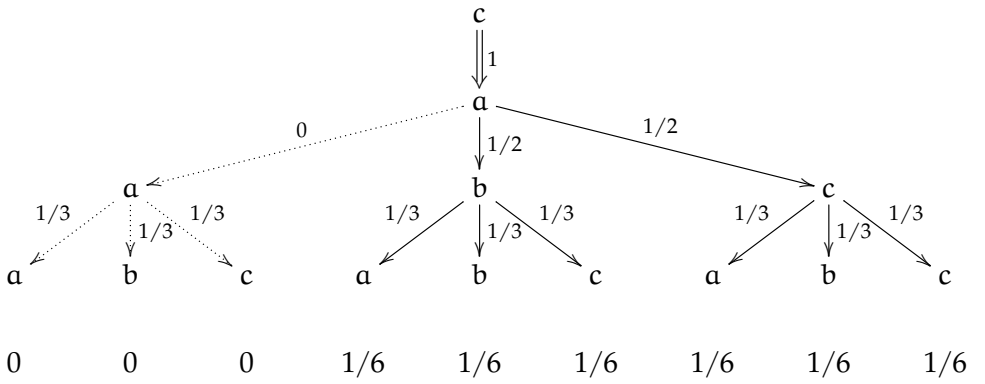

Figure 2.9: The attack in Figure 2.8 after guessing 'caaa' and seeing a prefix length of 2. The left subtree has been eliminated, and the conditional probabilities of the others adjusted, changing the probabilities at the leaves.

So in this case, all secrets that don't match at s_j are ruled out, while all those that do are scaled up by a factor of $1/P(s_j|s_{j-1}, \ldots, s_1)$.

This is not the only inference we can make, however. It's always possible that we have guessed more than one additional character correctly: cpl could be anywhere from j to n. In this case, we eliminate even more possibilities—those that don't match at some point between j and cpl, inclusive. To visualise this, it is useful to view the space of secrets as a tree. Figure 2.8 illustrates the attacker's distribution over secrets, partitioned according to Equation 2.7, with $m = 3$ and $n = 4$. The attacker knows that the first two characters are 'c'

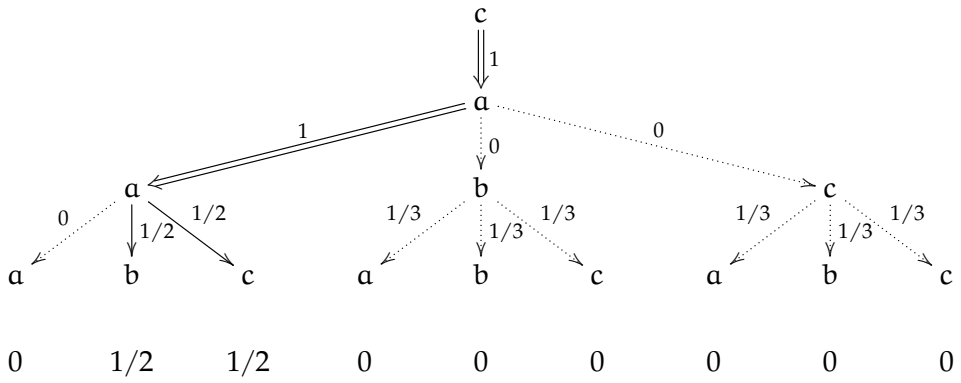

Figure 2.10: The attack in Figure 2.8 after guessing 'caaa' and seeing a prefix length of 3. The middle and right subtrees have been eliminated, as has the subtree rooted at 'caaa'.

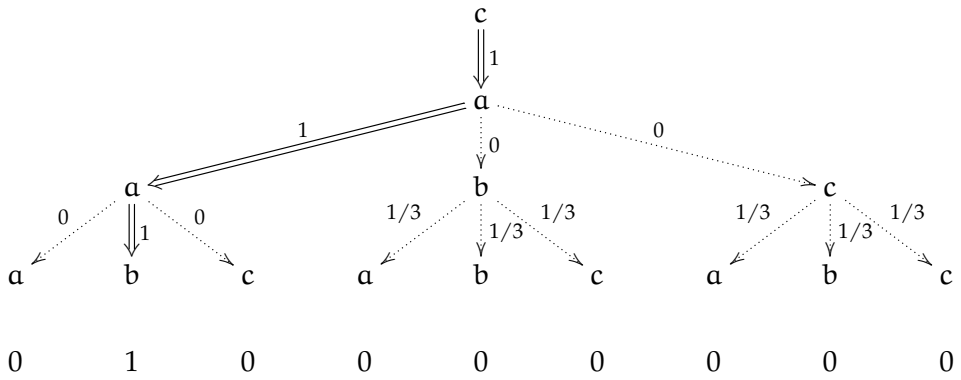

Figure 2.11: The attack in Figure 2.8 after guessing correctly that position j is 'a', *and* that position $j + 1$ is 'b'. The middle and right subtrees have been eliminated, as have 'a' and 'c' in the final row.

and 'a', and is attempting to guess the third. In this example, the distribution of secrets is initially uniform, and is recorded in the final row of the figure, under the leaves. The probability at a leaf is the probability of the secret spelt out by following the path from the root. For example, the probability that the secret is 'cacb' is 1/9. This is the product of the probabilities attached to the edges along the path, in this case $P(c) \times P(a|c) \times P(c|ac) \times P(b|cac)$, or $1 \times 1 \times 1/3 \times 1/3$.

Observing the response lets us update the conditional probabilities along the edges between the last known position and the observed prefix length. For example, Figure 2.9 shows the result of guessing incorrectly at position

3. Here, the subtree rooted at the first incorrect character, s_3, is eliminated and the conditional probabilities leading to the others are scaled accordingly. Note that only the conditional probability for the guess we actually made is updated: the conditional probabilities lower in the eliminated subtree are not changed, but the zero at the root ensures that any path through that subtree has probability zero.

Figure 2.10 shows the result of a correct guess against position 3, which fails at position 4, corresponding to an observed prefix length of 3. Here all other subtrees at depth 3 are eliminated, as is the subtree at depth 4 rooted at our first incorrectly guessed position. Finally, Figure 2.11 shows what happens when more than one position is guessed at once: all other subtrees are eliminated at every level down to the first mismatch. In this case, we have guessed the entire secret correctly, but if there were further levels, the subtree corresponding to our hypothetical mismatch at level 5 would be eliminated, as for Figure 2.10.

Note that in every case, we eliminate some number of subtrees and scale the rest uniformly. Importantly, this means that if the secrets are ordered by probability then this does not change, we simply knock holes as we go. This makes simulating the attack tractable.

So far we have considered only intrinsic leakage. The effect of adding side-channel observations is to change the relative weights of the branches that are left un-pruned, and thus the bottom-row probabilities for complete secrets. We have left this out of our discussion so far, but it does not effect the basic pattern of pruning a branching model that we have established. It is taken into account in the following section, where we simulate a statistically-optimal attacker.

Updating the Leakage Model

This attack strategy is too complicated to analyse exactly, as we did in the uniform case, especially as it depends on the details of the prior distribution on secrets. To understand how it behaves, we instead run an exhaustive simulation. We attack every possible secret with two strategies: one that picks the most likely guess (maximising V_0), and one that guesses to maximise the conditional probability at each branch (maximising leakage, or $V_0(k+1) - V_0(k)$). To keep the simulation tractable, we assume that the distribution on secrets is Markovian: the probabilities for position $j+1$ depend only on the

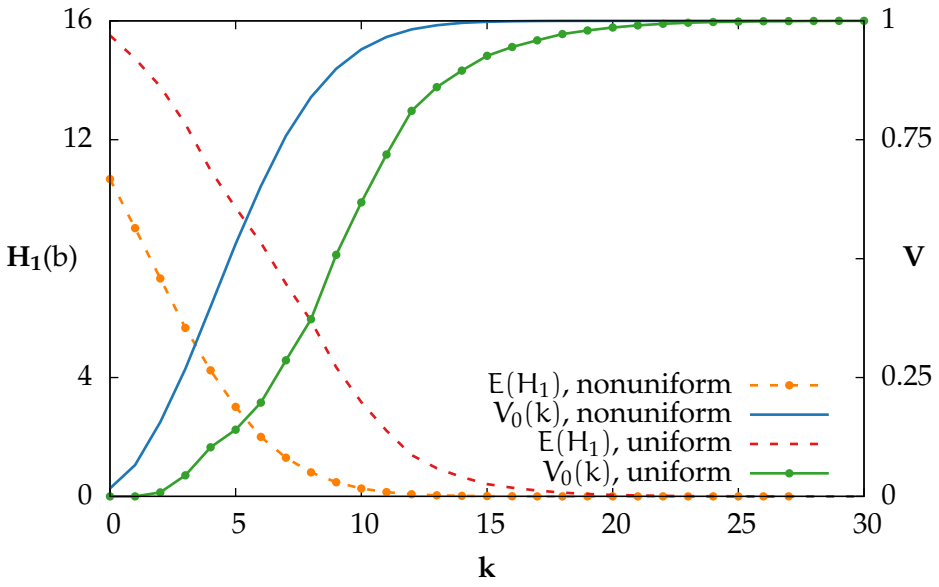

Figure 2.12: Comparison of one-guess vulnerability (V_0) and expected entropy (H_1) for a uniform, and a Markov prior. $m = 6, n = 6$.

character in position j. This is a reasonable model for the broad statistical structure of natural text, in particular passwords, which is of clear relevance to this particular example [Dell'Amico et al., 2010].

The benefit to tractability is that the incremental probabilities all collapse: $P(s_{j+1}|s_j, \ldots, s_0)$ is simply $P(s_{j+1}|s_j)$—every node in the tree has the same pattern of conditional probabilities among its branches. We compress the attacker's model by only storing those branches that have been updated—any branch that hasn't yet changed is simply assumed to have its prior, Markovian, transition probability. To unify the treatment of the initial character (as there's no previous character to transition *from*), we extend the alphabet with a special 'start-of-word' character, $\cdot$. Our table of transition probabilities was generated from the file `alice_29.txt` of the Canterbury data-compression corpus [Arnold and Bell, 1997], representing the English text of Lewis Carroll's Alice in Wonderland [Carroll, 1865].

Figure 2.12 shows the result of an exhaustive simulation of the $m = 6, n = 6$ case (the alphabet was restricted by taking the m most common characters, ordered by position-independent probability, and rescaling the transition probabilities). Also reproduced are the equivalent results in the uniform

case, from Figure 2.5. The expected vulnerability, $V_0(k)$, for the non-uniform case is blue, while that for the uniform case is green. We see that the non-uniform vulnerability follows roughly the same path as the uniform case, but starts at a higher value. This is expected, as in a non-uniform distribution, at least one secret must have greater-than-average probability. We see that the known prior is roughly equivalent to an additional 4 guesses' worth of information. There was little or no difference between the two guessing strategies: these results are for individually-maximised branch probabilities. This is an interesting result in its own right, and means that the attacker may as well optimise locally (for just the next character).

The two additional curves in the figure (red and yellow) need a little more explanation. Rather than the expected uncertainty set size, as plotted in Figure 2.5, we here plot the *Shannon entropy*, or H_1. This is a quantity of fundamental importance in information theory—a measure of the 'size' of a set, taking into account uneven likelihood amongst its members. For a distribution P, the Shannon entropy is defined as:

$$H_1 = -\sum_x P(x) \log_2 P(x) \tag{2.10}$$

Here the logarithm is taken to base 2, giving an answer in *bits*. The entropy under a different base differs only by a constant factor. Note that if the distribution P is uniform i.e. $P(x) = 1/|S|$, then $H_1 = \log_2|S|$. Thus, by plotting V_u on a logarithmic scale, we actually snuck Shannon entropy in some time ago, under the guise of the uncertainty set. The red curve in Figure 2.12 is thus identical to that in Figure 2.5. Importantly, the new yellow curve, giving the expected entropy vs. guess number for a non-uniform prior has the same slope, shifted down according to the difference in the initial entropy. This gives meaning to our log-linear leakage measure: saying that 'set size drops by a factor of 6 every 3 guesses' means that we are leaking (on average) $\log_2 6 \approx 2.58$ bits every 3 guesses, or 0.86 bits per guess. Changing the prior distribution has not affected the *rate* of leakage, only the initial entropy, and our linear model is still valid.

There is an alternative definition of entropy that gives a worst-case, rather than an average-case summary: the *min entropy*. This measure depends on the greatest probability in the distribution alone:

$$H_\infty = -\log_2 \max_x P(x) \tag{2.11}$$

Note the relation to Equation 2.6: the min entropy is simply the (negated) logarithm of the one-guess vulnerability for an optimal attacker. We compare leakage measures derived from these two definitions shortly.

Given that we have replaced the expected set size by the entropy, we must ask whether the pessimistic correction that allowed us to bound one-guess entropy still holds, and if not, with what we should replace it. As already noted, Smith [2009] demonstrated that for some distributions, Shannon entropy is a very poor guide to one-guess vulnerability—the two may diverge dramatically. This observation has lead to a strong move away from Shannon entropy in the literature, toward min entropy, as a safer measure.

Knowing that they diverge in *some* cases only establishes that entropy does not give us a universally applicable bound. However, if we can bound this divergence, we can show that Shannon entropy *can* be used as a safe security measure. What we need to know is: *How bad can the vulnerability be, if we know the entropy?* We derive this divergence exactly, as the subject of the next section.

2.5 Reevaluating Shannon Entropy

Figure 2.12 shows that the simple model of Figure 2.7 still holds. There we saw that the size of the uncertainty set fell geometrically i.e. the logarithm fell linearly. We now see that the generalisation of the (log) uncertainty set size to a non-uniform distribution, the *Shannon entropy*, also shows a linear decrease (down to the point at which we reach a vulnerability of 0.5, at least). We thus continue to use a linear leakage model. Just as in Lemma 2 however, we need to take into account that we have only a summary measure of the distribution. There we knew the expected uncertainty set size, while here we know only the entropy of the distribution—there may be many distributions with a given entropy, with different vulnerabilities. We thus need to establish *the greatest possible vulnerability given Shannon entropy*.

The Divergence of Vulnerability and Shannon Entropy

Our key result is the following lemma, which tells us how to maximise vulnerability while holding entropy constant. From this, we show that the vulnerability increases (approximately) linearly, as entropy decreases.

Lemma 4 (Maximising V_0 Given H_1)**:** Let

$$Q = \{P : H_1(P) = H\}$$

be the set of distributions over the set X with Shannon entropy H. Let $N = |X|$. Fix $n \in \mathbb{N}$ and for each $P \in Q$, partition X (disjointly) as $Y \cup Z$ such that

$$\forall y \in Y, z \in Z.\ y \geqslant z \wedge |Y| = n$$

For all distributions $P \in Q$, $P(Y)$ (the combined probability of all $y \in Y$) is bounded above by the curve

$$h(p) + p \log_2 n + (1-p) \log_2 (N - n) \tag{2.12}$$

where

$$h(p) = -p \log_2(p) - (1-p) \log_2(1-p) \qquad \text{and } 0 \log_2 0 = 0$$

Moreover, this bound is tight. For $H \geqslant \log_2 |Y|$, it is reached by a distribution of the form

$$P(x) = \begin{cases} P(Y)/\|Y\| & x \in Y \\ (1 - P(Y))/\|Z\| & x \in Z \end{cases}$$

If $H < \log_2 \|Y\|$, a solution exists with $P(Y) = 1$.

Proof. See Appendix A □

Corollary 1: The greatest one-guess vulnerability, V_0, of any distribution with given H_1 entropy H is a solution of:

$$H = h(V_0) + (1 - V_0) \log_2 (N - 1) \tag{2.13}$$

Proof. By substituting $n = 1$ and $p = V_0$ into Equation 2.12. □

Figure 2.13 plots Equation 2.13 for $N = 8$. Note that between 0 and 1:

$$H'(V_0) = \log \frac{\frac{1}{V_0} - 1}{N - 1}$$

and

$$H''(V_0) = \frac{-1}{V_0 - V_0^2}$$

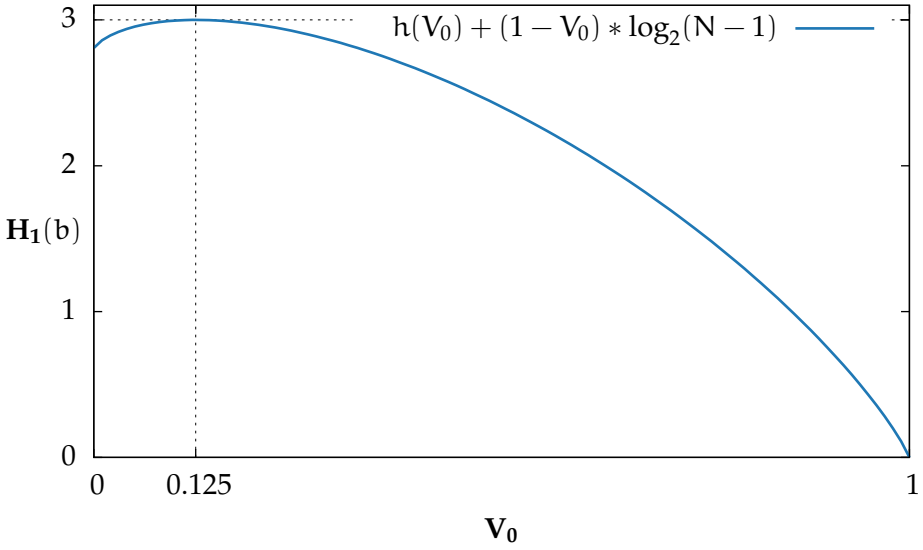

Figure 2.13: Graph of Equation 2.13, showing maximum at $V_0 = 1/8, H_1 = 3$. $N = 8$.

We see that the second derivative is negative everywhere in $(0, 1)$ (the function is convex), and the first is zero only at $1/N$, the maximum. Thus, H is maximised by the uniform distribution ($P(x) = 1/N$), giving an entropy of $\log_2 N$. For a given entropy, there are generally two solutions, but as we are solving for the *greatest* vulnerability, we need only consider the portion of the curve to the right of the maximum. On the interval $[1/V_0, 1]$, $H(V_0)$ is strictly monotonic, as the derivative is negative everywhere in $(1/V_0, 1)$. As H is also continuous, it is therefore invertible on this interval.

Figure 2.14 is the diagonal reflection of Figure 2.13 i.e. its inverse. This graph gives the function $\mathcal{K}_N$—the worst-case one-guess vulnerability for a given H_1 entropy, given N possible secrets. As the inverse of an anti-monotonic, convex function, $\mathcal{K}_N$ is also anti-monotonic and convex. Thus, by Jensen's inequality, given an expected entropy H, $E(\mathcal{K}_N(H))$ is maximised by the distribution that assigns all probability to H. Therefore, given an expected entropy, we need only look at the curve of $\mathcal{K}_N$ to find the maximum expected vulnerability. We therefore do not need to correct for using the expectation rather than the true value.

As N increases, the term $h(V_0)$ in Equation 2.13 is dominated by $\log_2(N-1)$,

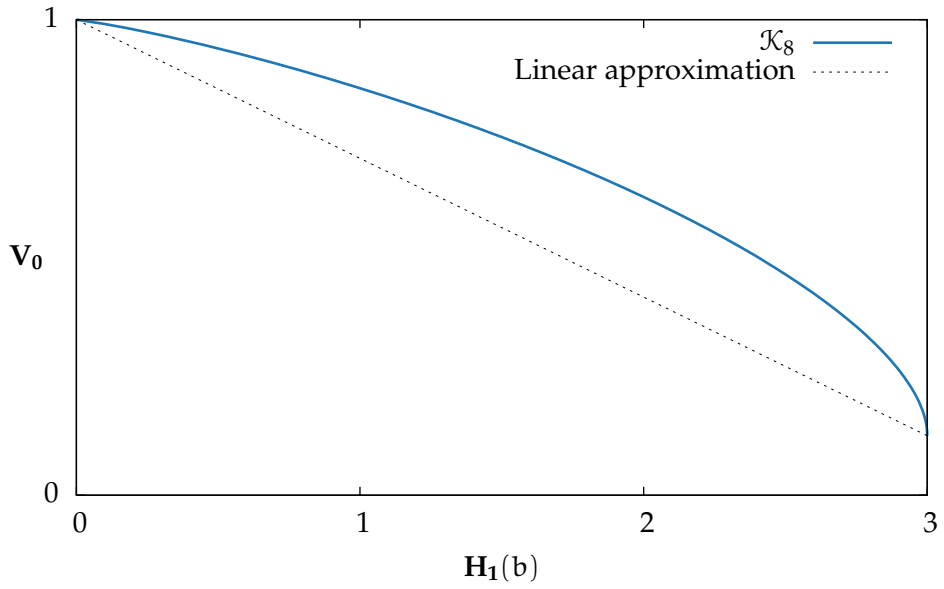

Figure 2.14: Pessimistic correction function, $\mathcal{K}_8$, derived from Figure 2.13.

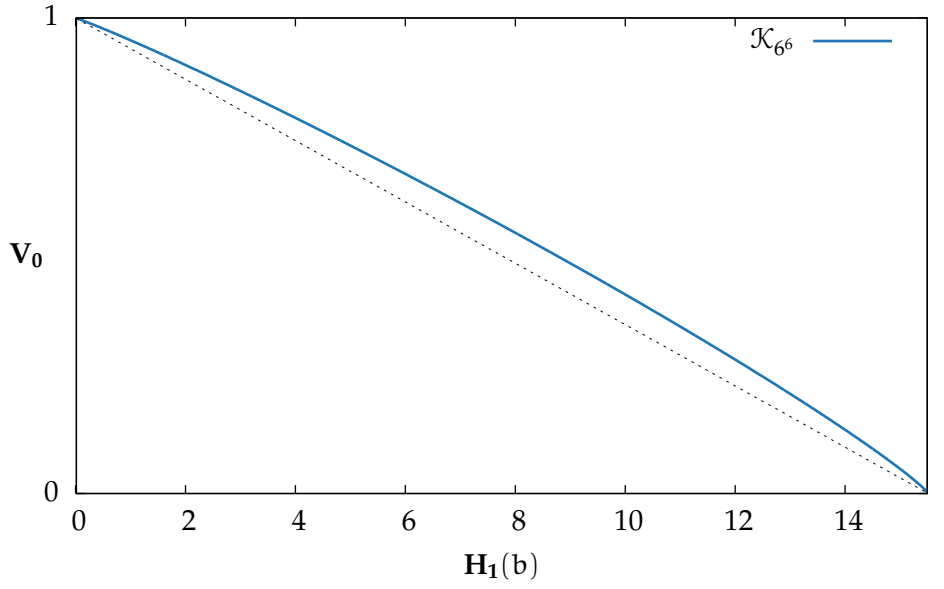

Figure 2.15: Worst-case expected one-guess vulnerability given H_1 entropy, for a distribution over 6^6 secrets, showing nearly linear behaviour.

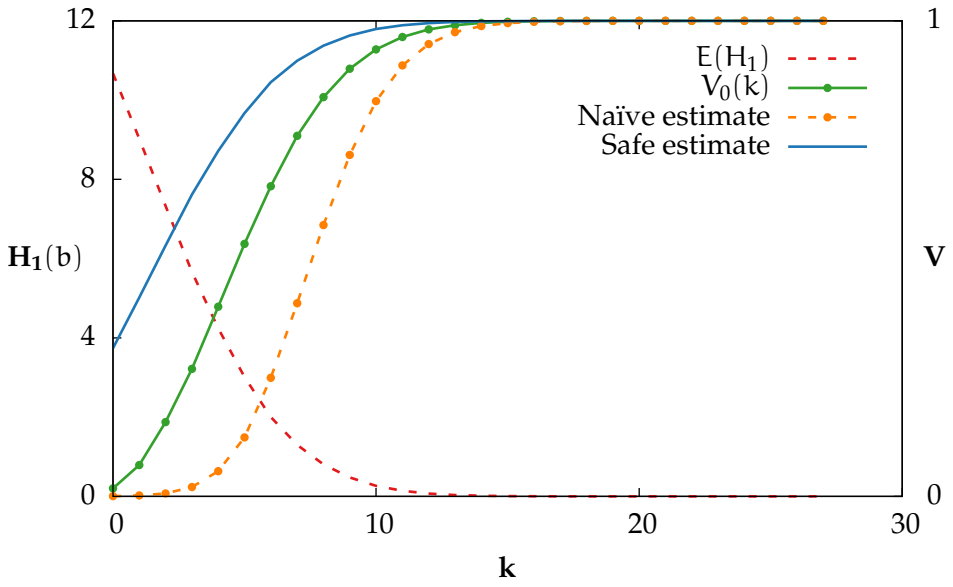

Figure 2.16: Expected vulnerability and entropy for $m = 6, n = 6$, showing both a naïve vulnerability estimate, and its safe correction.

and $\mathcal{K}_N$ converges on a straight line between the points $\{0, 1\}$ and $\{\log_2 N, 1/N\}$. Figure 2.15 shows $\mathcal{K}_{6^6}$, and its linear approximation.

Updating the Leakage Model

We use this linear approximation to take the expected Shannon entropy, H_1 from Figure 2.12, and calculate a bound on true vulnerability, plotted in Figure 2.16. Entropy is in red, and vulnerability in green. We see that our upper bound is indeed safe, and only moderately pessimistic. We substantially overestimate the vulnerability initially, but converge on the true value. Included for reference is a naïve approximation of vulnerability: 2^{-H_1}. This estimate is correct if the distribution is uniform; it is a lower bound in general.

2.6 Noisy Channels & Information Theory

We have so far assumed that the attacker is able to measure response time precisely: that it is able to sample the side channel with unlimited fidelity. In practice, an attacker is unlikely to have access to perfect measurements and in any case, while the number of loop iterations may be the principal factor

affecting runtime, there will typically be many more, making the attacker's job more difficult. This is particularly true for a remote exploit, where the attacker runs on a separate machine, and must make its measurements over a shared network, perhaps even the Internet as in the remote side-channel exploit against OpenSSL reported by Brumley and Boneh [2003], or lucky thirteen attack of AlFardan and Paterson [2013] that we investigate in Section 3.6. In this case, it is likely that the variation in runtime that the attacker is attempting to measure will be small compared with the confounding effects, principally *noise*.

Despite this, it is often possible to exploit such a channel remotely (as we will demonstrate), exploiting standard signal-processing techniques to extract even a minuscule signal from overwhelming noise. The optimal attacker in this case is Bayesian, as in the example attack of Section 2.4. We first recap several standard results in information theory, before we describe how they are applied in the context of a side-channel attack.

Information theory arose from the study of communication over noisy channels [Shannon, 1948], between a *sender* and a *receiver*, with obvious parallels here. In our case, the receiver is the attacker, while the sender is either a trojan horse (for a covert channel), or an unwitting trusted program (for a side channel). Information flow is defined as the increase in the receiver's knowledge over time, through observing the sender's effect on the channel. In practice, we usually work with an equivalent formulation: by measuring the *decrease* in the receiver's *uncertainty*.

Given a noisy channel, the receiver will never be completely certain what message was actually sent: the best it can do is to assign *probabilities* to the various possibilities. The state of maximal uncertainty is one in which the receiver assigns to each message the general probability of it being sent, averaged over a long interval. That is, the receiver can only guess based on how the sender usually behaves, but has no idea *what the sender actually did*. Anything that allows the receiver to refine its assessment of probabilities, reducing its uncertainty, represents a transfer of information from sender to receiver. In the case that any message is equally likely (for example, an encryption key being leaked directly over a side channel), the distribution of maximal uncertainty will be uniform.

To fully state the receiver's uncertainty, or state of knowledge, it is sufficient to present its current belief distribution: this is the only *complete* ac-

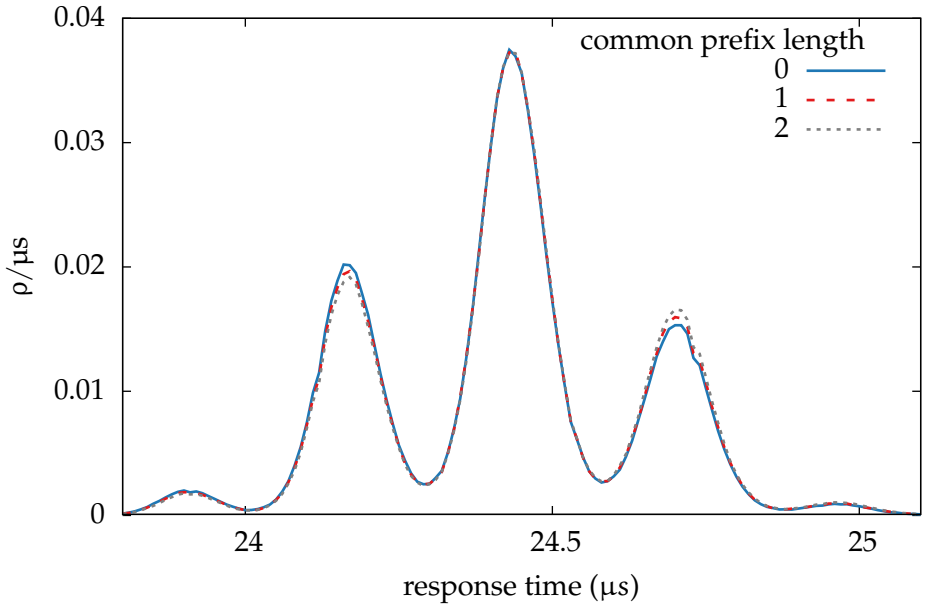

Figure 2.17: `strcmp` execution time distribution, via RPC over the Linux networking stack, showing probability density (ρ).

counting. However, to define information flow, and compute the capacity of a channel, working with full distributions is hopelessly unwieldy. We instead choose a *summary statistic* for the distribution that is consistent with our definition of information. The rate of information flow is the rate at which this parameter (the uncertainty) reduces, as the channel is used.

The standard summary measure is the *Shannon entropy*, H_1, as defined in Equation 2.10. The rate of decrease of H_1 in the attacker's belief distribution is the rate at which it is receiving information, in bits per second (assuming base 2 logarithms). If we can bound the rate at which information leaks, we can place a lower bound on the expected entropy remaining after a given number of guesses. Then, by applying our correction function, $\mathcal{K}_n$, we can bound the expected vulnerability. Measuring the change in min entropy, H_∞, gives an alternative definition of information flow, and we compare the two shortly.

What is the limit on the rate of information flow via the side channel? Consider Figure 2.17, which gives the distribution of response times for `strcmp` applied in a realistic scenario: Here, we implement our example password checker, comparing the supplied string to a known secret. To complicate matters for the attacker, it is only able to call the password checker remotely:

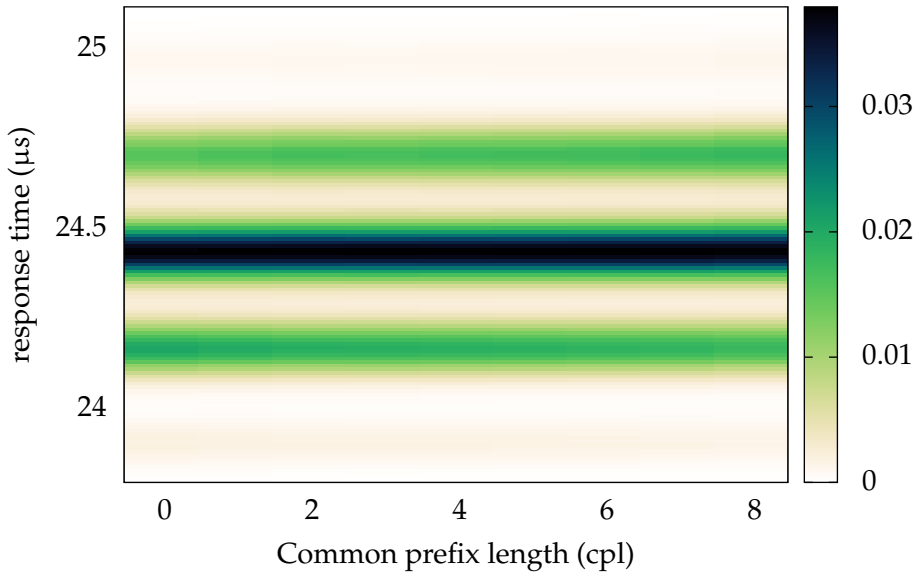

Figure 2.18: Channel matrix for `strcmp` leakage. $C = 2.34 \times 10^{-3}$b.

via a network socket. All response times thus include two trips through the operating system's network stack (Linux in this example).

The round-trip time dominates the execution time of `strcmp`: the median response time is slightly less than 24.5µs, while the underlying execution time is on the order of 10ns. Curves are shown for a common prefix length of 0, 1, and 2. The general pattern continues for larger values. The probability mass outside the range shown is negligible.

Given the difference between the execution time of `strcmp` and the round-trip time, it is unsurprising that the overall response time is not proportional to the prefix length. We do see, however, a multimodal distribution with peaks roughly 250ns apart, corresponding to some unknown quantisation effect in the packet processing path. Our small runtime variation is sufficient to cause a small change in the relative weights of the second and fourth modes: comparisons with a low common prefix length are slightly more likely to appear in the left-hand peak, and those with a high prefix length, in the right. Is this subtle variation enough to exploit the side channel? And if so, how well?

To quantify this leakage, we first take the curves from Figure 2.17, together with those for prefix lengths up to 8, and place them side-by-side to

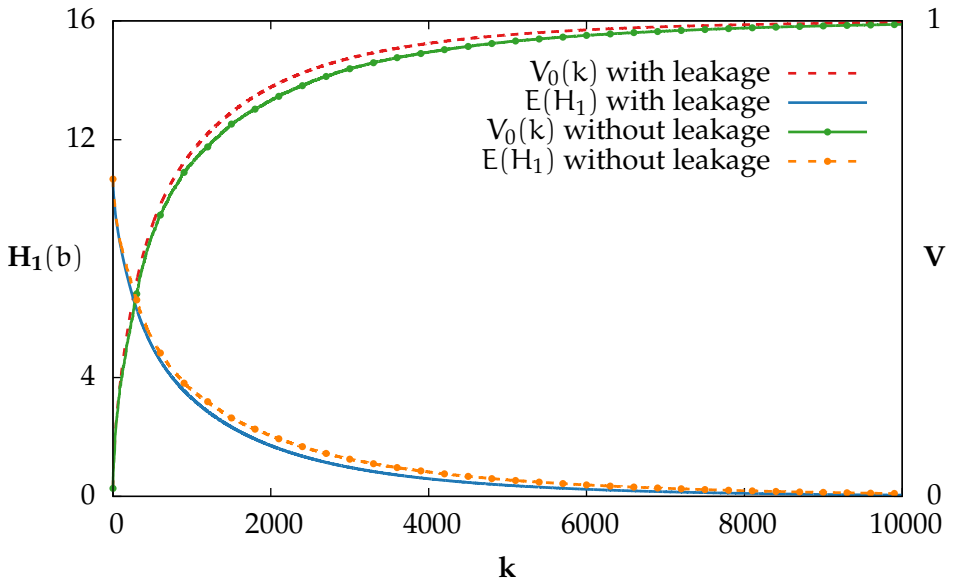

Figure 2.19: Intrinsic leakage showing both vulnerability and entropy, contrasted with leakage including the side channel.

create Figure 2.18. In this figure, common prefix length increases across the horizontal axis, and response time along the vertical. The shading gives the probability density (the height of the original curve). A vertical slice through the figure at cpl $= 2$ is thus just curve 2 in the original figure. This is a *channel matrix*, which is simply a matrix of conditional probabilities: the value at point $\{a, b\}$ is the conditional probability of seeing response time b, if the prefix length is a. We are thus modelling the leakage as a *channel*, taking the prefix length as input, and giving a response time as output. We can place an upper bound on the rate of leakage, by calculating the *capacity* of this channel—the greatest *average* rate of information flow possible.

This channel can transmit up to 2.34×10^{-3}b every time it is used. We use channel matrices extensively in Chapter 3.

We use this channel matrix to simulate a large number of attacks against the system, by the optimal attacker introduced in the previous section. The attacker's strategy is essentially identical to that described in Section 2.4, except that the attacker now only has a distribution on prefix lengths, informed by its observation of the channel output. The branch probabilities are thus updated to the weighted sum of their values for each possible prefix length.

The result of running 1000 attacks per secret, with the Markov prior of Section 2.4 is shown in Figure 2.19, showing the trajectory with the side channel enabled and disabled. We see that vulnerability rises faster, and the remaining entropy drops faster, once the side-channel output is incorporated by the attacker. Recall our criterion: we judge the security of the system (with respect to side-channel leakage) by the difference between the *intrinsic* vulnerability (the no-leakage curve) and the total vulnerability. The remainder is the *extrinsic*, or side-channel leakage, which is the *difference* between the curves.

We see immediately that while the difference in ultimate vulnerability is small, the effect of the side channel is clearly discernible. The important question here is whether we can apply the techniques developed in the previous sections to *predict* this leakage, given only a reduced description of the system: we'd rather not have to run the extensive (and expensive) simulations required to produce this figure for every system we design. The answer is yes, and relies on combining the measurement of channel capacity with our newly-established vulnerability corrections.

Capacity is defined using two new quantities: *conditional entropy*, and *mutual information*. The entropy of the random variable S, given observation o, or $H_1(S|o)$, is the entropy that remains in the distribution on S, once o is observed. For example, imagine that s, a 10 bit secret chosen uniformly at random, is a particular value of the random variable S, and o is a single bit of s. Knowing o, we can eliminate half of the possible values of s, but those that remain will still be uniformly distributed among themselves. The initial entropy, $H_1(S)$ of 10 bits, has thus dropped to 9 on observing o.

The conditional entropy $H_1(S|O)$ is the expected value of $H_1(S|o)$ over all possible values o:

$$H_1(S|O) = \sum_o P(o) H_1(S|o)$$

In our example, both $o = 1$ and $o = 0$ are equally likely, and both give $H_1(S|o) = 9$. Thus $H_1(S|O) = 9$: 1 bit of the information in S is revealed by observing O, on average. Viewed as a channel, we say that the output O carries 1 bit of information from the input S. This expected increase in information about S on observing O is the mutual information:

$$I(S;O) = H_1(S) - H_1(S|O)$$

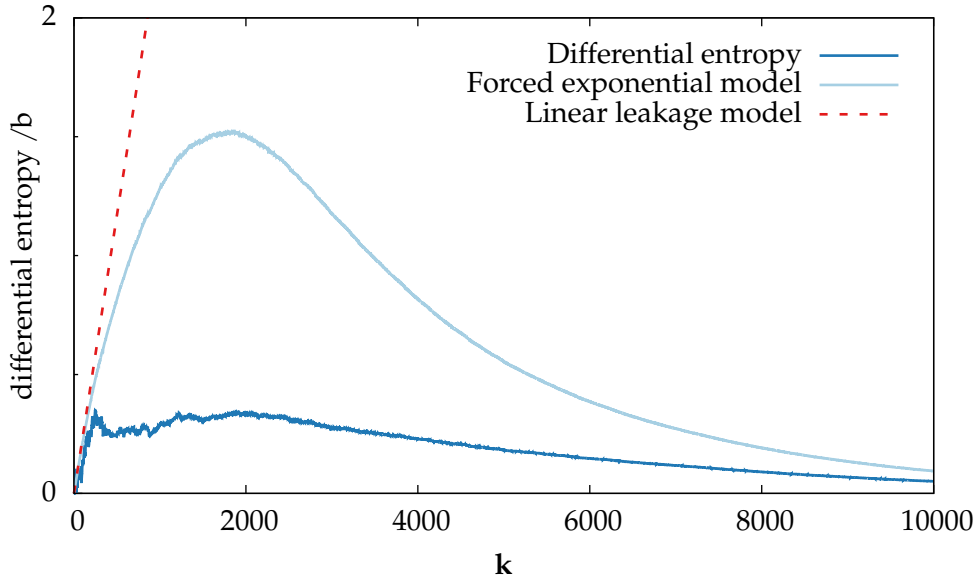

Figure 2.20: Increase in leakage due to the `strcmp` side channel, including forced exponential model. Also shown is the maximum possible leakage given channel capacity.

Note that this flow is parameterised by the distribution $P(S)$ (given $P(S)$, $P(O)$ is determined as $P(o) = \sum_s P(o|s)P(s)$). The capacity of the channel is the *greatest* mutual information between input and output for any distribution on S (and hence on O):

$$C = \sup_{P(S)} I(S;O)$$

This is the greatest (average) rate at which information can flow over the channel, and there exist coding schemes that approach this capacity arbitrarily closely [Shannon, 1948].

By calculating the capacity of the channel matrix, we bound the rate of information flow from above, and thus the expected entropy from below. Recall that we are concerned with the difference in vulnerability between the system with no leakage (its intrinsic vulnerability), and its vulnerability with the side channel included. We are thus concerned with the additional reduction in entropy, above that due to intrinsic leakage.

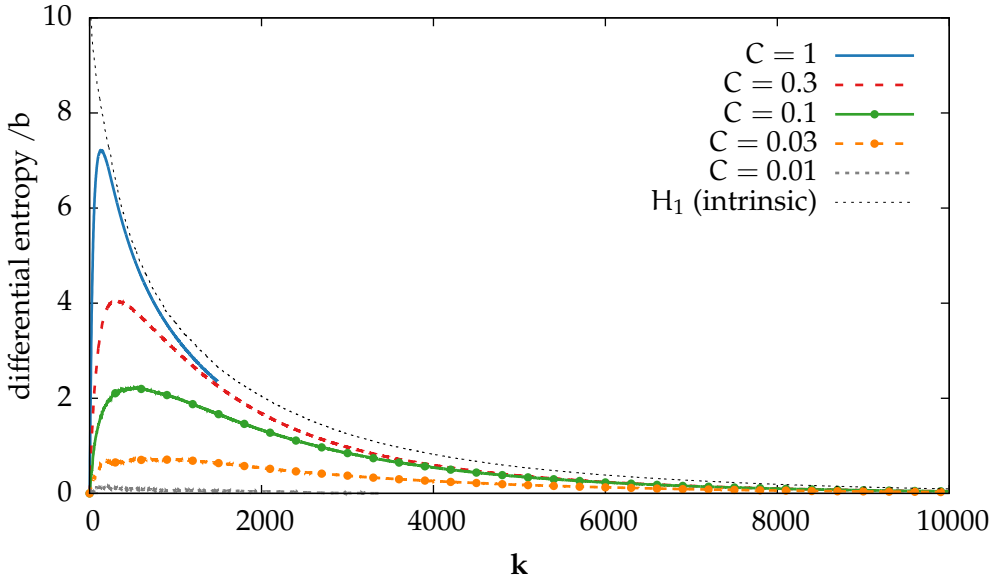

Figure 2.21: Entropy differential for simulated channel matrices of varying capacity, bounded by the intrinsic entropy and showing asymptotic behaviour.

2.7 A Safe Leakage Model for `strcmp`

The final refinement of our approach is to show that by using the channel capacity, we can derive a model for the combined leakage (intrinsic and side-channel), that provides a reasonable approximation of the observed behaviour across a number of leakage rates. Further, by applying the pessimistic correction of Section 2.5, we derive a safe bound on vulnerability that is significantly tighter than that given by min leakage, beyond about 100 guesses.

Figure 2.20 plots the difference between the two entropy curves in Figure 2.19 in dark blue. This is the information flow due solely to the side channel—the extrinsic leakage. The red line is the greatest possible leakage due to the side channel given k observations, or kC where C (the channel capacity) is 2.34×10^{-3}b (see Figure 2.18).

We see that the observed leakage is close to the theoretical maximum for a relatively brief period of less than 100 guesses, and then tapers off rather than continuing to increase. We can better understand this behaviour if we look at the differential entropy curves for channels of different capacities, in Figure 2.21.

This figure shows the measured differential entropy for 6 algorithmically-generated channel matrices with capacities ranging from $C = 1$b down to $C = 0.01$b. These curves bear a clear resemblance to each other, and to that of Figure 2.20, initially rising in line with the channel capacity before reaching a peak and decaying in a roughly exponential fashion to 0. Note that none of the curves ever exceeds the intrinsic leakage. This is as expected, as doing so would force the final entropy to be negative. As C increases, the differential entropy converges (pointwise) on the intrinsic leakage curve.

This linear rise and exponential decay resembles, roughly, the curve xe^{-x}—a term that arises as the solution of a linear first-order differential equation. Following this observation, we hypothesise the following model:

$$D'(k) = \frac{H_{\text{intrinsic}}(k) - D(k)}{H_{\text{intrinsic}}(k)} C + \frac{D(k)}{H_{\text{intrinsic}}(k)} H'_{\text{intrinsic}}(k)$$
$$D(0) = 0$$

Here we calculate the *rate* of side-channel leakage (D), as a linear combination of the channel capacity and the rate of intrinsic leakage, with a total weight of 1 at every point.

The intuition is that if linearly-increasing side-channel leakage is combined with uncorrelated intrinsic leakage, then as we learn more and more, a greater fraction of what we learn via the side channel simply duplicates something we already know via the intrinsic channel. The above equation assumes complete independence: if we have 2, of a total of 8 bits via the side channel, and 4 via the intrinsic channel, we should expect an overlap of $4 \times \frac{2}{8} = 2 \times \frac{4}{8} = 1$b of duplicate information. We would thus have a total of 5 bits of knowledge, rather than 6, as would be the case if the information were disjoint. Under this model, as the differential entropy approaches the intrinsic entropy, its rate of growth should slow, until it eventually reverses (and the differential entropy starts to drop), as the intrinsic leakage accounts for more and more of the information that was previously unique to the side channel. The differential leakage will eventually converge to zero, as the intrinsic and total leakage both converge on the initial entropy.

Mathematically, the differential equation is forced by H'_{initial}. This "forced exponential" model is plotted in Figure 2.22, for a subset of the channel matrices used in Figure 2.21.

We see that the shape of the curves is correct, as is their asymptotic behaviour. That they never cross the intrinsic entropy (which would imply

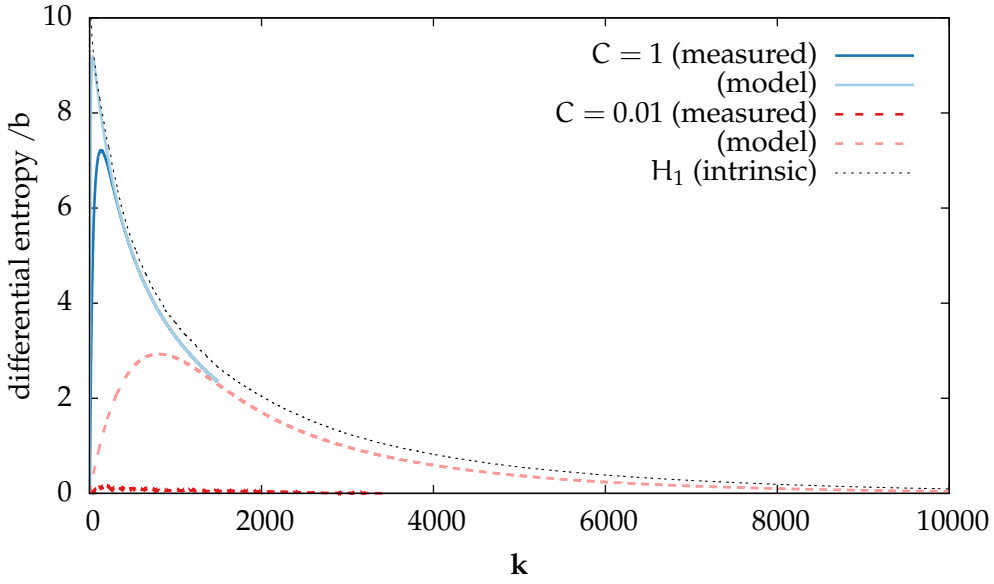

Figure 2.22: Differential leakage contrasted with forced exponential model.

a negative total entropy) is a good sanity check. We also have a good fit for $C = 1$ (the true curve is shown dashed), which gets progressively more pessimistic as C decreases. There is clearly another effect at work which is suppressing the leakage at low capacities by more than we anticipate. We could continue to refine the model, but it is already sufficient to make our point: Shannon entropy is a natural summary measure for the leakage in this system, and leads to a straightforward model of its behaviour. Figure 2.23 shows the behaviour of our approximation for small numbers of guesses. In each case, we overestimate the initial rate of leakage—see the difference between the two blue curves in Figure 2.21.

We are now, at last, in a position to model the leakage of the full system: using a non-uniform prior, and with leakage via the empirically established channel matrix of Figure 2.18. We do so by first estimating the side-channel (extrinsic) leakage using Figure 2.20, and then calculating a safe bound on expected vulnerability using Equation 2.13.

The results of our estimation are presented in Figure 2.24. Here, the solid blue curve is the true vulnerability, and the red curve the true entropy. The orange curve is the entropy estimated using our forced exponential model. The green curve is the vulnerability bound calculated using the true entropy,

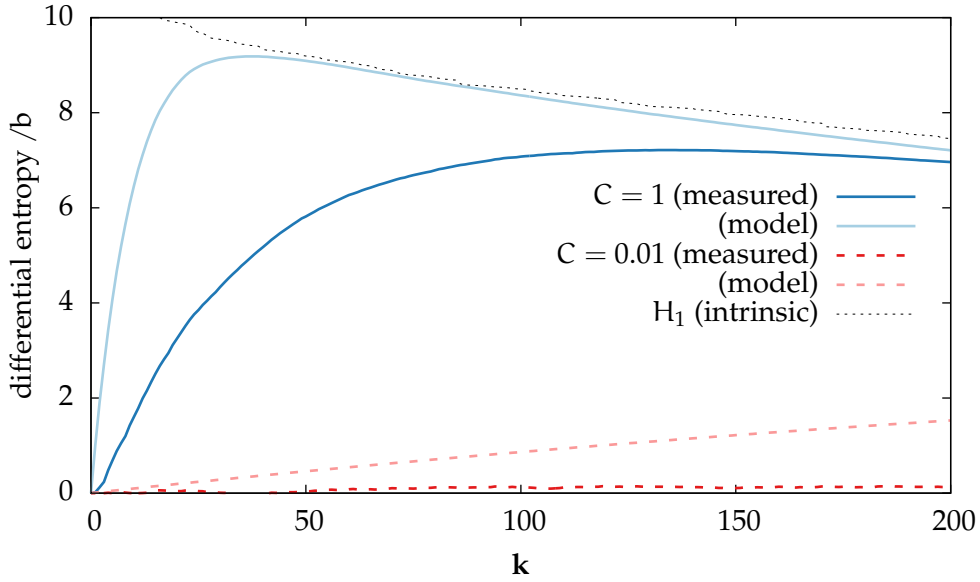

Figure 2.23: Detail of Figure 2.22, showing behaviour near $k = 0$.

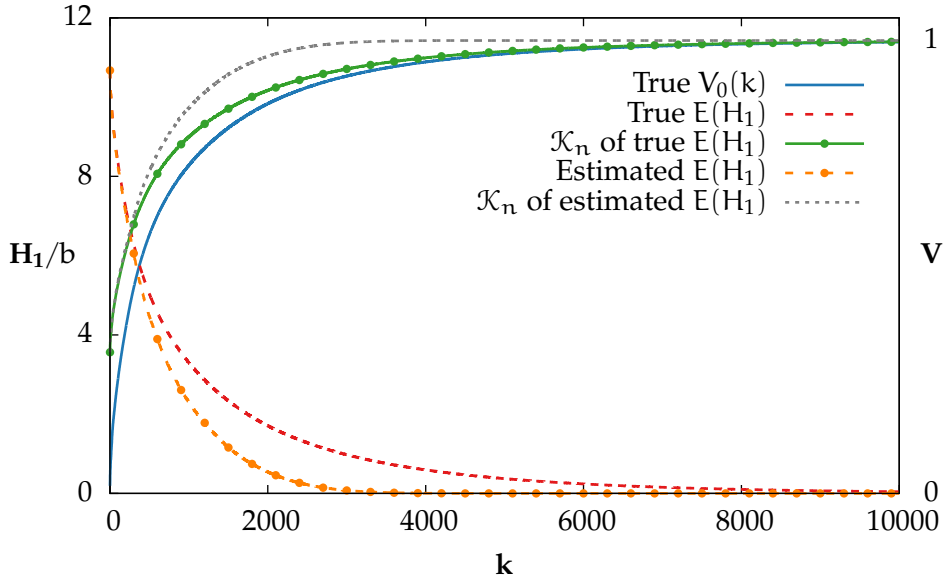

Figure 2.24: Vulnerability and expected entropy for $m = 6, m = 6$, including side-channel leakage. Shows vulnerability estimated from true entropy, entropy estimated using uncorrelated model and derived vulnerability estimate.

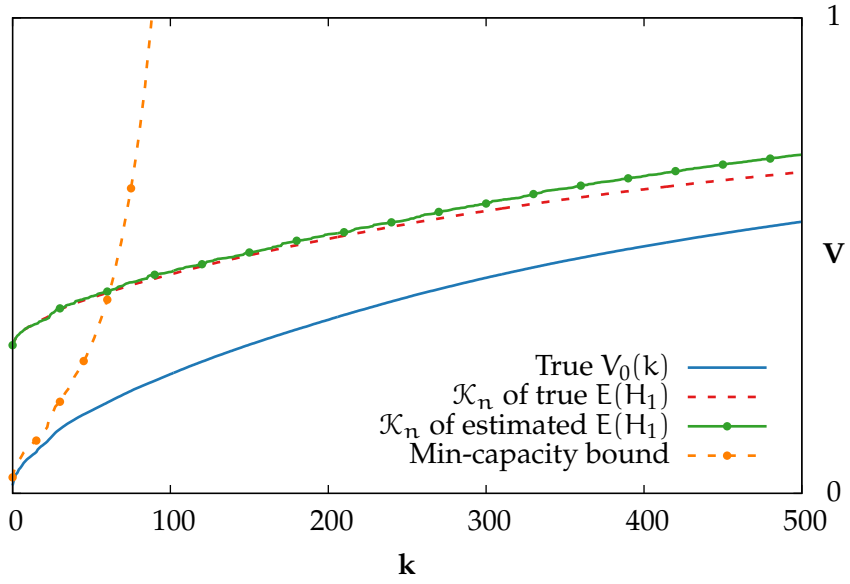

Figure 2.25: Detail of Figure 2.24, showing low guess numbers. Includes the min-leakage bound for comparison.

and the grey curve gives the vulnerability calculated from the estimated entropy. This final estimate depends only on the capacity of the side channel: the differential entropy (and hence total entropy) is calculated from the intrinsic leakage and channel capacity, with vulnerability estimated using our established vulnerability correction. As we see from the graph, the bound is safe, and only moderately pessimistic.

Finally, Figure 2.25 is a detail of Figure 2.24, in the region around $k = 0$. This graph includes a vulnerability bound using min entropy, H_∞, and its associated leakage measure, the min capacity: $\mathcal{ML}$. The initial min entropy is logarithmic in the initial vulnerability of $\approx 1.68 \times 10^{-2}$, giving $H_\infty(0) \approx 5.90$b. The min capacity bounds the rate at which the min entropy drops, in a manner analogous to the Shannon capacity of the channel. It is calculated from the channel matrix as:

$$\mathcal{ML} = \log_2 \sum_o \max_s P(o|s)$$

From Figure 2.18 we calculate a min capacity of $\approx 6.26 \times 10^{-2}$b. The orange curve in Figure 2.25 shows the effect of a linear reduction in min entropy:

$$H_\infty(k) = 5.90\text{b} - k \times 6.26 \times 10^{-2}\text{b}$$

The exponential shape of the curve arises once we convert from min entropy back to vulnerability:

$$V_0(k) = 2^{-H_\infty(k)}$$

As we see, while the min capacity model is safe, it is also wildly pessimistic. Our corrected Shannon entropy derived bound, by contrast, tracks the true vulnerability much more closely. However, note that for fewer than approximately 50 guesses, min leakage gives a tighter bound, as in this area the pessimism introduced by the Shannon entropy correction is relatively large. Therefore, while we will generally use Shannon capacity as our measure of choice, for attacks involving small number of guesses we instead quote min capacity (for example, in the remote attack presented in Section 3.6).

2.8 Related Work

Side and covert channels have traditionally been treated as noisy communication channels, as studied in the field of telecommunications [Shannon, 1948]. Naturally, the relevant techniques (channel matrices, mutual information and channel capacity) were applied to these new channels. For example, the US Trusted Computer System Evaluation Criteria (orange book) standards [DoD] specify a maximum covert channel bandwidth (0.1 or 1 bit per second, depending on level), for certified systems. Academic and industrial systems generally applied the same methods, for example the analysis of fuzzy time, by Trostle [1993], or in the work on countermeasures of Gray [1993, 1994].

More recently, there has been a growing realisation in the field, first clearly articulated by Smith [2009], that the uncertainty measure in standard information theory, Shannon entropy, gives only weak security guarantees. Smith proposed that min entropy was more appropriate, as it directly addresses vulnerability to guessing. This approach has since been broadly adopted, for example by Braun et al. [2009]. Further measures are still occasionally proposed, for example the work of Clarkson et al. [2005] on the possibility of incorrect beliefs on the part of the attacker. We follow the consensus in using min entropy as a safe measure, retaining Shannon entropy for its attractive modelling properties. Köpf and Basin [2007], in analysing cryptographic primitives, likewise settle on average case measures for reasons of practicality.

Formal work on min entropy and other measures continues to progress. Köpf and Smith [2010] present simple mathematical bounds on min leakage,

the analogue of channel capacity, some of which we recreate from our formal model in Chapter 6. Andrés et al. [2010] demonstrate that channel matrices can be efficiently derived for small formal system models.

A guessing attack, while broadly applicable, is not the only, or the most general threat model. This model implicitly assumes that the security state is fundamentally binary: compromised or secure; the attacker achieves complete compromise by guessing the secret, and the system remains secure for as long as it guesses incorrectly. *Gain functions*, and the associated *g-leakage* measure[Alvim et al., 2012] give a more general model. Here, the attacker's "reward function" is not binary: the attacker may get some benefit from a partially-correct guess. This generalises min-entropy: it reduces to it for a gain function that assigns 1 to the correct secret and 0 to everything else. Alvim et. al. also show that min-capacity gives an upper bound on both g-leakage and Shannon leakage although, as we have demonstrated, min-leakage is generally highly pessimistic. Our present work should, in principle, be extendible to g-leakage, although we have not as yet investigated this possibility.

The work of Espinoza and Smith [2012] further demonstrates that channels characterised by min leakage can be composed in the same fashion as classical channels. Most recently, Morgan et al. [2014] have shown that a novel ordering on processes, employing *gain functions* to represent the value of a secret to the attacker, subsumes other orders (including min leakage).

Our work is aimed at establishing workable approaches to analysing information leakage that arises in real systems. We thus take an experimental, bottom-up approach. We broadly follow the existing consensus, recognising that a worst-case measure such as min leakage is necessary to give a truly worst-case bound. We nevertheless see that Shannon capacity is desirable, in that it matches the empirically observed behaviour of systems, and can be used to give safe bounds if appropriate corrections are made. Where appropriate however, such as for the remote guessing attack we present in Chapter 3, we measure min leakage rather than Shannon capacity.

2.9 Summary

In this chapter, we have laid out our security model. We note that side channels and covert channels are simply two manifestations of the same underlying problem, and that by attacking the mechanisms, we can eliminate both. We

establish the *guessing attack* as our principle threat model, limiting our scope to systems where it is possible for the attacker to compromise the system through brute force, although the addition of side-channel leakage makes its job easier. We measure vulnerability using the chance of compromise over a fixed interval, rather than the expected time to compromise, as it has a clearer interpretation when we are concerned with a nontrivial chance of compromise in a comparatively short period. This contrasts with the usual approach in cryptography which is concerned with the abilities of a well-resourced attacker operating over comparatively long timescales.

We note that we are concerned with practical, and often imperfect, implementations. We want to *quantify* the extent to which the implementation diverges from the specification; hoping, of course, to keep this small. We lay the groundwork for evaluating countermeasures that aim to eliminate this extra, implementation-dependent leakage, without compromising the correct operation of the system. To this end, we divide the information leakage from a system into *intrinsic* leakage—that portion that is demanded by the specification, as a consequence of correct operation, and *extrinsic* or *side-channel* leakage, that is solely due to implementation (or hardware) details. The sum of these components is the *total leakage*. We judge a system by its *differential vulnerability*—the difference between its vulnerability considering total leakage, and its vulnerability due to intrinsic leakage alone.

As a motivating example, we detail the timing channel that arises in a simple password checker, as a result of the non-constant runtime of `strcmp`. This example, while simple, displays sufficiently challenging features to require the full weight of our techniques to analyse: it has strong intrinsic leakage—the yes-or-no response; the optimal attack is dynamic—the attacker must update its guessing order as it observes the responses; and the leakage depends on both the secret, and the attacker-supplied input. We are able to model the differential leakage quite simply, needing to know only the capacity of the side channel, and the intrinsic leakage curve. We have also established that a model such as this, of *expected Shannon entropy* **can** be used to place a safe bound on one-guess vulnerability. We proved the accuracy of our *vulnerability correction*, and established that seeking to bound channel (Shannon) capacity is still a productive approach to eliminating leakage, with the benefit that it lets us build on standard information theory. We nonetheless recognise that for a true worst-case accounting, the min entropy and min capacity approach

is necessary—our insight is that in practical cases it is often unnecessarily pessimistic.

3 Practical Countermeasures

This chapter presents joint work. The original implementation of cache colouring in seL4 is due to Johannes Schlatow, under the supervision of Gernot Heiser. The implementation of instruction-based scheduling in seL4 is joint work between Qian Ge, under the supervision of Kevin Elphinstone, and myself. This work is the subject of the following paper:
David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile; an empirical study of timing channels on sel4. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 1–12, Scottsdale, USA, November 2014. ACM. doi:10.1145/2660267.2660294. (to appear)

In this chapter, we take the lessons of Chapter 2 and apply them to real systems. We analyse a pair of local channels: the *cache contention channel* and the *bus contention channel*, and a remote channel: the lucky thirteen attack of AlFardan and Paterson [2013]. For each channel we make an exhaustive empirical analysis on five different platforms, and for each we calculate the appropriate leakage measure, as established in Chapter 2: Shannon capacity for the local channels and min leakage for the remote.

We evaluate the effectiveness of three mitigation strategies. For the cache channel we analyse *cache colouring* (CC), which mitigates the channel by partitioning the cache between domains, and *instruction-based scheduling* (IBS), which attempts to prevent exploitation of the channel by removing clocks. We also examine the effectiveness of IBS against the bus contention channel, where colouring is inapplicable. For the remote channel, we propose a novel countermeasure, *scheduled delivery* (SD), which uses OS mechanisms to effec-

tively mitigate the channel with better performance and lower overhead than the existing state-of-the-art solution.

The empirical results suggest that while mitigation for local channels is broadly possible, it is often undermined by subtle (and undocumented) architectural quirks, which are becoming steadily more difficult to manage in recent processors. The capacity calculations in this chapter, both with and without mitigation, also provide the data to instantiate the leakage models of the previous chapter.

In general, we attempt to have as little performance impact as possible, and thus we avoid solutions that require expensive operations, where possible. For example, the cache channel could be mitigated by flushing all cache levels (and all other transient processor state) on every context switch. While effective, the performance cost would be prohibitive.

We consider an optimisation, which is only applicable to systems using a lattice-based classification scheme, in Chapter 5: *lattice scheduling*. This approach provides complete isolation (as the cache is always flushed between high- and low-classification domains), but still requires regular cache flushes, in addition to only being applicable to a restricted class of systems. In this chapter we focus on more general-purpose mechanisms, and leave lattice scheduling as an example in which we are able to *prove* security, rather than having to measure it. Lattice scheduling is a case in which we can formally verify the complete security result, while the results of this chapter provide the parameters for the analytic formulae just presented.

3.1 Experimental Setup

As stated, this chapter presents the results of an extensive empirical evaluation. We instrumented the five platforms shown in Table 3.1, including chips released between 2005 and 2012, with two instruction set architectures (ARM and x86), from four different manufacturers. The processors used also represent a dramatic spread of architectural complexity, from the statically-scheduled in-order ARM1136, to the fully out-of-order, speculative, multicore Conroe.

For each combination of platform, channel and countermeasure (for the local channels), we added a job to the existing seL4 regression-test setup that would take a few hours' observations early each morning, while the test

Processor	iMX.31	E6550	DM3730	AM3358	Exynos4412
Manufacturer	Freescale	Intel	TI	TI	Samsung
Architecture	ARMv6	x86-64	ARMv7	ARMv7	ARMv7
Core type	ARM1136JF-S	Conroe	Cortex A8	Cortex A8	Cortex A9
Released	2005	2007	2010	2011	2012
Cores	1	2	1	1	4
Clock rate	532 MHz	2.33 GHz	1 GHz	720 MHz	1.4 GHz
Timeslice	1 ms	2 ms	1 ms	1 ms	1 ms
RAM	128 MiB	1024 MiB	512 MiB	256 MiB	1024 MiB
L1 D-cache					
size	16 KiB	32 KiB	32 KiB	32 KiB	32 KiB
index	virtual	physical	virtual	virtual	virtual
tag	physical	physical	physical	physical	physical
line size	32 B	64 B	64 B	64 B	32 B
lines	512	512	512	512	1024
associativity	4	8	4	4	4
sets	128	64	128	128	256
L2 cache					
size	128 KiB	4096 KiB	256 KiB	256 KiB	1024 KiB
line size	32 B	64 B	64 B	64 B	32 B
lines	4096	65,536	4096	4096	32,768
associativity	8	16	8	8	16
sets	512	4096	512	512	2048
colours	4	64	8	8	16

Table 3.1: Experimental platforms.

platforms were generally unused. In this way we accumulated around 1000 hours of observations over a six month period, and 4.7GiB of compressed sample data.

The Channel Matrix

From the data for a particular combination we construct a plot, such as that in Figure 3.1, that summarises the effect of the sender's actions on the receiver's observations. In this instance, the plot shows the number of cache lines touched by the receiver in a fixed interval (on the vertical axis), if the sender manages to evict a given fraction of the cache (on the horizontal). The shading at a point indicates the conditional probability of that observation given that input (on a log scale). For instance, if the sender evicts 512 lines, the receiver will most likely touch around 8000. This figure is the *channel matrix* introduced in Section 2.6, from which we calculate channel capacity, in this case 4.25b.

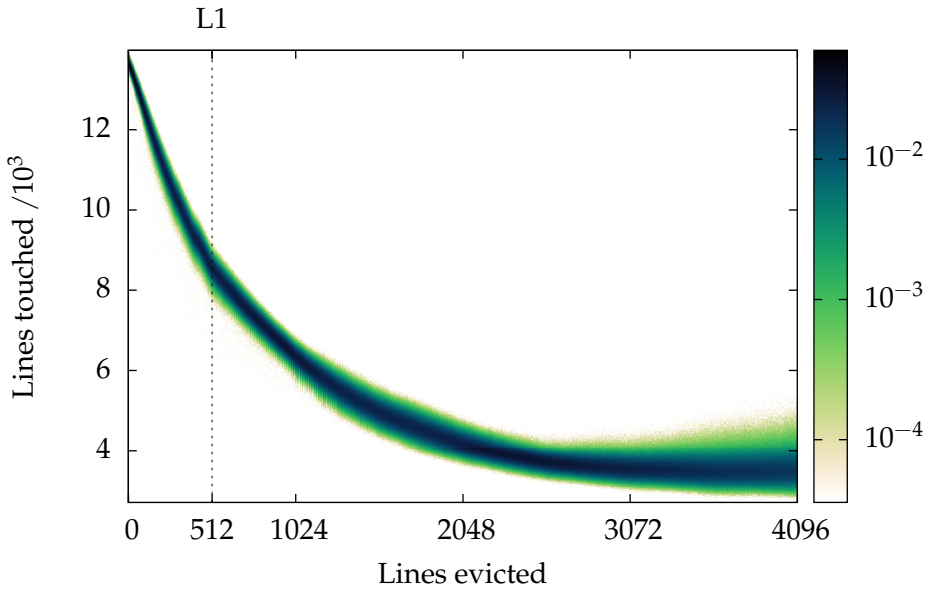

Figure 3.1: iMX.31 cache channel, no countermeasure. C = 4.25b. 7000 samples per column.

From the figure it is clear that evictions by the sender reduce the receiver's rate of progress.

To calculate the capacity, we use an improved form of the Blahut-Arimoto algorithm (ABA) [Arimoto, 1972; Blahut, 1972], due to Yu [2010]. Briefly, this is an iterative optimisation algorithm that converges on the capacity, and the input distribution that achieves it. Recall from Section 2.6, that the capacity is defined as the maximum mutual information between the input and output distributions, over all possible input distributions. The algorithm starts with any valid distribution (we take a uniform one) and, at each step, produces a new input distribution with a greater mutual information, together with an upper bound on the capacity. The lower bound (the observed mutual information) and the upper bound converge monotonically (to the limit of the arithmetic precision), and thus calculating the capacity to any desired precision (upper bound minus lower bound) simply requires taking a sufficient number of steps.

Due to the large number of input and output symbols (columns and rows), the channel matrices themselves become very large—that for Figure 3.10 would occupy 61GiB if stored densely. We take advantage of the sparseness

of the matrices: for any given input there are a relatively small number of outputs that we observe with non-zero probability. Even so, the largest of our matrices still occupies 379MiB, even using single-precision floating point to store its entries.[1] We have developed our own set of optimised sparse-matrix implementations of this standard algorithm, which are publicly available.[2]

The capacity calculations tend to be numerically unstable: with a large symbol alphabet, some symbols are assigned vanishingly small probabilities, which underflow to zero, but which make a disproportionate contribution to entropy (recall that the entropy is the sum of terms $-p \log_2 p$, and note that the logarithm grows as p shrinks). Compounding this is the loss of precision inherent in floating-point logarithms, occasionally preventing the algorithm converging to the needed precision. In most cases the arithmetic precision we achieve is orders of magnitude smaller than the statistical precision, and is thus ignored. In the few results where there is at least one significant figure of arithmetic imprecision, we mark it with the following notation: $x\,{}^{y}_{z}$, meaning that the notional value x is bracketed by the upper and lower numerical bounds y and x.

Residual Channels

While the existence of a channel in Figure 3.1 is obvious, for Figure 3.2 the answer is not straightforward. The figure is visually uniform, with no apparent correlation between lines evicted and lines touched. However, we nonetheless calculate a small but non-zero capacity of 2.14×10^{-2}b. Is this a real, residual channel, or is it just statistical noise?

Note that even if there really is no channel (and thus the columns of Figure 3.2 are actually drawn from the same distribution), the fact that we have only a finite number of samples will lead to apparent differences in the columns, particularly in areas of low probability, where a single extra sample leads to a proportionately larger error. The effect is to make two columns appear distinguishable when in fact they are not, and thus to increase the apparent capacity.

[1] In all cases, we have fewer than 10^5 samples per column, and thus single-precision values (with 23-bit mantissa) give sufficient precision to store the conditional probabilities. Arithmetic is carried out in double precision (or longer), to avoid precision loss.

[2] `http://ssrg.nicta.com.au/software/TS/channel_tools/`, or in the attached material.

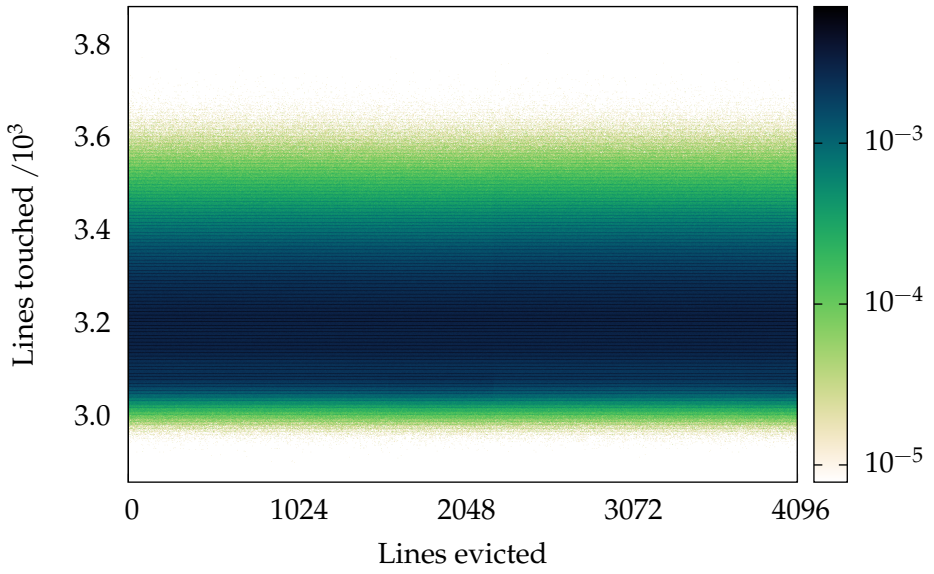

Figure 3.2: iMX.31 cache channel, partitioned. $C = 2.14 \times 10^{-2}$b, $CI_0^{max} = 1.13 \times 10^{-2}$b. 63,972 samples per column.

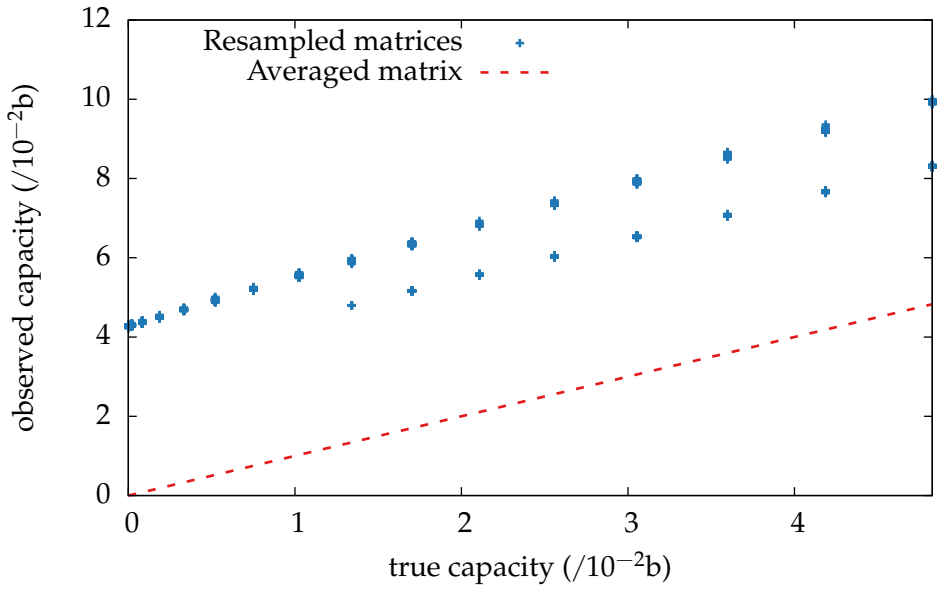

Figure 3.3: Capacity of sampled Exynos4412 cache-channel matrices (partitioned), using 7200 samples per column.

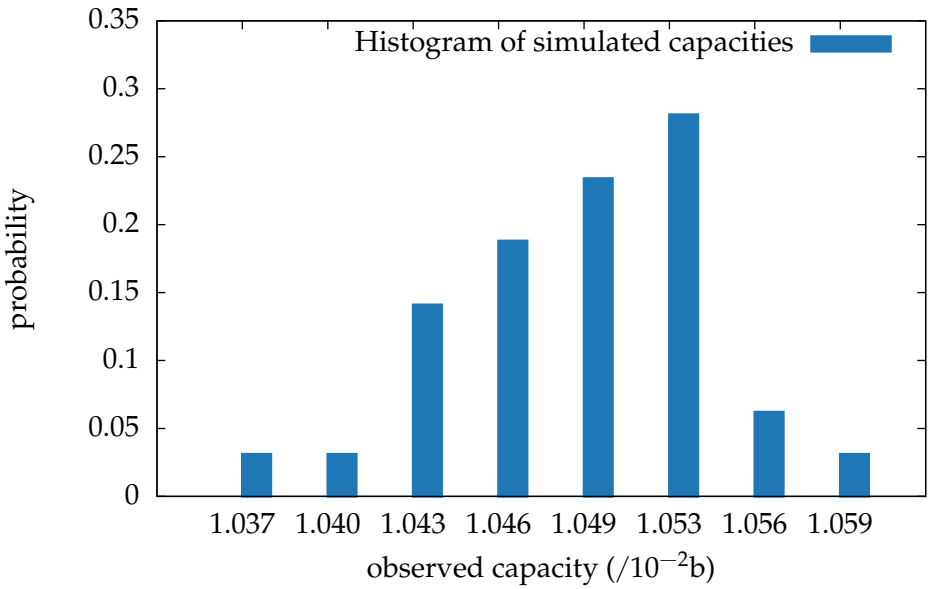

Figure 3.4: Distribution of capacities for 64 simulated zero-capacity matrices derived from Figure 3.2.

To quantify the effect, we ran a number of simulations to produce Figure 3.3 (which draws on the matrix in Figure 3.17, for colouring on the Exynos4412, itself very similar to Figure 3.2). Here we modified the real channel matrix by introducing a small discontinuity, in order to simulate the effect of a small residual channel, with a capacity on the order of 10^{-2}b. From this artificial matrix, we generated a large number of simulated observations by taking the same number of samples used to generate the original matrix (in this case 7200). For each of these we calculate its apparent capacity, plotted in blue against the real underlying capacity in red. We see that the effect of taking only 7200 samples is to increase the apparent capacity by around 4×10^{-2}b, and that this effect is consistent across the range investigated. Importantly, we see that the effect is always to *increase* the apparent bandwidth, and thus the observed capacity is always an *upper bound* on the true capacity.

From this result we develop a statistical test for residual channels: We first establish what we would *expect* to see if there were no residual channels. In this case, as already discussed, all columns of the matrix would actually have been drawn from the same distribution, and thus by averaging the columns we have a histogram of this distribution constructed from the number of sam-

ples per column *multiplied by* the number of columns. We then sample 1000 new matrices from this distribution, using the original number of samples. We now have a corpus of observations consistent with the no-residual-channel hypothesis. For each of these we calculate the capacity, and label the largest $\mathrm{CI}_0^{\max}$, or the largest value in the 99.9% confidence interval for the null hypothesis. If the observed capacity is greater than $\mathrm{CI}_0^{\max}$, our test suggests that there is at most a 1 in 1000 chance of it being produced given no residual channels, which is strong evidence that such a channel exists, even if it is not apparent in the figure. Note that if the capacity is less than $\mathrm{CI}_0^{\max}$, this does *not* imply that there is no channel—in this case the test is simply inconclusive.

Figure 3.4 show the distribution of observed capacities for simulated zero-capacity matrices generated from Figure 3.2, binned at 3.2×10^{-5}b. Of 64 simulated matrices, only two display an apparent capacity of at least 1.059×10^{-2}b and thus, if the original matrix has greater capacity than this (it does), we infer that there is only a 3% chance that this is a statistical fluke. The larger number of samples used in the real test allows us to improve this to 0.1%.

Returning to Figure 3.2, we see that the observed capacity of 2.14×10^{-2}b is greater than $\mathrm{CI}_0^{\max}$ (1.13×10^{-2}b), and thus we infer that a residual channel exists. This channel is due to TLB contention, as we demonstrate in Section 3.3.

3.2 The Local Channels

Before considering mitigation strategies, we first establish the behaviour of the unmitigated local channels: cache contention and bus contention.

The Cache Contention Channel

Modern processors invariably employ caches to compensate for the fact that memory access times are now orders of magnitude greater than cycle times. At any point in time, some subset of memory is held close to the CPU in fast but expensive (and thus small) caches. These caches are usually divided into fixed-sized blocks called *lines* (often of 16–64B) which store blocks of the same size loaded from memory.

Any two processes that execute concurrently (either truly, on a multiprocessor, or time-sliced on a uniprocessor) compete for space in the processor's caches. There are only a finite number of cache lines and thus loading data

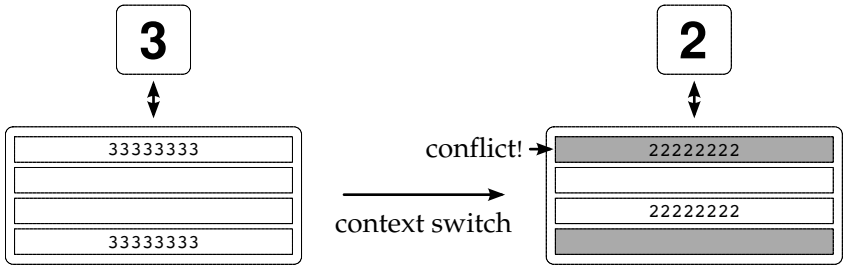

Figure 3.5: The cache-contention channel.

from one process generally implies evicting that of another. One process can deliberately evict the lines of another process, by touching appropriate parts of its memory. Figure 3.5 illustrates the effect, where process 3 has executed for some time, filling a fraction of the cache with its own data. Once a context switch (to process 2) occurs, process 3's data is no longer accessible (greyed out in the figure), although it remains resident. Process 2 then begins to fill the cache with its own data, eventually attempting to fill the same line as 3 did. At this point process 3's line will be *evicted*.

The time taken to execute a memory load or store varies dramatically depending on whether the data is cached and, if so, in which cache (L1, L2, ...) it lies. For example, on the Intel Core architecture (as implemented in the E6550) [Intel 64 & IA-32 AORM, §2.2.5, table 2-16], a load that hits in the L1 data cache has a latency of 2 cycles, one that misses in L1 but hits in L2 has a latency of 14 or 15 cycles, while an access to the L3 cache takes approximately 110 cycles (the manufacturer does not make precise guarantees). An access that misses in all caches and thus goes to main memory can easily take thousands of cycles to complete. This difference is easily measurable, especially if a large number of accesses are executed together and the total time is observed. One process (the sender) can thus cause an effect (by evicting cache lines) that is measurable by another (the receiver), even if all explicit communication channels are removed.

This is a hardware-mediated *timing channel*.

The receiver, of course, needs a clock with which to measure the effect of these cache misses. We assume that if the system designer is concerned with covert channels, then obvious clock sources (any sort of `clock()` system call, for example) have been eliminated. Therefore, to construct a sample exploit, we take advantage a of more subtle timer, that is almost universally available,

```
1  /* TX */
2  char A[LINES][64]; int S;
3  while(1)
4    for(i=0;i<S;i++) A[i][0] ^= 1;
5
6  /* RX */
7  char B[LINES][64]; volatile int C;
8  void probe(void) {
9    while(1) {
10     for(i=0;i<LINES;i++) B[i][0] ^= 1;
11     C++;
12   }
13 }
14
15 void measure(void) {
16   int R, C1, C2;
17   while(1) {
18     C1= C;
19     do { C2=C; } while(C1==C2);
20     R=C2-C1;
21   }
22 }
```

Figure 3.6: Code to exploit a cache channel.

and often of very high accuracy: the *preemption tick*.

Imagine that we are executing the code of Figure 3.6 on a single-processor system, with the code under `TX` (lines 3–4) in the sender, and both functions under `RX` (`probe()` lines 8–13, and `measure()` lines 15–22) in the receiver, each in a separate thread. The sender varies its working-set size under the control of the input variable `S` (for send)—modifying a varying number of elements of the array `A`. Each modification ensures that one cache line is both resident and dirty (this example assumes 64B lines). The receiver walks endlessly over its array, `B`, dirtying every line, and recording its progress in the variable `C`. The two arrays are sized to entirely cover the addressable range of the cache. The receiver is entirely memory-bound, and number of lines that it manages to touch in a given interval depends on how often it hits in the cache, and thus on how aggressively the sender is evicting its cache lines.

To measure the receiver's progress we employ a second thread, executing

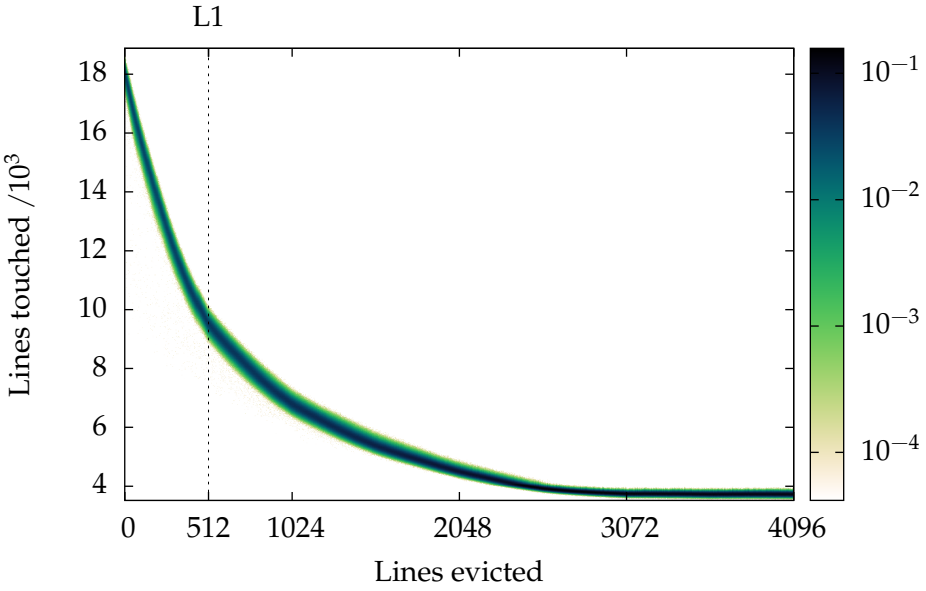

Figure 3.7: AM3358 cache channel, no countermeasure. C = 4.69b. 6000 samples per column.

the function `measure()`. This thread watches the value of the counter, `C`. For as long as it executes (and hence prevents either the sender or the receiver from executing, as all are executing on the same processor core), the value remains constant. A jump in the counter's value indicates that it has been preempted, and the receiver thread (and thus also the sender, if the scheduler is round-robin) has executed. The size of the jump gives the number of cache lines that the receiver managed to touch during a single timeslice. Preemption (and thus timeslice length) is typically driven by a regular timer interrupt (often at 1000Hz on a modern system, giving a timeslice of 1ms). The receiver is able to accurately measure its rate of progress, `R` (for receive), from which it infers the value of the input variable, `S`.

Figure 3.1 gave the result of exploiting this channel (under seL4) on the iMX.31 SoC, while Figure 3.7 and Figure 3.8 show the same channel on a pair of closely-related chips: the AM3358 and the DM3730, both based on the Cortex A8 core and with 256kiB L2 caches divided into 4096 64B lines. The only difference between these is the somewhat higher values in Figure 3.8, due to the DM3730's higher clock rate (1GHz versus 720MHz). In all these matrices, we have flushed the L1 cache on every context switch, to isolate

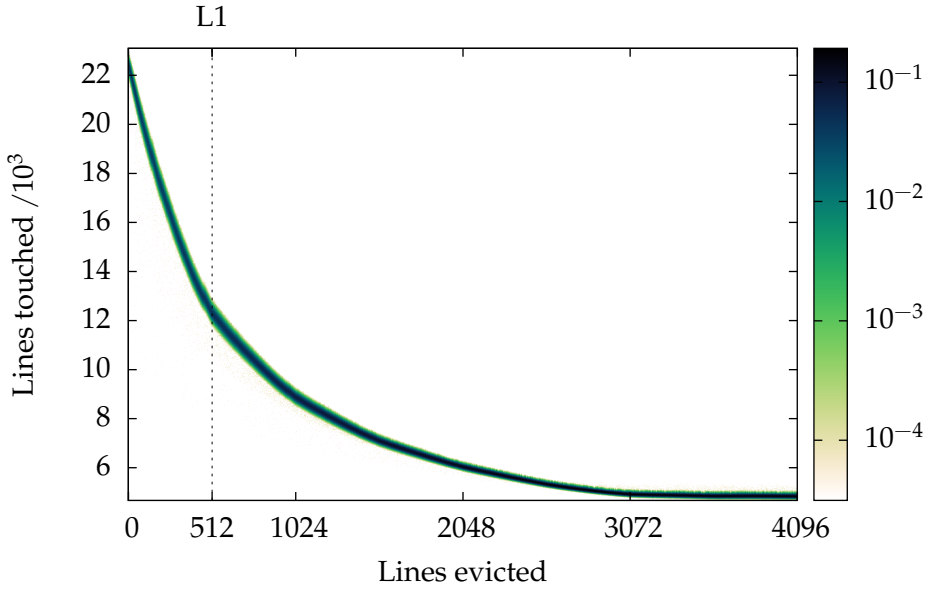

Figure 3.8: DM3730 cache channel, no countermeasure. C = 5.28b. 8000 samples per column.

the effect of L2 contention. We will see that this flush is required for cache colouring in any case, in Section 3.3.

The number of lines evicted by the sender, the input variable S—its working set size, varies along the horizontal axis, while the number touched by the receiver, the output variable R, along the vertical. The colour at point $\{s, r\}$ gives the conditional probability of seeing r lines touched by the receiver, given that s were evicted by the sender. Intuitively, the more the figure varies from left to right, the more easily the receiver can distinguish two different input values, and the higher the channel capacity.

The calculated capacity of 4.69b for Figure 3.7, and of 5.28b for Figure 3.8 translate into usable bandwidths of 1.56 and 1.76kB/s respectively, when multiplied by 333Hz (the rate at which the ensemble of three threads complete one cycle of preemptions). A higher bandwidth can obviously be achieved by using a higher sample rate: On a uniprocessor, the useful sampling rate is limited by the preemption rate (1000Hz) but on a multiprocessor (or in the case of simultaneous multithreading), the sampling rate is essentially unlimited as the sender and receiver need only execute a single memory access each in order to communicate. In this case the channel might be exploited at a

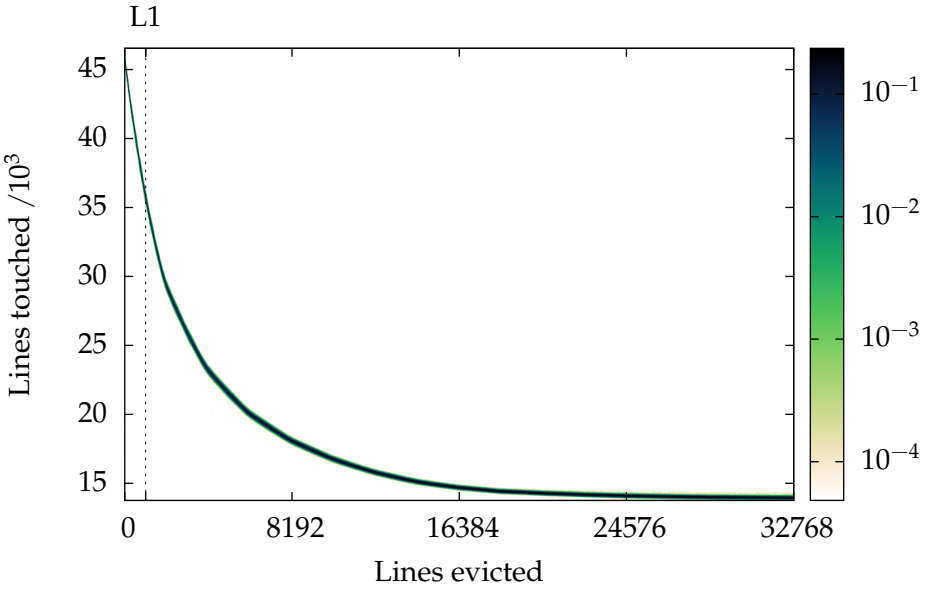

Figure 3.9: Exynos4412 cache channel, no countermeasure. C = 7.04b. 1000 samples per column.

very high bandwidth (megabytes per second, at least), but the *capacity*, or the number of bits transferred per sample will not increase. In fact, the capacity is likely to decrease somewhat, as the effect of noise will become greater as the range of variation that the sender exploits becomes smaller. We thus note that the quantity of importance is the capacity (in bits), and that the bandwidth (in bits per second) depends on the maximum sampling rate, which is generally out of our control. However, reducing the capacity will reduce the bandwidth. From now on therefore, we quote only the channel capacity.

The next plot, Figure 3.9, is for the Samsung Exynos4412, a 4 core ARM Cortex A9. This chip has a significantly larger L2 cache than any of the other ARM SoCs tested, at 1024kiB. It also returns to the 32B cache lines of the ARM11, giving 32,768 lines. As we see, this leads to the substantially increased bandwidth of 7.04b, as the sender now has finer-grained control over its degree of interference. A further increase in clock speed (to 1.4GHz), coupled with cache lines that are half as long, is again sufficient to explain the increase in the peak rate from 22×10^3 to 45×10^3 lines touched per preemption.

Our final unmitigated channel matrix, Figure 3.10, is for the Intel Core 2 Duo E6550. This is much larger and more powerful than any of the ARM

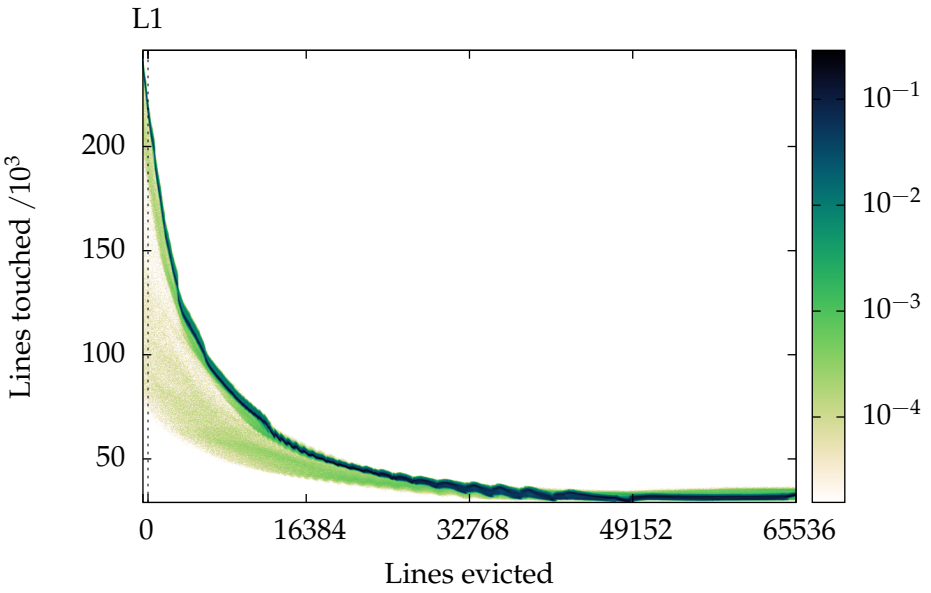

Figure 3.10: E6550 cache channel, no countermeasure. C = 8.82b. 1000 samples per column.

cores we have considered, and has a commensurately larger L2 cache, at 4MiB. Cache lines are 64B. The peak work rate is significantly higher, at $\approx 250 \times 10^3$, which the increase in clock speed alone (from 1.4GHz to 2.33GHz) is insufficient to explain. This core clearly has a significantly higher load/store capability, and a greater L2 bandwidth to match. The result is a capacity of 8.82b.

Figure 3.11 summarises the unmitigated channel results: The capacity of the L2 cache-contention channel scales roughly with the logarithm of the number of L2 cache lines. The confounding factors in this plot are the varying clock speeds (which, in fact, increase monotonically with cache size for the cores we consider), and that the x86 kernel has a minimum preemption interval of 2ms, versus the 1ms of the ARM kernel, which would tend to slightly overstate the capacity for the E6550. The dotted line is a linear model, fitted to the E6550.

The Bus Contention Channel

The second local channel that we analyse is one that only appears in systems with multiple processors: *bus contention*. In a multiprocessor system, the processors will generally share not only some levels of the cache hierarchy

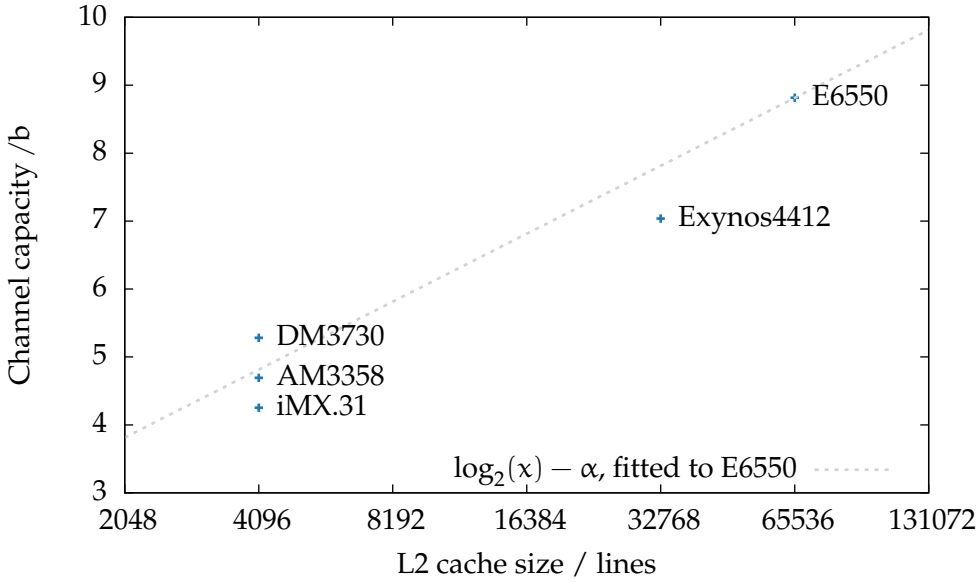

Figure 3.11: Channel capacity against L2 cache size in lines.

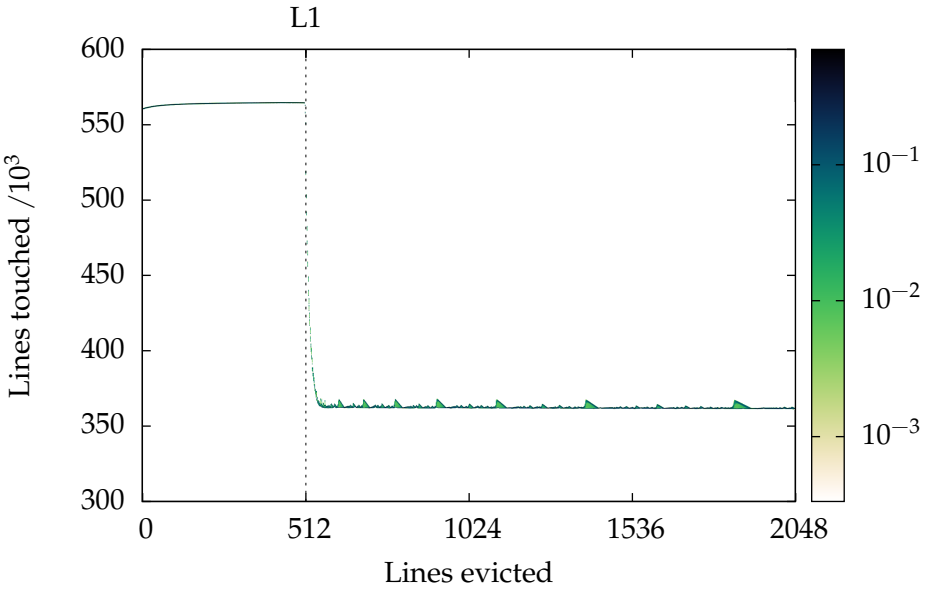

Figure 3.12: E6550 bus channel, no countermeasure. $C = 5.80^{5.81}_{5.80}$b. 3000 samples per column.

(usually at least the last level), but they must also compete for bandwidth on system buses, e.g. to the memory or to peripherals. Thus, even if two processes execute on separate processors, share no memory or other explicitly allocated resources, and do not compete for cache lines, they can still interfere with each other by monopolising bus cycles, which are typically underprovisioned relative to the combined peak requirements of all processors.

Figure 3.12 shows the effect of contention between two processors, again executing the code of Figure 3.6, but this time with the sender and receiver each executing on one of the two separate cores of the E6550. On this chip, the two cores have separate L1 caches, but share a unified L2. The point at which the two processors begin competing for access to the L2 cache (and memory) is clear in the figure, occurring as soon as the sender spills outside its L1 cache. Here the interference is not due to competition for space, as the both sender and receiver are restricted to only 2048 (out of 65,536) cache lines, and thus see essentially no capacity or conflict misses. The effect is rather due to the fact that the two cores combined are able to issue more load and store requests (per second) than the L2 cache is capable of performing, and thus an increase in the number issued by the sender reduces the number available for the receiver. The capacity of this channel is between 5.80 and 5.81b (our numerical precision in this example was limited to three significant figures).

This channel is obviously only measurable on the two multiprocessor platforms (the E6550 and the Exynos4412), of which only the E6550 had a working multiprocessor port of seL4 when our experiment began.

3.3 Cache Colouring

The first countermeasure that we consider, *cache colouring*, eliminates the cache contention channel by exploiting a widespread feature of modern caches, *set-associativity*, to partition the cache between multiple domains.

Figure 3.13 illustrates the technique. Imagine a machine with a 256B physical address space addressed by an 8b word. In this machine, the memory is divided into 16 frames, each of 16B. This is a simplified example, but the principles are identical for systems with much larger address spaces.

As already described, the cache is divided into *lines*, here of 4 bytes each, which are always a size-aligned block of adjacent bytes. The line is the smallest addressable unit in the cache—loads and stores are always of a multiple of

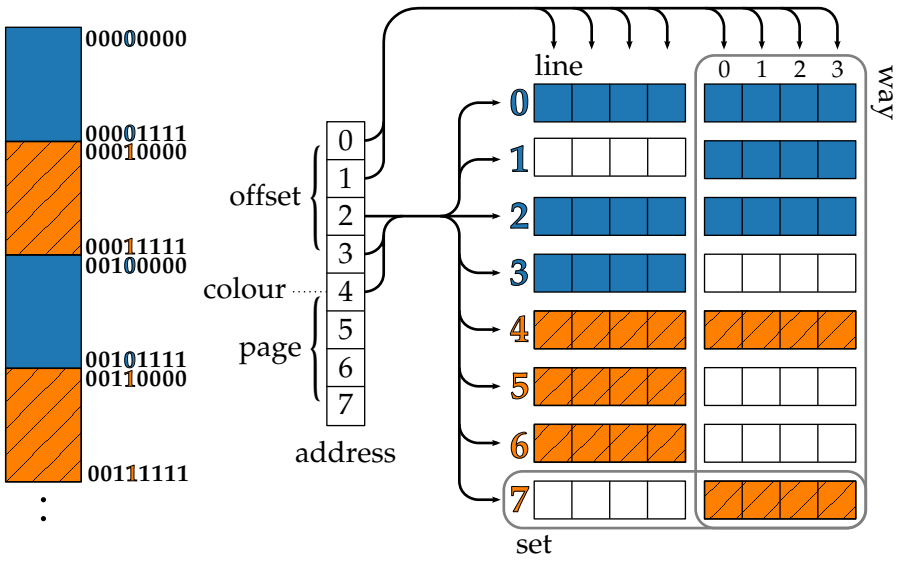

Figure 3.13: Cache colouring.

the line size. Our system has 16 cache lines, for a total of 64 bytes of cache. The bytes within a line are indexed by the lowest bits of the address words, in our case bits 1 and 0.

For each load or store, the cache is searched to see if it holds the requested address, and signals a miss if not. There are several approaches, the simplest being *direct mapping*: the line number is taken directly from the next lowest bits after the line offset. In our example, 16 lines require 4 address bits, and we thus use bits 5–2. While simple to implement, a direct-mapped cache has a substantial disadvantage: all addresses that are equal modulo the cache size alias, and loading any will evict any other that is resident. Direct-mapped caches suffer from a high rate of *conflict misses*.

The opposite extreme is the *fully associative* cache. Here, the address is compared in parallel against every line, and a given line may thus be stored anywhere, irrespective of its address. A fully associative cache never suffers conflict misses, as *any* set of lines (of up to the cache size) may be co-resident. The disadvantages of the fully associative cache are that it is expensive to implement, power hungry and the parallel match becomes slower as the cache becomes larger. This is important, as the match is in the critical path.

The usual compromise between these is the *set-associative* cache. Here cache lines are divided into a number of associative *sets*. The set number is

calculated by direct mapping, while the lookup within each set is associative. This takes advantage of the speed of direct mapping, while still allowing some degree of co-residency for aliased addresses. The size of the sets, or associativity, is the number of *cache ways*. For example, the iMX.31 has a 4-way associative L1 data cache of 512 lines, and thus has 128 direct-mapped sets. In contrast, the L2 cache of the E6550 is 16-way associative, with 65,536 lines, giving 4096 sets. Higher associativity is usually seen in outer-level caches, where latency is less critical.

Returning to our example, our 2-way associative, 16-line cache has 8 sets. These are indexed by the next 3 bits after the line offset: 4–2. The 2-way associativity has reduced the number of direct-mapped bits from 4 to 3 (2^n way associativity eliminates n direct-mapped bits). Any contiguous set of $2^{3+2} = 32$ addresses will map one 4-byte line into every cache set. The important point is that this range, the *direct-mapped range*, is larger than the page size of 16 bytes. Therefore, any adjacent pair of pages will map onto disjoint ranges of cache lines, as shown in Figure 3.13. We thus divide the physical address space into pools of *coloured pages*, such that no two pages from different pools can collide in the cache. If processes are assigned to distinct pools (using the virtual-memory system to map these non-adjacent *frames* into a contiguous range of virtual *pages*), then the cache channel between them is removed.

We can only colour memory when the size of the direct-mapped range is at least two pages, and the number of colours available (the size of the range divided by the smallest page size) varies greatly between platforms. For the L2 cache, there are 4 colours available on the iMX.31, and 8, 8, 16 and 64 for the AM3358, DM3730, Exynos4412 and E6550, respectively. We cannot colour the L1 cache, as on every platform it either has only a single colour, or is virtually-indexed. We instead flush it on every context switch. The cost of this is relatively small, given its small size.

Turning to our results, Figure 3.2 showed the result of applying two-way cache colouring on the iMX.31, and Figure 3.14 and Figure 3.15 give the equivalent result for the two Cortex A8 cores, the AM3358 and DM3730. In contrast with the unmitigated channels in Figure 3.1, Figure 3.14 and Figure 3.15 respectively, all visible variation has been eliminated. The capacities drop by a factor of 200, 400 and 1000 respectively, to on the order of a hundredth of a bit. Note that all three results fail the statistical test introduced in Figure 3.1,

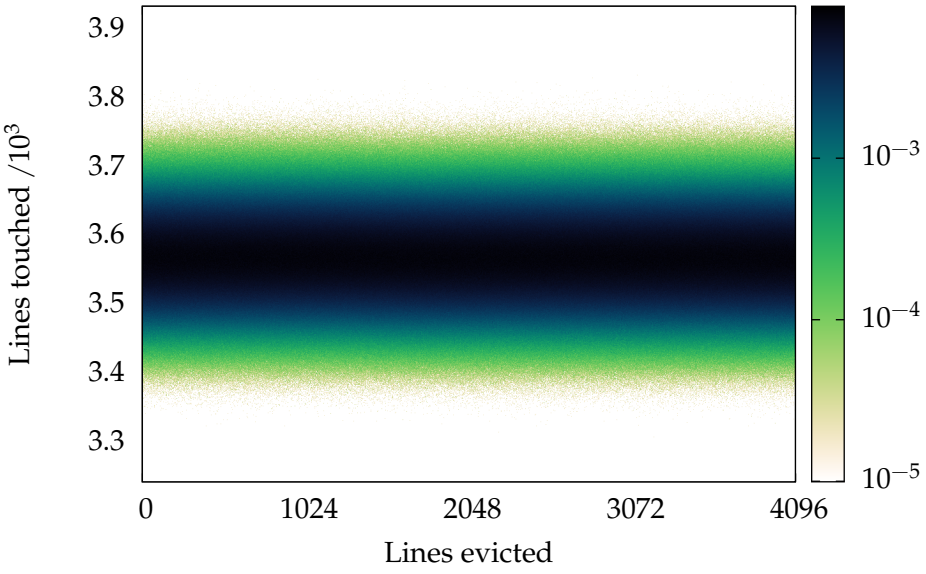

Figure 3.14: AM3358 cache channel, partitioned. $C = 1.51 \times 10^{-2}$b. $CI_0^{max} = 8.10 \times 10^{-3}$b. 49,600 samples per column.

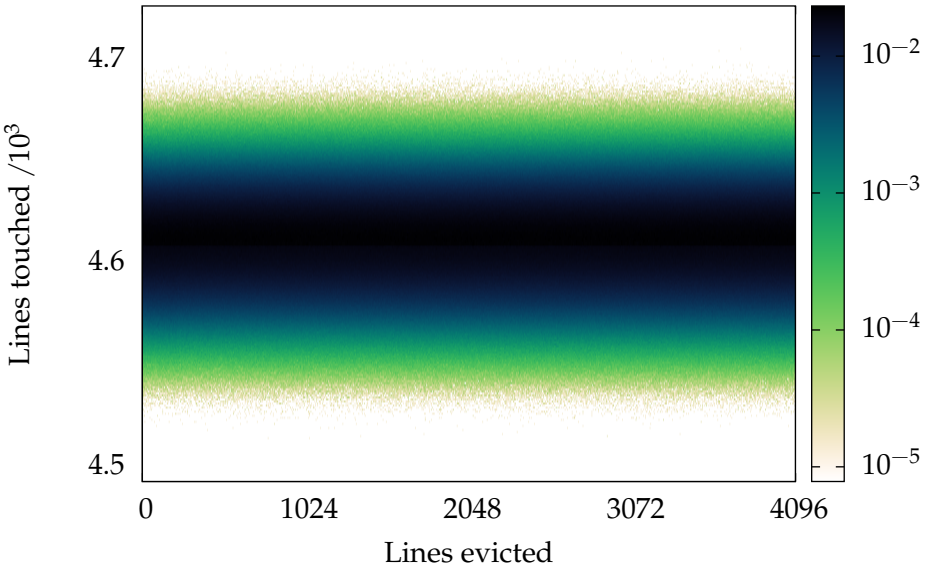

Figure 3.15: DM3730 cache channel, partitioned. $C = 5.02^{5.02}_{5.01} \times 10^{-3}$b. $CI_0^{max} = 2.75 \times 10^{-3}$b. 63,200 samples per column.

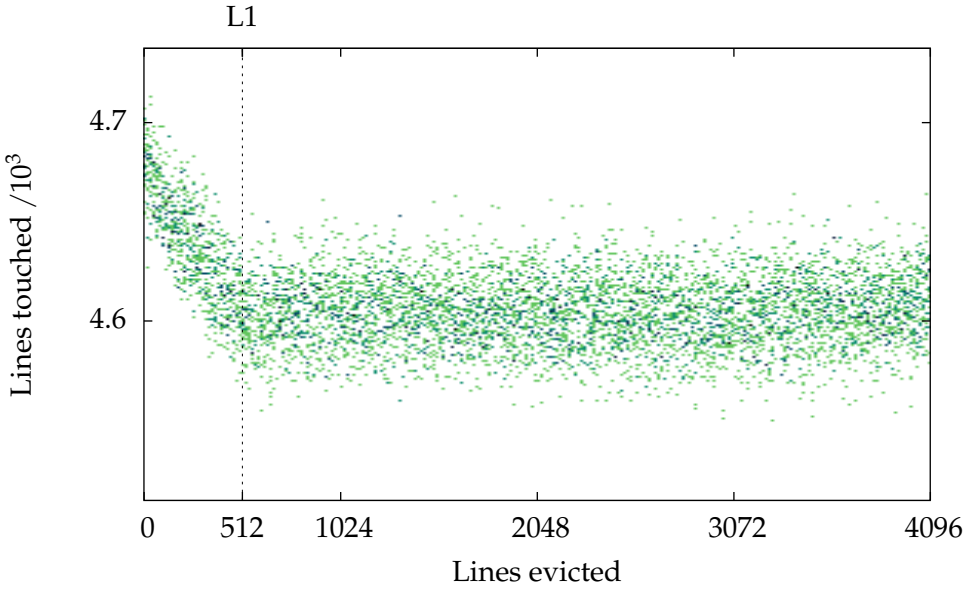

Figure 3.16: DM3730 cache channel, partitioned, showing L1 contention.

implying that there is a real residual channel here. We will return to this point shortly, and show that this residual channel is due to TLB contention.

Figure 3.16 shows the effect of not flushing the L1 cache on the DM3730. The number of samples is insufficient to reliably estimate capacity, but the existence of a channel is obvious.

Figure 3.17 shows the results on the Exynos4412, reducing the capacity of Figure 3.9 by a factor of less than 100, from 7.04b to 8.13×10^{-2}b. Together with the apparent capacity lying above the threshold of 4.37×10^{-2}b, it is clear that the residual channel is now seriously undermining the countermeasure.

The cause of this remaining channel is clear from the top blue curve of Figure 3.18, which plots the expected value of each column of Figure 3.17. The variation of about 5 parts in $10,000$ is obvious, and is consistent with the corresponding blue curve in the lower plot, showing the number of CPU stall cycles due to TLB misses per line touched by the receiver. A larger number of TLB misses leads to a smaller number of lines touched.

The TLB, or *translation lookaside buffer*, is a small per-processor cache of virtual-to-physical address translations, which must be consulted on every memory access, to determine the correct physical address to issue to the external memory subsystem. This cache is shared between processes and is

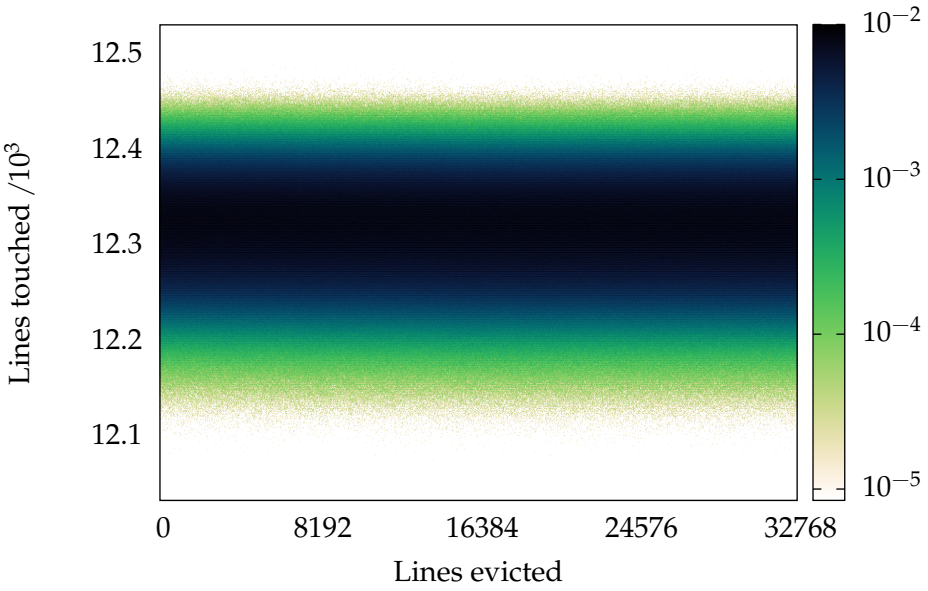

Figure 3.17: Exynos4412 cache channel, partitioned. $C = 8.13^{8.14}_{8.12} \times 10^{-2}$b. $CI_0^{max} = 4.37 \times 10^{-2}$b. 7200 samples per column.

a source of contention. Flushing the TLB when switching between domains eliminates the remaining contention, at the cost of a greater rate of TLB-induced stalls, as shown by the red curves. This channel also appears on all other ARM platforms tested, explaining all the residual capacities noted above.

The residual TLB channel only became apparent after a large amount of data was collected, and we thus do not yet have sufficient data to verify the absence of a channel (after flushing) to the level of precision represented by Figure 3.17. Figure 3.18 however, suggests that if any variation remains, it has been reduced by at least an order of magnitude.

Finally, Figure 3.19 shows the effect of colouring the L2 cache of the E6550. In contrast to the ARM examples, here there is a distinct artefact at half the cache size, which is more clearly visible in the red curve which plots the column averages, as in Figure 3.18. Here we see a clear effect at half the cache size, and another at the L1 size. The latter occurs as the x86 architecture provides no mechanism to selectively flush the L1 cache, and so in this run it was left alone on preemption, allowing the two threads to contend. Together with the half-cache artefact, this leaves us with a bandwidth of 4.54×10^{-1}b:

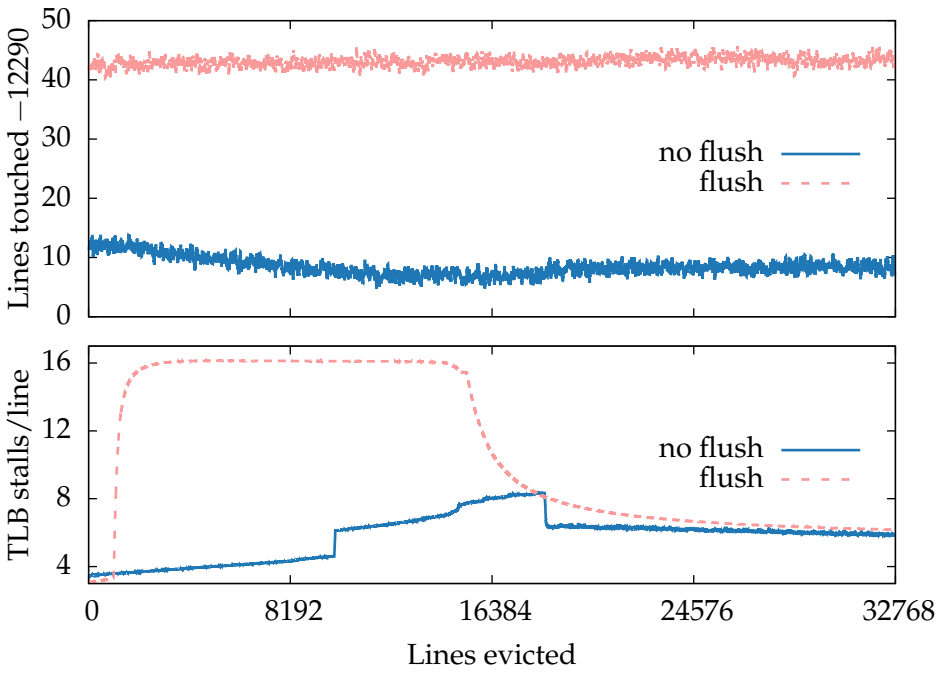

Figure 3.18: Exynos4412 TLB contention.

only a 20-fold reduction over the raw bandwidth of 8.82b.

The L1 interference can be reduced by manually overwriting the contents of the cache on a context switch, by having the kernel walk an array of a small multiple of the L1 size. This is has non-trivial performance implications, and is only somewhat effective, as the bottom-most, blue, curve in Figure 3.19 shows. It is unclear whether this residual channel can be closed on x86 without substantial performance cost, although colouring would be effective between cores that share a colourable last-level cache, but with separate L1s.

The remaining artefact (at half the cache size) appears to be due to L2 cache misses generated by the sender. To confirm this, the blue curve was generated by reducing the sender's coloured pool to cover only a quarter of the cache, rather than a half in all other examples. We see that the artefact moves neatly to a quarter of the cache size—the point at which the sender starts to cause a large number of capacity misses by evicting its own previously loaded lines.

This contention cannot directly affect the receiver, as the two are parti-

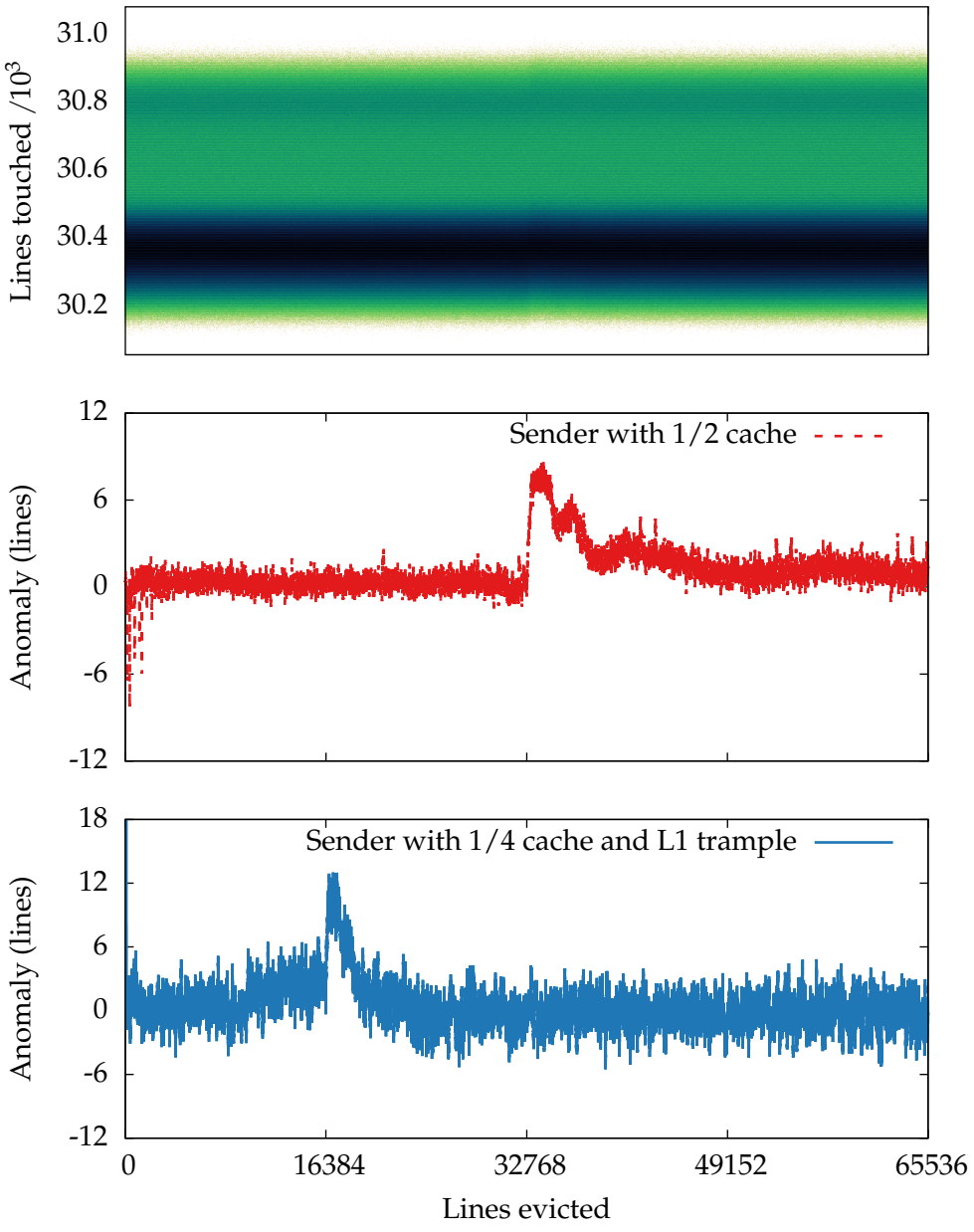

Figure 3.19: E6550 cache channel, partitioned. $C = 4.54 \times 10^{-1}$b. $CI_0^{max} = 2.53 \times 10^{-1}$b. 4836 samples per column.

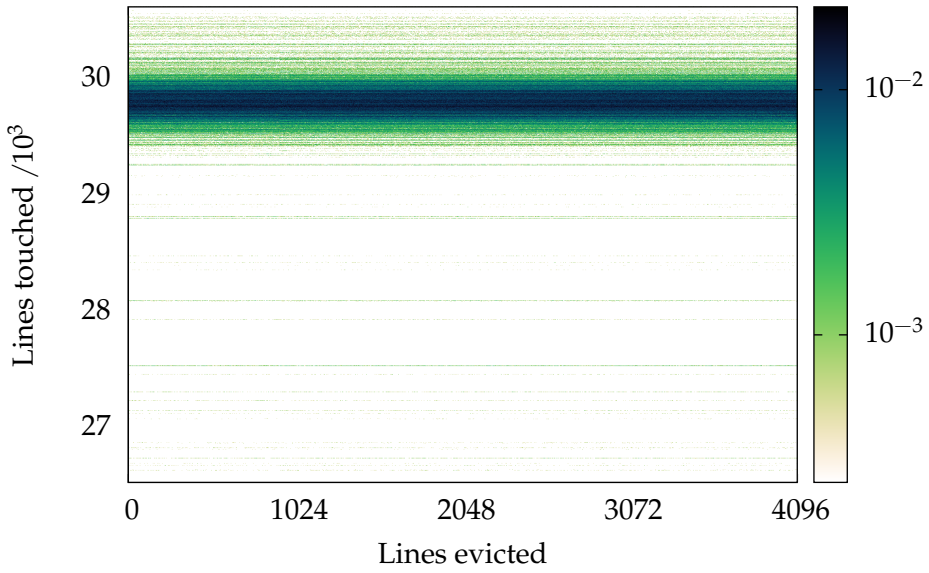

Figure 3.20: DM3730 cache channel, partitioned, with the receiver's working set restricted to 1024 lines. 2000 samples per column.

tioned, but certain cache misses in the *kernel* may still be observable. Both kernel code and data are coloured, and hence partitioned, but there is a brief interval between switching between coloured kernel images and resetting the preemption timer, where a cache miss (particularly an instruction cache miss) can affect the length of the receiver's timeslice. Specifically, the kernel resets the preemption timer, and hence starts counting time against the next thread's timeslice *before* switching kernel images. The sender can cause cache misses in this interval (as we are still executing in its colour), that will be counted against the receiver's timeslice. As the receiver uses its timeslice length to measure its progress, this will show up as an exploitable channel. This channel should be resolved by a more careful re-implementation of kernel colouring, which is underway at the time of writing.

Performance Cost

Partitioning the cache is not without a cost. The effective cache size available to any process is divided by the number of colours. If two processes were previously sharing the cache evenly, then this may have minimal impact on performance, and could even show an improvement by better isolating them

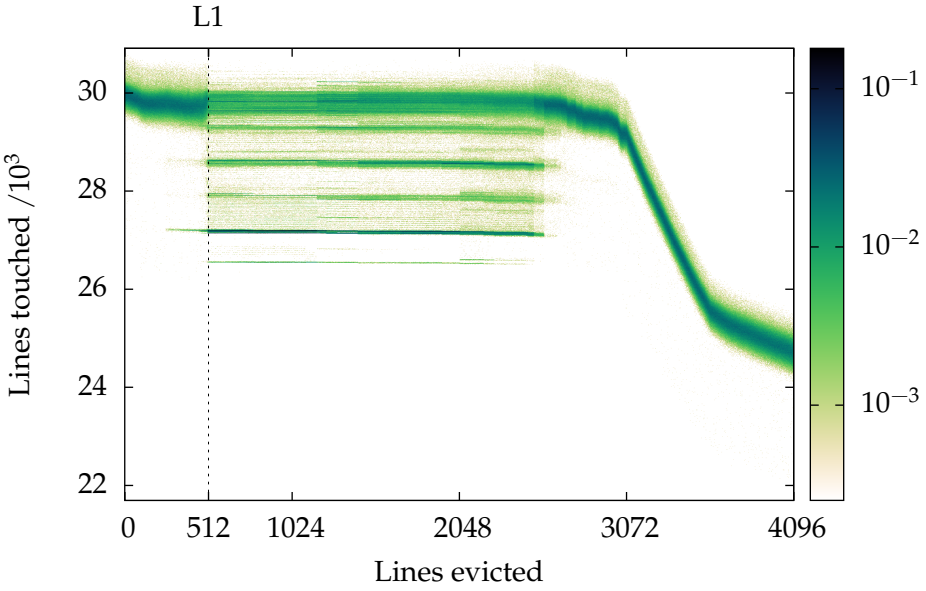

Figure 3.21: DM3730 cache channel, unmitigated, with the receiver's working set restricted to 1024 lines. 2000 samples per column.

[Tam et al., 2007]. In the case of a single process with a large working set however, we are likely to see a loss of throughput.

The receiver in our exploit represents the worst case: a process whose working set exactly matches the L2 size. Prior to partitioning, after a single walk through the array warms the cache, every subsequent access will be a hit. Halving the available cache by partitioning ensures that we will now see a miss on every access, as the process will evict all of its old cache lines before it gets a chance to reuse them. We see this in, for example, Figure 3.15, where the results cluster around 4600 iterations per timeslice, which mirrors the worst-case seen in Figure 3.8, where the receiver's miss rate was driven to 100% by the sender. The worst-case penalty is thus a 5-fold reduction in performance—consistent with simply having a smaller cache.

For a process whose working set fits within a partition, the situation is much better, as shown by Figure 3.20. Here, we have run the same receiver thread, but with its working set restricted to 1024 lines—comfortably within its partition of 2048. Here we see no loss of throughput compared to the equivalent unpartitioned results in Figure 3.21, until the sender evicts 3/4 of the cache and begins to interfere with the receiver, causing the sudden drop at

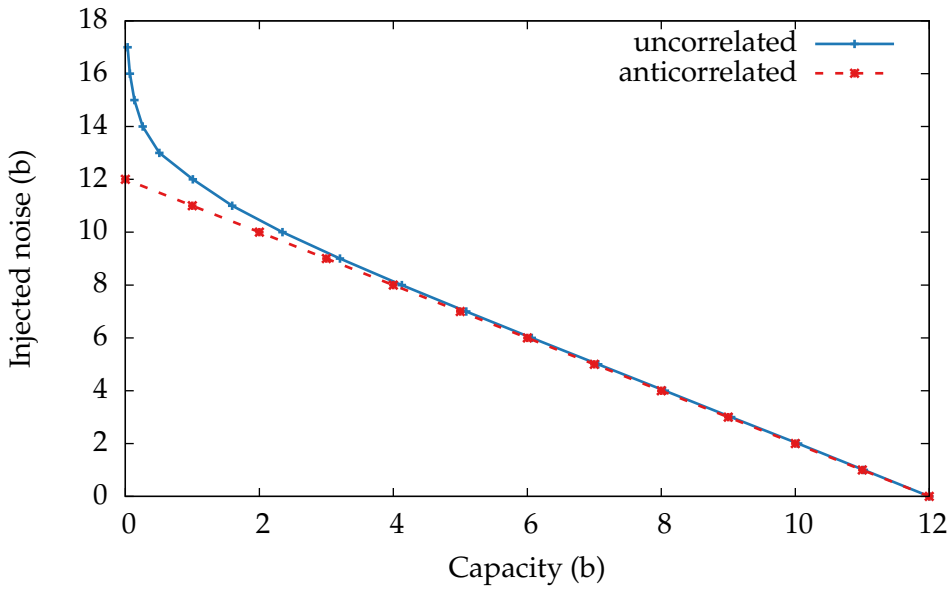

Figure 3.22: The effect of correlated versus anticorrelated noise on channel capacity.

3072 lines.

Thus in performance terms, a partitioned cache behaves as a larger number of smaller, separate caches.

3.4 Noise versus Determinism

We observed that to exploit a timing channel the receiver needs a clock. For clocks (such as the preemption tick) that cannot be easily removed, there are two basic ways to prevent their use in exploiting a channel: reducing their precision by adding noise, or reducing the variation relative to observations in the channel. Reducing variation can be viewed as adding *anticorrelated* noise, while the former approach uses *uncorrelated* noise.

Consider Figure 3.22. This plots the amount of noise required to reduce the capacity of a 12b channel to any desired level, assuming that it is either uniformly distributed and uncorrelated with the channel output, or perfectly anticorrelated with it. To reduce the capacity to half its original value takes roughly the same quantity of noise in either case. Once we start trying to reduce the capacity toward zero, however, the uncorrelated noise required

increases rapidly. In fact, this curve has a vertical asymptote at zero—no quantity of noise will completely close the channel, some signal always remains. In contrast, given 12 bits of anticorrelated noise, the signal is gone and the channel closed. Therefore, to reach small channel capacities, adding anticorrelated noise—reducing the signal, is more effective. This insight underlies the second countermeasure we analyse.

3.5 Instruction-Based Scheduling

As already noted, it is not sufficient for the sender to influence the receiver's performance—the receiver must be able to detect it. This is the essence of the *two-clock model* for a timing channel [Wray, 1991]. This abstracts from the true passage of time, requiring only the existence of two 'clocks', visible to the receiver, which can be manipulated by the sender.

A clock, in this context, need only be a sequence of *ticks*, by observing which the receiver detects that some amount of time has passed. Given only one clock, the receiver has no idea *how much* time has elapsed since the last tick, and if the ticks are not at a constant interval, this alone gives the receiver no information.

Once the receiver has a *second* clock, it can use its ticks to measure the intervals between those of the first—the receiver extracts information from the *difference* in the rate of the two clocks. If one clock *does* run at a constant rate (although the receiver, of course, does not know this), then the receiver measures the true rate of the other. If the sender is able to manipulate one of the two clocks, without affecting the other, the rate measured by the receiver will change. All that matters is that the sender can affect the *relative* rates of the two clocks, and that they are both visible to the receiver.

In general, the sender only needs to be able to affect the relative *arrival order* of two streams of events (the ticks), which are visible to the receiver. Conversely, if we can arrange that the relative arrival orders of all events visible to the receiver are predictable (the clocks run at the same rate), the channel disappears. This is the motivation for *instruction-based scheduling*, which attempts to prevent a process from using its own execution as a clock.

Returning to the cache channel, any process that shares a cache with the receiver is a potential sender. The sender is able to manipulate the receiver's *rate of progress* by varying its cache usage. Thus, the receiver's *program counter*

C source		ARM instructions	x86 instructions
	1		`mov %eax,%edx`
	2		`shl $0x6,%edx`
	3	`ldr r0, [r1,r3,lsl #5]`	`mov 0x8059b80(%edx),%ecx`
`B[i][0]^= 1;`	4	`mvn r0, r0`	`not %ecx`
	5	`str r0, [r1,r3,lsl #5]`	`mov %ecx,0x8059b80(%edx)`
	6	`ldr r0, [r2]`	`mov 0x804f1c0,%edx`
`C++;`	7	`add r0, r0, #1`	`add $0x1,%edx`
	8	`str r0, [r2]`	`mov %edx,0x804f1c0`
	9	`add r3, r3, #1`	`add $0x1,%eax`
	10	`cmp r3, ip`	`cmp $0x10000,%eax`
	11	`movgt r3, #0`	`cmovge %ebx,%eax`
`while(1)`	12	`b 3`	`jmp 1`

Figure 3.23: Disassembled machine code corresponding to lines 9–12 of the cache channel exploit in Figure 3.6.

forms a clock that the sender can manipulate. Note that the program counter is *always* implicitly available, even if the register itself is hidden—the receiver need only run in a tight loop and count the number of iterations to measure its progress. The receiver now needs only one more clock, that isn't tied to its program counter, to exploit the channel. In Figure 3.6 we used the preemption tick, assuming that all more straightforward clocks would be removed. We thus attempt to tie the preemption tick to the receiver's rate of progress, and thus make the channel unusable, without needing to remove the underlying source of interference.

Note that, in order to to be effective, *all* clocks need to be either removed, or made deterministic with respect to each other. This is likely to be very difficult to achieve in a realistic system, especially if I/O is required. Instruction-based scheduling has been implemented, specifically to address timing channels (e.g. by Stefan et al. [2013]), although its applicability to complete systems is not yet established. What we establish here empirically, is the limitations imposed on the technique by modern hardware.

Triggering preemptions based on the progress of a process is straightforward. We take advantage of the *performance counters* that are available on any modern CPU, and allow us to generate an interrupt once a fixed number of instructions have been retired. Replacing the timer interrupt in the seL4/ARM kernel with an interrupt generated by the PMU (Performance Management Unit) required the modification of only 18 lines of code. The implementation on seL4/x86 was not much harder.

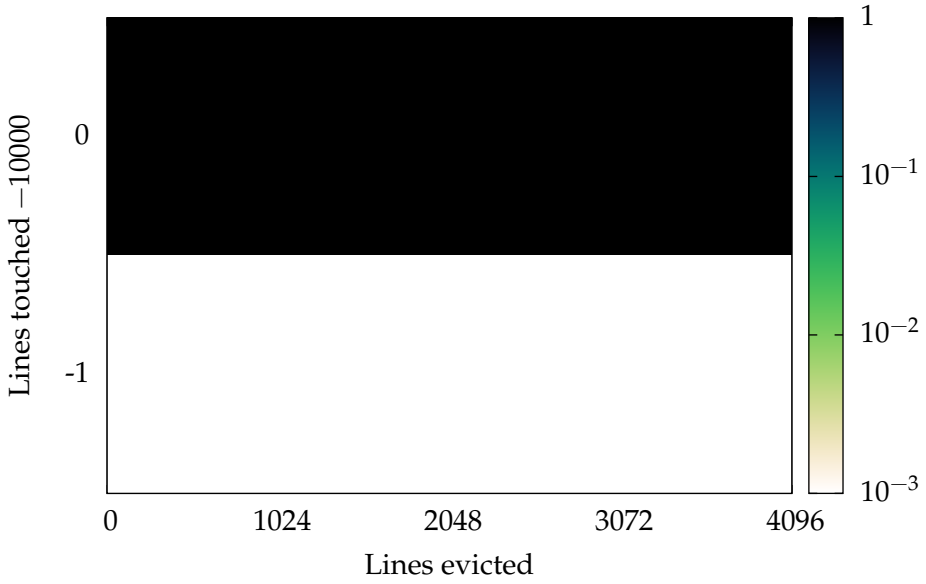

Figure 3.24: iMX.31 cache channel, instruction-based scheduling. $C = 2.12 \times 10^{-4}$b. $CI_0^{max} = 5.41 \times 10^{-5}$b. 10,000 samples per column.

Figure 3.23 shows the machine code corresponding to the main receiver loop (lines 9–12 in function `probe()`) of Figure 3.6, for both ARM (the ARMv6 and ARMv7 versions are identical at this point) and x86. Note that the nested loops have been merged by the compiler. The loop on ARM is 10 instructions long, while on x86 it takes 12. The two correspond almost one-to-one, with the two extra x86 instructions doing the work of the ARM barrel shifter, which can be accessed by any arithmetic instruction. On ARM, the base of array `B` is held in `r1`, and the address of the counter `C` in `r2`. On x86, these are at addresses `0x8059b80` and `0x804f1c0`, respectively. Setting the preemption interval to 100000 instructions on ARM, and 120000 on x86, we expect to see 10000 loop iterations (and hence 10000 lines touched) per preemption.

Figure 3.24 shows the results for the iMX.31, with the baseline of 10000 subtracted. The countermeasure behaves essentially perfectly here, with the overwhelming majority of samples showing exactly 10000 iterations. While the probability is so low that it is invisible, even on a log scale, we do very rarely see 9999. This is due to the small number of instructions executed by the kernel between resetting the preemption counter and returning to the user-level thread. This means that slightly fewer than the desired 100000

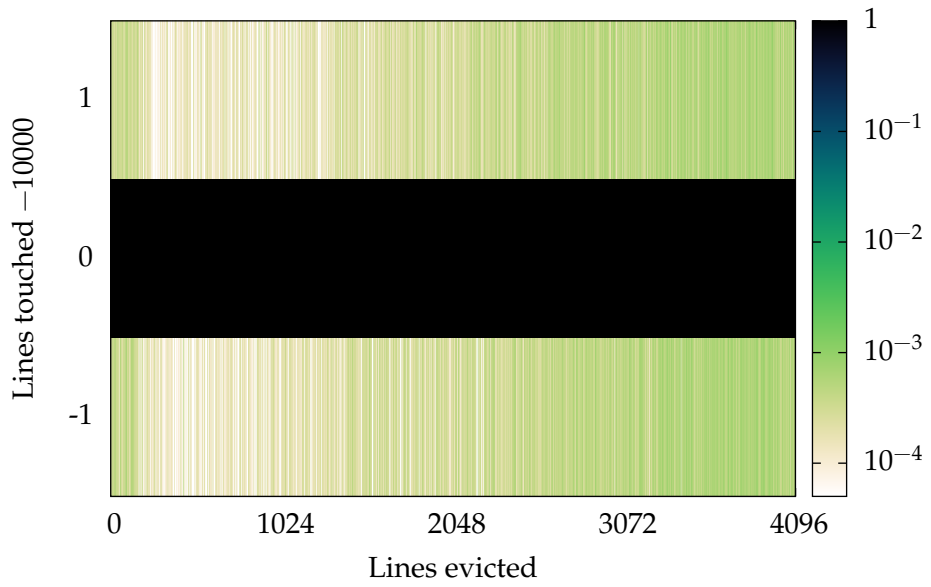

Figure 3.25: AM3358 cache channel, instruction-based scheduling. $C = 1.86 \times 10^{-3}$b. $CI_0^{max} = 9.56 \times 10^{-4}$b. 10,000 samples per column.

instructions are executed at user level, and thus the preemption point will slowly vary, resulting in the occasional short count of iterations. As this effect is not influenced by the sender, it does not constitute an exploitable channel. The true bandwidth is thus zero.

Figure 3.25 and Figure 3.26 show the results for the Cortex A8 cores: the AM3358 and the DM3730. Here, while the majority of samples give the expected value, we see a nontrivial number at both $+1$ and -1, the frequency of which is correlated with the sender's eviction rate. This is a small, but exploitable channel of capacity 1.86×10^{-3}b for the AM3358, and 1.49×10^{-3}b for the DM3730. The deviations of $+1$ are correlated with those of -1, and looking at the raw sample data, we see why. Deviations from the nominal value always appear as a sequence of the form: $\ldots, 10000, 10001, 9999, 10000, \ldots$. That is, a single extra iteration in one preemption interval, and one fewer in the next.

We would expect to see this behaviour if the preemption point were slightly uncertain, but occurred just before the store that increments `C` at line 8 of Figure 3.23. Then, a slight delay in preemption would see one extra store in the prior interval, and one fewer in the following. We infer that the

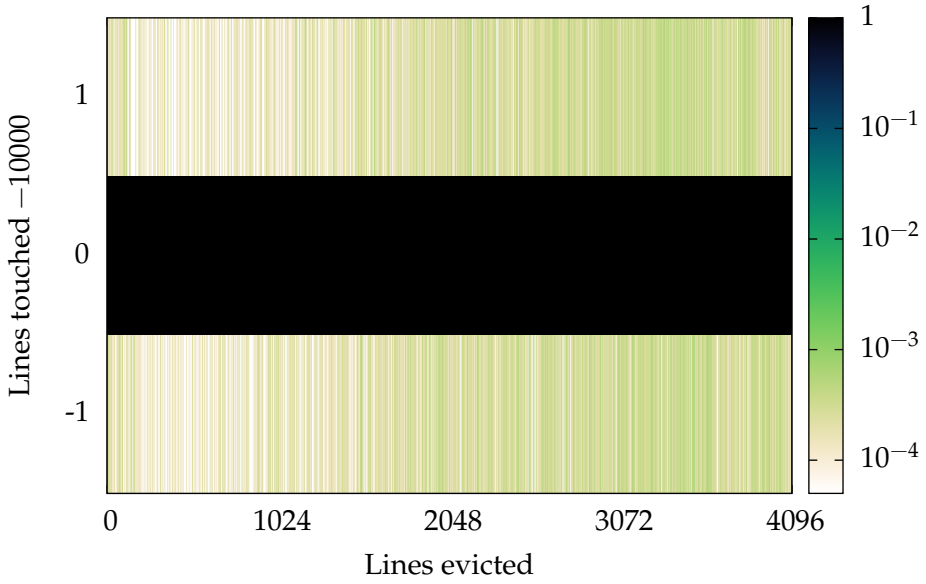

Figure 3.26: DM3730 cache channel, instruction-based scheduling. $C = 1.49 \times 10^{-3}$b. $CI_0^{max} = 7.74 \times 10^{-4}$b. 10,000 samples per column.

instruction count is accurate in the long run, by observing that the long-term average of the iteration count is precisely 10000. It thus appears that, while the count reported by the PMU is accurate, the delivery of the overflow interrupt is delayed if one of the core's two execution pipes is stalled on a cache miss.

In Figure 3.27, which gives the results for the Exynos4412, the residual variation gets worse. Here we see a dramatic variation in the counter value at preemption—far more than can be accounted for by a small variation in the preemption point.

Apart from a small number of deviations of -1, which are consistent with our understanding for the iMX.31 (the small number of kernel instructions causing slightly short intervals), we see that the number of instructions executed is usually significantly *greater* than we expect, and that this error is strongly affected by cache misses—this channel has a capacity of 1.22b. Under heavy contention, or a close to 100% probability of a miss, the most likely result is an overshoot of 4 iterations, or 40 instructions.

There is a clear discontinuity at around the L1 size, which we infer to be the effect of a single L2 miss—with a working set much smaller that the L1 there is only a small probability of a miss (as the sender will only be dirtying

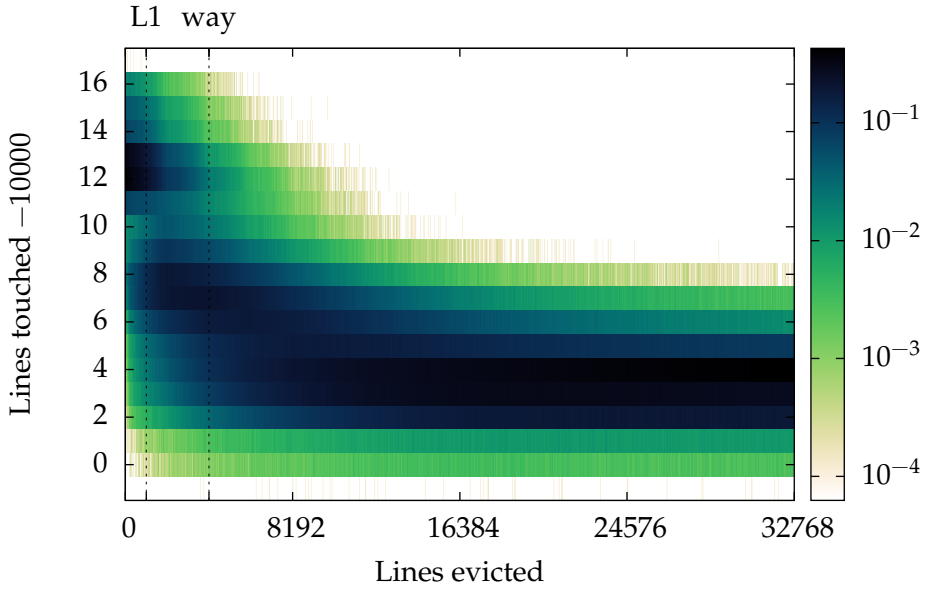

Figure 3.27: Exynos4412 cache channel, instruction-based scheduling. C = 1.19b. $\mathrm{CI}_0^{\max} = 3.76 \times 10^{-3}$b. 1000 samples per column.

its own L1 cache lines), while with one much larger, a miss is almost certain. The most likely value for a small working set (no miss) is 10012, against 10007 with a miss, with the relative likelihoods equal at the L1 size. This suggests an L2 miss penalty of 5 iterations, or 50 cycles. That there is no further variation *within* the range of the L1 cache is expected, as it is flushed on every preemption.

The documentation for this core states that the particular counter that we rely on gives only an approximation of the number of speculatively-executed instructions, which is in turn an even coarser approximation of the number actually retired [Cortex A9 TRM, §11.4.1, table 11-5]. What is most interesting is that the error we see is exactly the opposite of that we would expect from speculation—counting speculated instructions should *overestimate* the true value, as some of these will be discarded. In contrast, this is an *underestimate*.

This fact suggests that what we are seeing is not (at least primarily) an inaccuracy in the instruction count, but imprecision in the delivery of the overflow interrupt, which is consistent with what we saw on the Cortex-A8. On this hypothesis, we are seeing an interaction between the performance counter and the relatively complex pipeline. The core appears to defer the

		ARM instructions
PMU load	1	`mcr 15, 0, r5, cr9, cr12, {5}`
	2	`mrc 15, 0, r2, cr9, cr13, {2}`
	3	`mov r3, r5`
Cache line store	4	`add r3, r3, #1`
	5	`and r0, r3, #255`
	6	`str r0, [sl, r0, lsl #5]`
	7	`cmp r3, r8`
	8	`blt 4`
PMU load	9	`mcr 15, 0, r5, cr9, cr12, {5}`
	10	`mrc 15, 0, r1, cr9, cr13, {2}`

Figure 3.28: Code to test PMU accuracy in the presence of L2 cache misses.

exception until there is a pipeline stall, which naturally tends to happen earlier when the cache miss rate is higher.

To test this hypothesis, we run the code in Figure 3.28 on the Exynos4412. This measures the change in the instruction count after a known sequence of instructions—10000 iterations of the loop between lines 4 and 8. The coprocessor operations on lines 2 & 10 load the counter, and we dirty one cache line per iteration on line 6. We should see 50002 instructions executed *between* the two counter loads, or 50003 if one endpoint is included (assuming that the counter increments consistently either before or after the load instruction).

In fact, we consistently see a count of 50006 instructions which tells us two things: The first is that the branch misprediction that occurs on loop termination does contribute to the instruction count, with the branch itself and the first two loop instructions being speculated (the store to memory is not speculated). The second, and more interesting, fact is that the count is not affected by L2 misses, which implies that the effect we see in Figure 3.27 is the delayed delivery of the overflow exception.

The result of instruction-based scheduling on the Cortex A9 is disappointing, giving only a 5.8-fold improvement: from 7.04 to 1.22b, and this same core gave the weakest performance under colouring. As an experiment, Figure 3.29 shows the channel with both countermeasures enabled. Relative to partitioning alone, relaxed determinism gives a further 3.5-fold improvement: from 8.13×10^{-2} to 2.32×10^{-2}b, our best result on this platform. Whether this small additional improvement is worth the hassle of removing all remaining clocks is not clear.

Figure 3.30 shows our results for the E6550 which, like the Exynos4412,

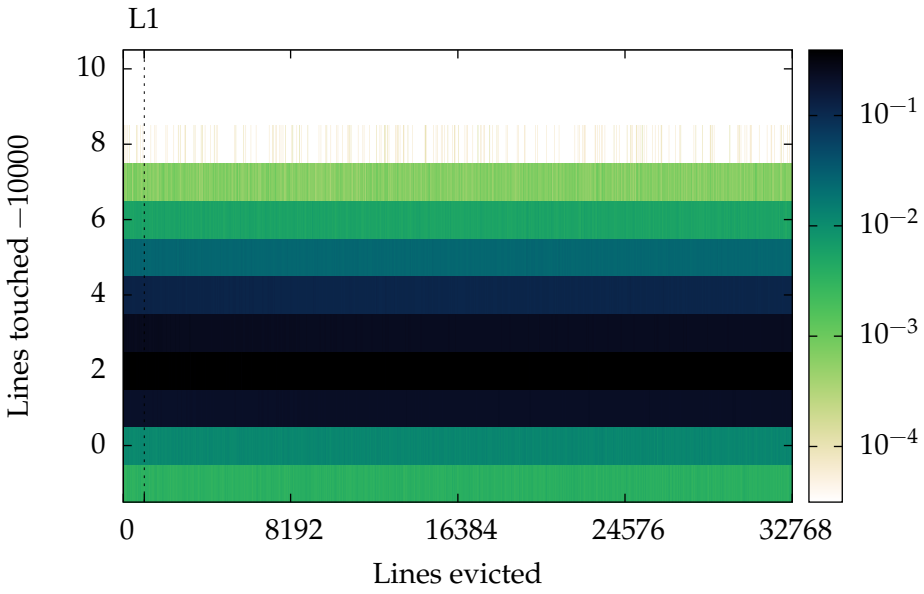

Figure 3.29: Exynos4412 cache channel, instruction-based scheduling and partitioning. $C = 2.32 \times 10^{-2}$b. $CI_0^{max} = 2.78 \times 10^{-2}$b. 500 samples per column.

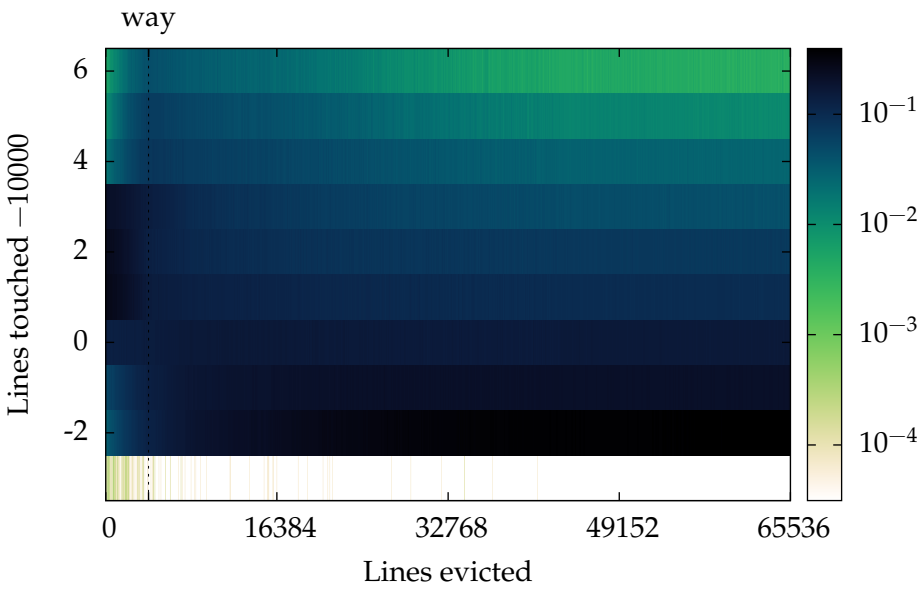

Figure 3.30: E6550 cache channel, instruction-based scheduling. $C = 5.71 \times 10^{-1}$b. $CI_0^{max} = 3.41 \times 10^{-2}$b. 600 samples per column.

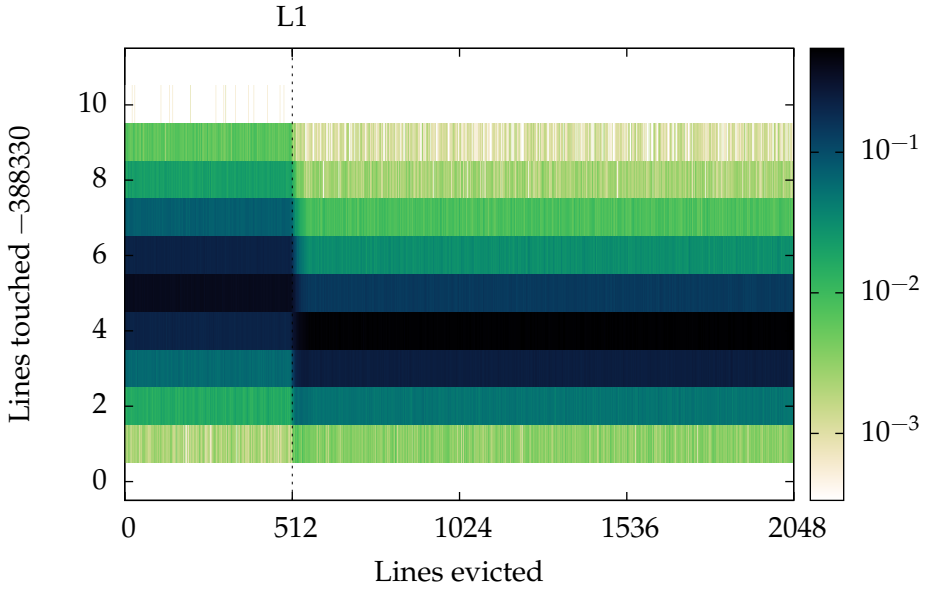

Figure 3.31: E6550 bus channel, relaxed determinism. $C = 2.90 \times 10^{-1}$b. $CI_0^{max} = 7.90 \times 10^{-3}$b. 3000 samples per column.

is a speculative, out-of-order CPU. Again, we see a variation in the number of iterations per interval, but here clustered more tightly around the correct value, and not showing the consistent overshoot that we saw in Figure 3.27. This suggests that the integration of exceptions with speculation and out-of-order execution, on this core, is more precise. We still see a significant left-to-right variation, correlated with the likelihood of a cache miss, giving a capacity of 5.71×10^{-1}b. The countermeasure is thus also comparatively ineffective on this core. We still appear to see exceptions being delayed until pipeline stalls.

Instruction-Based Scheduling and the Bus Contention Channel

The great attraction of instruction-based scheduling is that it prevents (in principle), the exploitation of *any* timing channel, while an approach such as cache partitioning, while effective, is limited to a particular channel (in this case cache contention).

Figure 3.31 shows the result of applying IBS to the bus contention channel of Figure 3.12. While the magnitude of the effect is decreased by a factor of 200,000, a clear artefact remains, and the capacity drops by only a factor

of 15. This result is not surprising given that we already know that the performance counter exception is affected by stall cycles, but it is nevertheless disappointing. Nonetheless, this is the greatest reduction in bus-channel bandwidth that we were able to achieve.

Performance Cost

In contrast to colouring, which limits the cache available to a process, determinism has no intrinsic *throughput* cost. Where it is likely to hurt is in *fairness* and *latency*.

Preemption is the means by which the CPU is shared between processes, and changing the conditions for preemption naturally affects fairness. While normally each thread has an equal share, we instead enforce an equal sharing of *work*—each thread retires the same number of instructions per timeslice. The wrinkle is that these instructions may take a very different amount of time to execute. At one extreme, a completely CPU-bound task that never stalls will complete in the minimum possible time. On the other hand a memory-bound task that stalls on almost every instruction will take dramatically longer. For example, a long sequence of pointer-chasing loads (where the address of the next load is calculated from the result of the previous, and thus cannot be speculated or prefetched) could well take 1000 times longer to complete than its CPU-bound competitor. Memory-bound tasks thus receive a proportionately higher share of CPU time than those that execute more efficiently. Whether this is a problem depends on the application.

A further effect of the ballooning of timeslices is that latency, the maximum interval between allocated timeslices, will increase. This may well be a problem for a latency-sensitive application.

3.6 Lucky thirteen as a Remote Channel

Our final countermeasure deals with a different threat model: the *remote attacker*. This scenario presents us with a different set of tradeoffs than those considered so far. On the upside, remotely-exploited side channels are generally of lower bandwidth, and the attacker has less ability to observe the system. On the downside, we have no control over the attacker itself. In particular, it must be assumed to have an accurate clock, and hence an approach such as instruction-based scheduling is impossible.

The practicality of remotely exploiting a timing-based side channel against a real system was demonstrated by Brumley and Boneh [2003]. Their work demonstrates a real attack against OpenSSL, which exploited variation in the runtime of its implementation of RSA [Rivest et al., 1978] that depended both on the secret key, and the attacker-supplied input. A notable feature of this attack is that it could be carried out at a significant network distance from the target, opening up the possibility of attacks on internet-facing systems.

We likewise demonstrate that the lucky thirteen attack on OpenSSL's implementation of datagram TLS (DTLS) reported by AlFardan and Paterson [2013] is exploitable at essentially unlimited network distance. We then show that OS-level techniques can be used to effectively mitigate it, without requiring any modification of the vulnerable cryptographic implementation. This approach is entirely *black box*.

The Attack

This attack exploits the fact that TLS first calculates the MAC (message authentication code, or digest), and then encrypts it. This allows a malicious man-in-the-middle to modify packets taken from the wire and submit them to the server, which will then decrypt and begin to process them *before* their authenticity is established. In this case, the attacker exploits the non-constant execution time of the MAC calculation itself by manipulating the padding bytes contained in the packet. We construct two packets: M_0 and M_1, that take a different length of time to process, before the server rejects them (as the MAC check on the manipulated packet will fail), where this time depends on the (encrypted) contents. The attacker can thus *distinguish* two encrypted packets by intercepting them and sending them to the server, on the client's behalf.

Figure 3.32 shows the distribution of response times for packets M_0 and M_1, as measured at the greatest network distance we could achieve: between an Amazon EC2 datacenter in Oregon, USA and our lab in Sydney, Australia: a distance of 12,000km. While there is a substantial overlap between the distributions, they are nevertheless easily distinguishable. An attacker observing only a single round trip has a 62% chance of guessing which packet was sent.

Table 3.2 list the vulnerability for a number of different attack scenarios, all conducted against OpenSSL running on the AM3358. While the vulnerability clearly drops with increasing distance, it only does so rather slowly. We thus

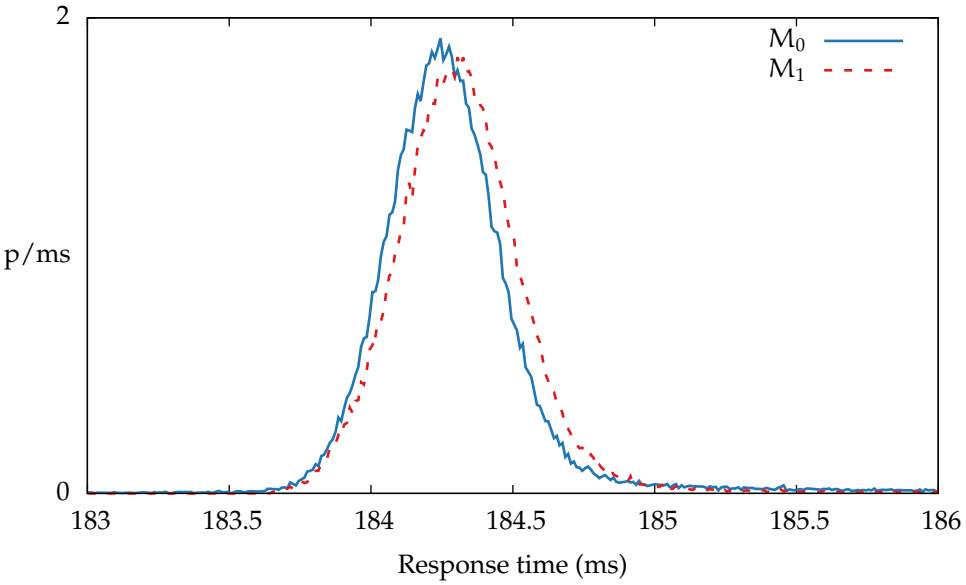

Figure 3.32: Histogram of DTLS echo times for OpenSSL 1.0.1c, at intercontinental distance—13 hops and 12,000km, see row 4 of Table 3.2. Generated from 10^5 samples per packet, and binned at 10µs. These peaks are distinguishable with 62.4% probability.

Version	Hops	D(km)	V_{max}	$\mathcal{ML}(b)$	RTT$\pm\sigma$(ms)	Note
1.0.1c (VL)	1	0	0.998	0.986	0.734 ± 0.007	
	3	0	0.597	0.109	1.193 ± 0.229	WLAN
	4	4	0.766	0.566	1.281 ± 0.056	
	13	12,000	0.624	0.207	185.1 ± 31.47	
1.0.1e (CT)	1	0	0.622	0.071	0.796 ± 0.005	
1.0.1c (SD)	1	0	0.570	0.030	0.751 ± 0.005	

Table 3.2: Vulnerability against network distance.

conclude that network distance offers little or no protection against even quite small timing attacks (the underlying variation is less than 10µs).

Also included in this table is the vulnerability for the latest version of OpenSSL (1.0.1e), which has replaced the vulnerable MAC calculation with a constant-time implementation. We see that the vulnerability is reduced, but not eliminated: an attacker can still guess with 62% probability. In the following section we demonstrate that we can mitigate this channel more

effectively, and with lower overhead, using OS-level techniques. We quote min leakage here, as we are concerned with vulnerability to a small number of guesses.

3.7 Scheduled Message Delivery

Our countermeasure relies on the observation that an attack via the network must use channels that are under the control of the operating system. In seL4 in particular, both synchronous communication (e.g. remote procedure calls or RPC) and asynchronous notification (e.g. interrupt delivery) are provided by kernel-managed objects: *endpoints*. We take advantage of this to impose a delay mechanism at the level of message delivery, to prevent the leakage of information via packet arrival times, without modifying the vulnerable component itself.

Our threat model consists of a public-facing component that manipulates sensitive data. From its own perspective, the attacker has a direct channel to this sensitive component. In practice, this channel traverses at least one kernel-controlled endpoint, where we apply our countermeasure. This is implemented as a lightweight *mechanism*, which implements the *policy* set by a separate user-level component.

We pace synchronous message delivery using real-time scheduling. Specifically, endpoints are modified to allow an authorised component to set a *minimum response time*, which is enforced by the kernel using a simplified EDF (earliest deadline first) scheduler.

Our prototype implementation is within seL4 itself, making high performance easier, although with careful design it could be replicated at user level, obviating the need for changes to the verified kernel.

The seL4 kernel already provides a mechanism for efficiently implementing RPC, the *reply capability* (cap). A client thread invokes the *Call* method (an seL4 system call) of an endpoint associated with the server. This transfers the message payload synchronously, blocking the calling thread until the server replies. The server blocks on the same endpoint for client requests. When one arrives, the server receives both the message and a cap. This reply cap functions as a one-shot endpoint: sending on it unblocks the client, delivering the reply and deleting the cap. This lets multiple clients use the same service endpoint, and the server direct its replies accordingly.

We piggyback on this mechanism: the minimum response time of the endpoint sets the earliest instant at which the reply may be sent. When a call is made, the kernel records the instant of invocation, adds the delay, and stores this *release time* in the newly-created reply cap. If the server invokes the reply cap before the delay has elapsed, it blocks until it does. By default the delay is set to zero, and the countermeasure is disabled.

Policy is provided by the user-level mitigator with the authority to set the delay. This delay is set on the client side, allowing the mitigator to transparently update it without the vulnerable server being aware. An existing server can thus be used unmodified. The delay parameter provides a hook to allow the mitigator to implement a dynamic delay policy. Zhang et al. [2011] present just such a policy, that adaptively adjusts the delay (with exponential backoff), such that under attack, the system will converge on its worst-case execution time, while leaking a provably-bounded number of bits.

There is some potential for a malicious client to reduce the system's performance by deliberately triggering the countermeasure. This is limited to the worst-case response time which, as we will see in Figure 3.34 is only 10% greater than the unmitigated response time, even for an unrealistically tough example. The extra time, moreover, is not wasted, but can be reclaimed for useful work (or used to enter a low-power idle state).

The authority to set the delay is controlled by the standard capability system of seL4, which allows very flexible, fine-grained distribution of authority. A feedback channel is also provided to allow the mitigator to detect overruns and adapt its policy.

As mentioned, delays are enforced by a simplified EDF scheduler: we maintain a heap of pending events, sorted by increasing release time, and set the timer to expire at the first. We enforce no deadlines, which simplifies implementation. This is thus more accurately described as an earliest-*release*-first scheduler, and controls all time-triggered behaviour: preemptions are scheduled alongside message delivery.

Scheduled Delivery against Lucky Thirteen

Figure 3.33 shows how we use scheduled delivery to protect OpenSSL. The system is partitioned into three layers, shown left-to-right, implemented by separate components. The network stack (left) is lwIP [Dunkels, 2001], the TLS layer OpenSSL 1.0.1c, and the application an echo server. Data flow

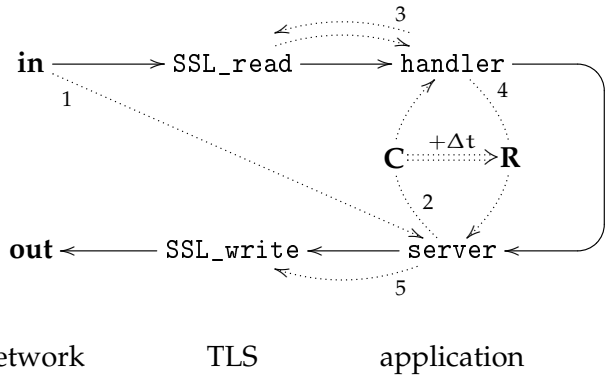

Figure 3.33: Scheduled delivery for OpenSSL. Solid lines are packet flow with dotted control flow. Server blocks at **C** (call), until **R** (reply), after *at least* Δt.

is shown by solid arrows and proceeds clockwise. Control flow is shown by the numbered dotted arrows, and begins with the server thread being notified of input (1). It then *calls* the handler to retrieve input (2), blocking (**C**) and creating a reply capability (**R**) with a delayed release time. The handler calls OpenSSL (3) to retrieve decrypted packets. The attacker's packets are processed by `SSL_read`, which may block, and returns once a valid packet is found. The handler then restarts the server (4), which blocks until its delay expires, and then forwards its output to `SSL_write`.

Figure 3.34 shows the response times for the two packets, as measured from a directly-connected machine (no switch). In the terminology of Chapter 2, the packets M_0 and M_1 are the two (secret) inputs to the channel, and the response time is its output. Each pair of peaks for M_0 and M_1 (labelled VL, CT and SD respectively) in Figure 3.34 is essentially a channel matrix with just two columns (M_0 and M_1), each built by taking 10^6 observations of the output for each input.

The leftmost pair of peaks (VL) are the response times for the vulnerable implementation of OpenSSL 1.0.1c, executing 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 deals with the difficulty that DTLS does not acknowledge invalid packets. The machine under attack is executing an echo server, protected by DTLS. As line 1 of Table 3.2 shows, these peaks are trivially distinguishable, allowing the attacker to correctly guess which packet

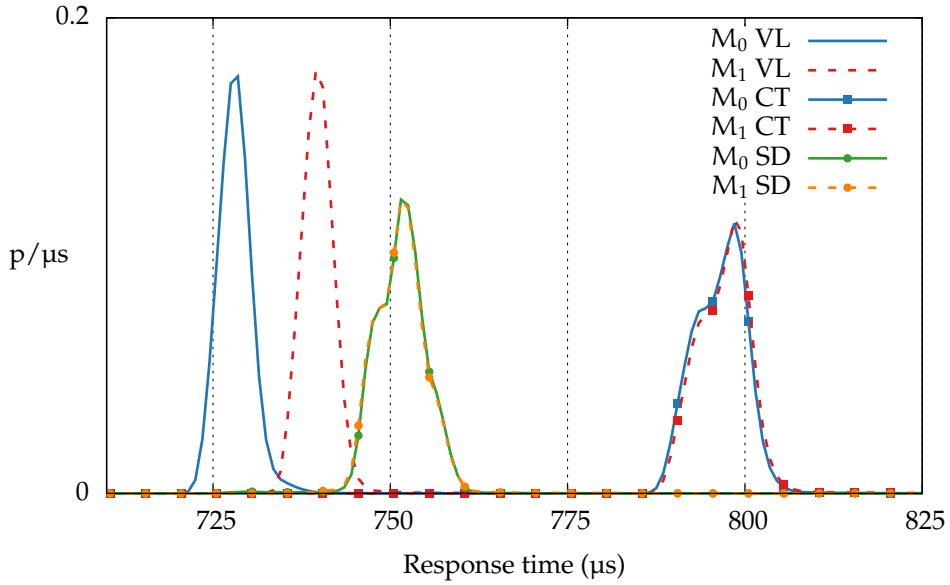

Figure 3.34: DTLS distinguishing attack—histogram of response times for packet M_0 versus M_1. Shows distinguishable peaks for vulnerable implementation in OpenSSL 1.0.1c (VL), constant time implementation in 1.0.1e (CT), and results with scheduled delay (SD) demonstrating reduced latency. All curves generated from 10^6 packets for both M_0 and M_1, and binned at 1µs.

was sent with 99.8% probability in the worst case. This corresponds to a leak of 0.986b of min entropy, out of 1 total.

The rightmost pair of peaks (CT) give the response time for the constant-time implementation of OpenSSL 1.0.1e, which substantially eliminates the vulnerability—the curves are *almost* identical. They are, however, not *precisely* identical, as row 5 of Table 3.2 shows—the two can still be distinguished with 62.2% probability, while to declare the system completely secure, the attacker should only be able to guess correctly half the time. This result emphasises the difficulty of producing portable cross-platform constant-time code, and indicates that the production version of OpenSSL (as of writing) is still somewhat vulnerable to this attack.

Under scheduled delivery, any variation in the execution time of OpenSSL's decryption routines is absorbed between steps 2 and 4. The results obtained (after manually tuning the delay) are given by the central pair of peaks (labelled SD) of Figure 3.34. As Table 3.2 shows, we (row 6) achieve a lower

vulnerability than the constant-time implementation (row 5) of OpenSSL 1.0.1e (57% distinguishability or 0.03b leak of min entropy versus 62% and 0.07b), with a 50µs (≈ 6%) better latency. The better matching between the curves occurs as nothing in our implementation is data dependent, and the only intrinsic latency penalty is the cost of blocking and restarting the server thread, which we see from the figure is ≈ 10µs.

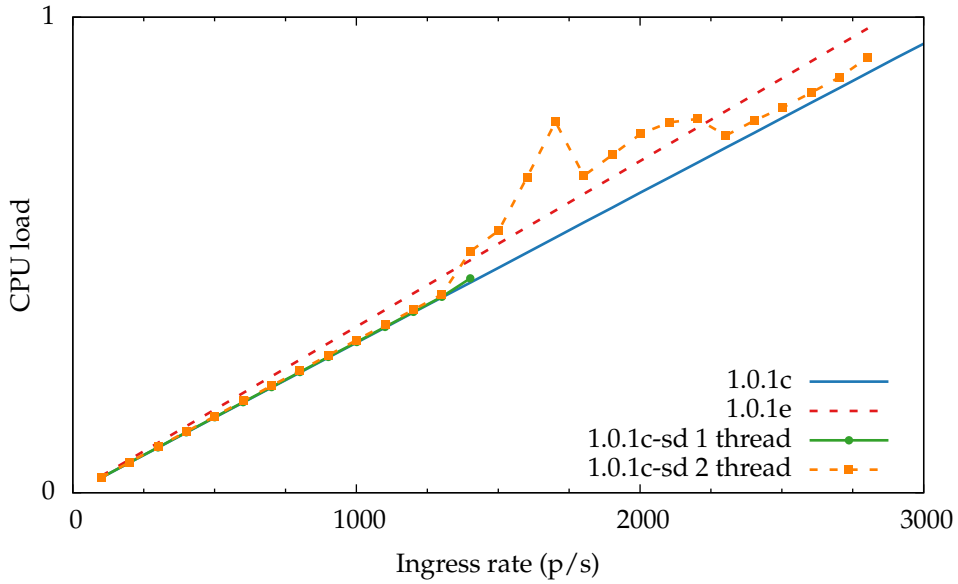

Figure 3.35: Load performance and overhead of scheduled delivery against unmodified OpenSSL 1.0.1c, and the constant-time implementation of 1.0.1e.

Figure 3.35 shows overhead at various loads. Each curve here plots CPU load against ingress rate, up to the point at which packet loss begins (this machine has a single core). For the unmodified implementation (1.0.1c, blue), load increases linearly with packet rate, with packet loss beginning with the onset of saturation at 3000 packets per second (p/s). The constant-time implementation of OpenSSL 1.0.1e (red) shows a 10.6% CPU overhead relative to this baseline, consistent with the increased latency observed in Figure 3.34, and shows a correspondingly earlier saturation, at 2800 p/s. This extra CPU time is wasted performing redundant computations, to ensure that the execution time for any packet is always the worst case.

The next curve (green), for a single-threaded server using scheduled delivery, tracks that for the original, vulnerable, implementation closely, showing

only a 1.7% overhead. This shows the benefit of avoiding the redundant computations inherent in a constant-time implementation. Instead of busy-waiting (as in the improved SSL implementation), our implementation idles by entering a (low-power) sleep state.

The same curve also demonstrates its weakness, with packet loss occurring at only 1400 p/s, as any packets that arrive while the server is sleeping are dropped, limiting achievable throughput. This benchmark is an extreme case, as the echo server does no work at all—all CPU time is devoted to OpenSSL. In a more realistic system, where the fraction of time devoted to vulnerable cryptographic routines is small, the throughput loss will be similarly low.

The orange curve is included to demonstrate that the slack time can be reclaimed. Here, we use two server threads to recover the sleep cycles (one can respond while the other is delayed). This is not safe to do if the results are visible to the attacker: running a second server thread (as we have done here) merely transforms exploitable latency variation into exploitable throughput variation. In a more realistic setting, however, this is not necessary as, as discussed above, the amount of time wasted sleeping will in practice be dwarfed by the time spent executing the application.

3.8 Related Work

The publication of the orange-book standards for high-assurance systems [DoD] (now superseded by the common-criteria standards [NIST]) drove academic and industrial interest in systems able to be certified to the highest levels defined (e.g. A1), which required formal treatment of covert channels.

As already discussed, the VAX VMM project attempted to formally address timing channels, introducing the fuzzy time technique [Hu, 1991, 1992b; Trostle, 1993] to inject noise into all visible time sources. In contrast, we focus rather on reducing signal levels by mitigating channels directly, having noted the limitations of adding uncorrelated noise to a channel. Numerous other channels were identified and analysed in connection with the project [Karger and Wray, 1991; Wray, 1991], in particular bus contention (which we analyse) and the disk-arm channel (which we do not).

Our empirical approach has some similarity to more recent work of Gay et al. [2013], on measuring the bandwidth of interrupt-related covert channels. In contrast to our approach of calculating the bandwidth directly from a

sampled channel matrix, they hypothesise a binomial model for the leakage mechanism they analyse, and fit it to the observed data. The validity of the model does not appear to have been extensively validated. We have the advantage of not needing to provide such a model, although it could provide insight into the underlying mechanism.

Cache colouring was originally proposed to allow real-time systems to partition the cache among tasks, in order to reduce contention-related jitter [Liedtke et al., 1997], and is still applied in contemporary systems [Ward et al., 2013]. Its potential as a mitigation technique was independently recognised by Godfrey [2013], who tested it on the Xen hypervisor against a live side-channel attack. Our results are complementary, showing by means of our synthetic benchmark that the underlying channel is effectively mitigated. A further point of contrast is that we partition kernel as well as user memory.

StealthMem [Kim et al., 2012] cleverly exploits a widespread cache-replacement policy (k-LRU) to provide a limited quantity of memory that will never be evicted by a competing process. While effective, this approach suffers (along with colouring and instruction-based scheduling) from relying on undocumented (and non-guaranteed) hardware behaviour. In contrast to cache colouring, it requires sensitive components to be modified to exploit the protected memory, whereas we mitigate components as black boxes.

In contrast with fuzzy time, other authors have made the same realisation as us that increasing determinism is more effective than injecting noise, and have repurposed deterministic execution frameworks, originally developed to aid in debugging multithreaded systems [Bergan et al., 2010; Aviram et al., 2010b,a; Ford, 2012], to mitigation. In contrast to this full determinism approach, requiring modification of vulnerable components, instruction-based scheduling, as implemented by us and in the Hails web application framework [Stefan et al., 2013], specifically targets timing channels and is thus simpler to implement and importantly, black box.

The scheduled-delivery countermeasure is inspired by the work of Askarov et al. [2010] and Zhang et al. [2011] who showed how to adaptively set delays in order to provide on-line defense against remote timing attacks. We present an efficient, light-weight *mechanism* capable of implementing such a policy.

While in this work we analyse and mitigate known channels on modern hardware, incidentally discovering a few new ones, various authors have tackled the difficult problem of systematically (or even automatically) identifying

channels. The shared-resource-matrix methodology of Kemmerer [1983, 2002] provides a systematic approach to identifying potential channels through known mechanisms, and has been influential in the literature. The problem of channels through known (kernel) mechanisms does, however, appear to be subsumed by more recent work on verified isolation [Murray et al., 2013]. Sidebuster [Zhang et al., 2010], in contrast, is a more application-specific, and automatic, approach to identifying a narrower class of channels: remotely exploitable timing channels in web applications. No matter how channels are identified, we argue that they require extensive empirical evaluation, without which any estimate of severity is, at best, an educated guess. In the case of hardware-mediated channels, close analysis and a familiarity with the hardware remain essential.

There are other countermeasure approaches that we have not covered, for example the work of Gray [1993, 1994]. The focus of Gray's work, and of much of the existing work in the field is on introducing noise, which we argue is an ineffective technique, when compared to increasing determinism, or eliminating contention. Lattice scheduling, an approach to minimising the number of cache flushes required to close the cache channel, due to Hu [1992a], is covered in more detail in Chapter 5, where we provide a formally-verified implementation.

3.9 Conclusions

The results presented in this chapter allow us to reach a number of conclusions regarding the effectiveness of mitigation, both by instantiating the models of Chapter 2 and observing the performance of the countermeasures on modern hardware.

Firstly, and unsurprisingly, the effect of unmitigated channels is devastating. Every platform leaks at least 4 bits per observation via the cache channel which, even if limited to a relatively low sample rate (333Hz in our example) gives a bandwidth of kilobits per second. The bus contention channel is similar. At these bandwidths, leaking a small but valuable secret, such as an encryption key, would require at most a few seconds. Even the transmission of bulk data is feasible, especially on a multiprocessor where the sampling rate is effectively unlimited. Even for the remote channel (the lucky thirteen attack), where the appropriate measure is no longer bandwidth but min leakage, we

see total compromise after only one observation: this system leaks 0.99 bits of min-entropy (out of a total of 1 bit possible) in a single attempt, despite being a side and not a covert channel, and thus being accidentally, rather than deliberately exploited. These figures are also well in excess of the 1b/s and 0.1b/s permitted by the thirty year old orange-book standards [DoD]. For systems concerned with information leakage, effective countermeasures are thus essential.

We find that colouring is generally effective against the cache channel, as long as care is taken to deal with all residual channels. We do see a trend of decreasing effectiveness on more complex hardware, with capacity reductions of 200 to 1000 times for the simpler cores (the ARM1136 and Cortex A8), compared to less than 100 for the more complex A9 (Exynos4412) and Conroe (E6550) cores. In the case of the E6550, we are greatly restricted by the lack of a mechanism to selectively flush the L1 cache, making it presently impossible to achieve both high performance and high security on the x86 platform. Our results do not yet include the effect of TLB flushing, but we nevertheless conclude that cache colouring remains broadly effective, although harder to effectively implement on recent hardware.

Instruction-based scheduling shows a similar pattern, but much more dramatically. It drops from begin extremely effective on simpler cores (apparently perfect on the ARM1136, and thousand-fold reductions on the A8), to being essentially useless on the A9 and Conroe cores. This is due to its reliance on the accuracy of performance counters, which we have demonstrated to be poor, and strongly affected by events under the sender's control. The alternative, of deliberately stopping short and then single-stepping the processor to the required point, abandons the advantages of simplicity and inherently low performance impact that make the approach attractive. Combined with the necessity of eliminating all timing sources available to the receiver leads us to conclude that this countermeasure is not practical. Even for the bus channel, its performance is poor enough not to be worth the effort.

It thus appears that while known countermeasures can be used to make a substantial improvement in the capacity of local hardware-mediated channels, success is highly reliant on very careful implementation and empirical analysis, and depends on the effect of undocumented and evolving low-level hardware details. For the time being, systems requiring a truly high level of local covert-channel security will need to continue to use either fully isolated or

specialised hardware. Cache colouring does, however, offer hope for reducing the scope for side-channel attacks on sensitive cryptographic components, given a robust and practical implementation.

The outlook for remote channels is much more positive. We see that while we do not yet have perfect mitigation for channels such as the lucky thirteen attack, black-box techniques allow us to outperform the state-of-the-art solution (constant time reimplementation), with lower implementation effort, better security *and* better performance. There is also nothing to suggest that a more carefully audited and implemented mitigation could not achieve perfect security, at low performance cost. The contrast with the situation for local channels is stark, and highlights the extremely large security challenge posed by shared hardware.

Finally, these experiments provide us with the data to instantiate the mathematical models we derived in Chapter 2, to see whether the level of mitigation that we can achieve is sufficient to provide meaningful protection against the established threat models. We take our very best (non-perfect) result, for instruction-based scheduling the DM3730, showing a residual channel of no more than 1.49×10^{-3}b, and substitute the matrix of Figure 3.15 for that of Figure 2.18.

This channel shows a min leakage of 4.32×10^{-3}b. Using the pessimistic min leakage models of Section 2.7, and re-using the example secret distribution, after 60 guesses we leak only 0.25b (out of 5.9b) of min entropy via the side channel, but already have a roughly 25% chance of guessing correctly simply by knowing the prior distribution on secrets. So, in the case of a very weak secret (6 characters selected nonuniformly from 6 possibilities), the effect of side-channel leakage is dominated by intrinsic leakage. This implementation is *safe* (its vulnerability is mostly intrinsic), but only barely.

The situation reverses dramatically for a stronger secret: A strong (uniformly random) 1024b secret could be completely transmitted in 700,000 samples (about 35 minutes) by a trojan horse (exploiting this as a *covert* channel), and could be guessed with 50% probability in only 120,000 (about 6 minutes) by an attacker exploiting a worst-case side channel. Additionally, even the best-case result for the remote channel, a min-leakage of 0.03b, implies a 50% chance compromise in only 15 guesses, although this can likely be improved with a more thorough implementation.

These results are all naturally pessimistic: they are the worst-case outcome

if the full possible bandwidth of the channel is exploited. Real attacks are likely to be less effective, and systems somewhat more secure in practice. Nevertheless, our results indicate that the state of the art in implementing leakage-resistant systems on commodity hardware is a long way from allowing us to give comfortable security guarantees given the pessimism of established leakage modelling theory, even if we are able to meet the rather arbitrary guidelines specified for high-assurance software.

4 pGCL for Systems

This chapter draws on work first presented in the following paper:
David Cock. Verifying probabilistic correctness in Isabelle with pGCL. In *Proceedings of the 7th Systems Software Verification*, pages 1–10, Sydney, Australia, November 2012. Electronic Proceedings in Theoretical Computer Science. doi:10.4204/EPTCS.102.15

After establishing our threat model, the guessing attack, and our approach to modelling leakage in Chapter 2, and evaluating the effectiveness of low-level mitigation strategies in Chapter 3, we now consider how the problem can be attacked from the other end—formally verifying the absence of (or limits on) leakage using a high-level specification. To do so, we need to bring probabilistic properties within the scope of an seL4-style proof.

The seL4 theorem [Klein et al., 2009] is a refinement from an abstract specification, via an intermediate executable model [Derrin et al., 2006], to a high-performance C implementation [Winwood et al., 2009]. The higher levels are specified in a locally-developed monadic framework [Cock et al., 2008], which allowed us to model the highly imperative style of the kernel in the pure executable fragment of the Isabelle [Nipkow et al., 2002] theorem prover's mechanised higher-order logic (Isabelle/HOL).

Additional variables (such as time) can be added to a such monadic specification, by extending the state type appropriately and stating how the system acts on it. What we cannot yet handle is the stochastic nature of these variables. The execution time of a complex system is seldom deterministic, and even when it is theoretically predictable, doing so is often so complex that we are

forced to retreat to statistical models. Therefore, we extend our toolkit to tackle statistical reasoning, without giving up existing capabilities.

To this end, we formalise the probabilistic guarded command language of McIver and Morgan [2004] (pGCL), in Isabelle/HOL. Using this, we can produce machine-checked refinement proofs for probabilistic systems, building on the extensive prior work on pGCL. Our formalisation integrates cleanly with our existing work, all of which is in Isabelle. We show that we can reuse existing results about seL4, by mapping our existing monadic Hoare logic into pGCL.

We have not conducted a full-system proof of probabilistic non-functional properties on the scale of L4.verified, but in this and the next two chapters, we demonstrate that the required tools are largely in place.

4.1 The Case for Probabilistic Correctness

The verification of the seL4 microkernel demonstrated that we can verify the functional correctness of real systems. Functional correctness, however, covers only properties that are *guaranteed* to hold. This excludes classes of properties relevant in practice, for example execution time. By allowing security properties that hold only with some probability, we can give a more nuanced classification of systems than with a functional property such as noninterference [Goguen and Meseguer, 1982]. In particular, we wish to classify systems according to the Shannon capacity, or min capacity, of identified side channels.

Probabilistic Behaviour in Systems

In formal specification, it is convenient to treat a system as predictable, and make absolute claims about its behaviour. Once implemented, while the system may be predictable *in principle*, its size and complexity (and often under-specification) makes this impractical.

The usual approach in this case is to retreat to a probabilistic model, informed by benchmarking. That this is true is apparent on reading the evaluation section of a systems paper: While the precise performance of a system is theoretically predictable and could thus be calculated, what we see are benchmarks and histograms. The tools of the evaluator are experiments and statistics! This is a testament to the immense difficulty of precisely predicting

performance. Once real world phenomena intrude, as in networking, exact prediction becomes impossible, even in theory.

Moreover, some properties can only be treated statistically. If such properties are correctness-critical, any proof must involve probabilistic reasoning. Our particular concern is variable execution time, particularly where it gives rise to leakage channels. Recall Figure 2.17, showing the response time of a system using `strcmp`. Reasoning about the distribution of response times was crucial in calculating the channel capacity. This is a correctness-critical security property that is unavoidably probabilistic.

Security Properties

Returning again to the guessing attack, imagine that our adversary guesses in decreasing order of probability, updating these probabilities dynamically, as did the optimal attacker described in Section 2.4. Here, as there, our security condition is that it does not guess correctly in fewer than k attempts.

For a functional security property, we would calculate the set of initial states that guarantee that the property holds in the final state: its *weakest precondition* (wp). Security could be assured by showing that the initial state lies in this set.

In a probabilistic system, however, it might be that from *any* initial state, there is a nonzero (albeit small) probability that the final state is insecure. This occurs, for example, if the attacker guesses randomly. The most we could then hope to find is the *probability* that the final state is secure.

What we need is a probabilistic analogue of the weakest precondition. Consider the weakest precondition as a function that takes only the value 0 or 1, depending on whether a state is in the weakest precondition set or not. Could we instead return some value *between* 0 and 1? Instead of answering "secure" or "not secure" for a state, answering "secure with probability p"? We could then show that the system remains secure with probability $\geqslant p$, if $\mathrm{wp} \geqslant p$ in the initial state. In pGCL we find just such an analogue. This is an extension of Dijkstra's guarded command language (GCL) [Dijkstra, 1975], with a unified treatment of both demonic and probabilistic nondeterminism.

Refinement and Security

The standard problem with refinement in security is nondeterminism. Writing $a\ ;;\ b$ for 'do a then b', $c \sqcap d$ for 'do c or d', and $h := x$ for 'assign x to h', consider the following program:

$$h := \text{secret}\ ;;\ (l := 0 \sqcap l := 1) \quad \text{where} \quad \text{secret} \in \{0, 1\}$$

Let h (high) be hidden, and l (low) be visible. We wish to formalise the statement: 'The value in l doesn't reveal the value in h'. Writing $a \sqsubseteq b$ for 'b refines a', meaning that every trace[1] of b is also a trace of a, the following is a valid refinement:

$$h := \text{secret}\ ;;\ l := h$$

This clearly violates the security property, but is permissible as every action it takes, could have been taken by the specification.

For a property Q to survive refinement, writing $\vdash$ for predicate entailment, we need that

$$\frac{a \sqsubseteq b}{\text{wp}\ a\ Q \vdash \text{wp}\ b\ Q} \qquad \text{whence by transitivity,} \qquad \frac{P \vdash \text{wp}\ a\ Q}{P \vdash \text{wp}\ b\ Q}$$

This simply states that any property that holds of the specification, holds of the implementation. We have demonstrated that some intuitively reasonable properties are not preserved by refinement—we need to make sure we choose one that is.

Returning to pGCL, we have a novel notion of entailment. We write[2] $P \vDash Q$ for comparison defined pointwise i.e. $\forall s.\ P\ s \leqslant Q\ s$. If $P \vDash Q$, then P has a lower value in every state than does Q. We also write[3] «P» for the embedding of the boolean predicate P as real-valued function (an *expectation*):

$$\text{«P»}\ s = \text{if}\ P\ s\ \text{then}\ 1\ \text{else}\ 0 \quad \text{noting that} \quad P \vdash Q \leftrightarrow \text{«P»} \vDash \text{«Q»}$$

We can now model our guessing attack as:

$$(h := 0 \sqcap h := 1)\ ;;\ (l := 0\ {}_{1/2}\!\oplus\ l := 1) \tag{4.1}$$

[1] A trace is any sequence of valid states of a program. Trace refinement implies that the implementation is allowed to take any step that the specification can take.

[2] We differ in syntax, as the symbol $\Rrightarrow$ is not readily available within Isabelle

[3] Again we differ, as the established syntax, $[\cdot]$, clashes with lists.

where the secret (h) is chosen nondeterministically, and the guess (l) randomly ($a \ {}_p\!\oplus\ b$ denotes probabilistic choice between programs a and b, with probability p for a).

We inherit a family of structural refinement rules, for example $a \sqcap b \sqsubseteq a$ (a choice is refined by either alternative), and the following relations:

$$\text{WP\_REFINES}\quad \frac{a \sqsubseteq b}{\text{wp}\ a\ Q \vDash \text{wp}\ b\ Q} \quad \text{whence} \quad \frac{P \vDash \text{wp}\ a\ Q}{P \vDash \text{wp}\ b\ Q}$$

which are the probabilistic equivalents of the previous refinement conditions. We have therefore, that $h := 0\ ;;\ (l := 0\ {}_{1/2}\!\oplus\ l := 1)$ is a refinement of Equation 4.1, but importantly, $(h := 0 \sqcap h := 1)\ ;;\ l := h$ is not. In contrast to demonic choice, probabilistic choice cannot be refined away.

A refinement in pGCL establishes a predicate with *at least as high a probability* as does its specification. Thus if our predicate is 'the system is secure', and we are content with establishing its minimum probability, refinement *by definition* only increases this.

This reasoning applies only to security predicates: security properties that can be established by inspecting only the current state. Not all properties can be expressed in this way: some can only be expressed by considering some set of possible traces of the system. Our guessing-attack model, however, is carefully constructed such that security *is* only dependent on the current state: a system is secure (by definition), if the attacker has not yet guessed the secret. While this property does refer to traces (specifically, the history from any point), this can be modelled using a shadow variable: the list of past actions is appended to the state. This is the approach we take in Chapter 6, where we formally derive leakage bounds on a guessing attack in pGCL.

The guessing attack model is thus refinement sound in pGCL.

4.2 The pGCL Language

We now summarise pGCL more completely, noting our small variations in syntax relative to the standard presentation. This summary is naturally incomplete, and the interested reader is directed to the aforementioned work of McIver and Morgan [2004].

Programs in pGCL have two interpretations: The first is as a probabilistic automaton, moving from a starting state to some final state, with well-defined

probability. The second is as an *expectation transformer*, mapping a real-valued function on final states (a post-expectation), to one on initial states (a pre-expectation). The *weakest pre-expectation* (wp, overloading this term) of a post-expectation at some initial state is the smallest *expected value* (minimised over demonic choices) of the post-expectation in the final state. For example, the weakest pre-expectation of the expression x, under the program

$$(x := 1 \;{}_{1/2}\!\oplus\; x := 0) \sqcap (x := 2 \;{}_{1/3}\!\oplus\; x := 1)$$

is

$$\min\left(\frac{1}{2}\times 1 + \frac{1}{2}\times 0\right)\left(\frac{1}{3}\times 2 + \frac{2}{3}\times 1\right) = \frac{1}{2}$$

We mechanise the expectation-transformer interpretation, but the two are equivalent. The probabilistic automaton is generally more intuitive, giving the most straightforward way to visualise results.

Programs are constructed using several operators, including some already mentioned:

- Sequential composition: $a\,;;\,b$.

  Composition is interpreted either as 'do a, then b', in the automaton case, or as 'apply a to the result of b' in the expectation transformer case.

- Name binding: n is f in $a\;n$, where f is a function from state to value.

  This is purely syntactic, and simply binds a complex expression for reuse. This primitive is novel to our formalisation, but is a common feature of functional languages.

- Applying a state transformer: Apply f.

  This is the primitive operator for state updates. It is usually hidden by the syntax translations for records described below.

- Demonic choice: $a \sqcap b$ or $\bigsqcap x \in S.\; a\;x$

  The first of these is binary choice, which can be repeated in the obvious way to express any finite choice. The second is convenient for expressing a either a large finite choice or infinite branching. In doing so, we must be careful that the program remains well-defined. For example, wp $(\bigsqcap x \in \mathbb{Z}.\ \text{Apply}\ (\lambda\_.\ x)\ x$ is *not* well-defined: there is no minimum (or even a well-defined infimum) for x's final value.

Rather than enforce this syntactically or through types, the user is forced to discharge a verification condition that the primitive is well formed, for example by appealing to finiteness. Our reason for this is that we will later (in Chapter 6) need to allow well-defined infinite choices, in order to abstract over possible strategies for an attacker.

- Probabilistic choice: $a \ {}_p\!\oplus \ b$, or $[]\, x \in S \cdot a\, x \,@\, P\, x$

  As for deterministic choice, but constrained by a particular distribution. The transformer interpretation is as the sum over possibilities, weighted by probability. This primitive is well defined even for infinite choices.

- Finite repetition: $a^n = \underbrace{a \,;; \ldots ;; a}_{n}$.

- Lifting from a non-probabilistic monad: Exec M.

  This connects with our existing work on monadic Hoare logic [Cock et al., 2008], by embedding a monad into pGCL. We return to this in Section 4.4, and again in Section 5.2, where we demonstrate its application and the lifting of Hoare triples.

- Recursion: $\mu x.\ T\, x$

  Recursion is defined as the fixed point of a single recursive equation. The μ operator binds a transformer (of type $(\sigma \to \mathbb{R}) \to \sigma \to \mathbb{R}$), to which the recursive equation T is applied. We expand on this in Section 4.4.

- Looping: do G $\to$ a od

  This is syntactic sugar for recursion, see Equation 4.3.

- Failure: **abort** & Success: **skip**

  The two remaining primitives are the diverging program **abort**, and the do-nothing program **skip**.

While programs may have any state type, in practice we use Isabelle's *record types*: tuples with labelled fields, analogous to C structures. The advantage is that through syntax translations, we are able to use the field identifiers

as pGCL variable names. For example:

$$x := v \Longrightarrow \mathrm{Apply}\ (\lambda s.\ x\_update\ (\lambda\_.\ v)\ s)$$
$$x :\in S \Longrightarrow \bigsqcap s \in S.\ x := s$$
$$x :\in S \text{ at } P\ x \Longrightarrow [\!]\ s \in S \cdot x := s\ @\ P\ s$$

The assertion language is shallowly embedded, closely resembling that of GCL. There are a few novel probabilistic constructions:

- Entailment: $P \vDash Q \Leftrightarrow \forall s.\ P\ s \leqslant Q\ s$ usually $\Rrightarrow$
- Conjunction: $P\ \&\&\ Q \Leftrightarrow \lambda s.\ \max 0\ (P\ s + Q\ s - 1)$ usually &
- Embedding: $«P» \Leftrightarrow \lambda s.\ \text{if } P\ s \text{ then } 1 \text{ else } 0$ usually $[P]$

We have already introduced probabilistic entailment and embedding. The form of probabilistic conjunction is chosen for compatibility with its boolean equivalent: $«P»\ \&\&\ «Q» = «\lambda s.\ P\ s \wedge Q\ s»$. We use this particular form (rather than, for example $P\ \&\&\ Q = \lambda s.\ P\ s \times Q\ s$, which gives the same results on embedded predicates) for technical reasons concerning the underlying semantic interpretation[4].

The following is an example specification in expectation-entailment style that illustrates the essential features of the logic:

$$«P»\ \&\&\ (\lambda\_.\ p) \vDash wp\ (a\ ;;\ b)\ «Q»$$

This states that from any initial state satisfying P, after executing a followed by b, we reach a state satisfying Q with probability *at least* p.

4.3 The pGCL Theory Package

We present a package of related theories, that implement a shallow embedding of pGCL in Isabelle/HOL. The sources can be found in the `pGCL` subdirectory

[4]Briefly, the definition given is the only option that is *sub-linear*, a generalisation (to real-valued functions), and weakening, of the *linearity* condition required of expectation transformers in pure GCL. All sub-linear transformers are linear, and sub-linearity reduces to linearity in the case of embedded boolean predicates, but (for example) demonic choice, $a \sqcap b = \lambda s.\ \min\ (a\ s)\ (b\ s)$, is not linear if a or b take values other than 0 and 1. All pGCL primitives are sub-linear. For further details, see McIver and Morgan [2004].

$$
\begin{aligned}
\mathrm{wp}\ \mathbf{abort}\ R &= \lambda s.\ 0 \\
\mathrm{wlp}\ \mathbf{abort}\ R &= \lambda s.\ \mathrm{bound\_of}\ R \\
\mathrm{wp}\ \mathbf{skip}\ R &= R \\
\mathrm{wp}\ (\mathrm{Apply}\ f)\ R &= \lambda s.\ R\ (f\ s) \\
\mathrm{wp}\ (a\ ;;\ b)\ R &= \mathrm{wp}\ a\ (\mathrm{wp}\ b\ R) \\
\mathrm{wp}\ (a \sqcap b)\ R &= \lambda s.\ \min\ (\mathrm{wp}\ a\ R\ s)\ (\mathrm{wp}\ b\ R\ s) \\
\mathrm{wp}\ (\bigsqcap x \in S.\ a\ x)\ R &= \lambda s.\ \inf_{x \in S s}\ \mathrm{wp}\ a\ x\ R\ s \\
\mathrm{wp}\ (a\ {}_p\!\oplus\ b)\ R &= \lambda s.\ p\ s \times \mathrm{wp}\ a\ R\ s + (1 - p\ s) \times \mathrm{wp}\ b\ R\ s \\
\mathrm{wp}\ (\llbracket x \in S \cdot a\ x\ @\ P\ x)\ R &= \lambda s.\ \sum_{x \in S s} P\ x \times \mathrm{wp}\ a\ x\ R\ s \\
\mathrm{wp}\ (\mu x.\ T\ x)\ R &= \mathrm{lfp}\ T\ R \\
\mathrm{wlp}\ (\mu x.\ T\ x)\ R &= \mathrm{gfp}\ T\ R
\end{aligned}
$$

Figure 4.1: The expectation-transformer interpretation for both wp and wlp.

of the attached material. With the few previously noted exceptions, we retain existing syntax. The advantage of a shallow embedding is the ease with which we can apply the existing machinery of Isabelle/HOL to the underlying arithmetic. The disadvantage is that we cannot appeal to results shown on a stricter type, in particular Isabelle's fixed-point lemmas, which are tied to the *typeclass* of complete lattices. We return to this point in Section 4.4.

As mentioned, the language models imperative computation, extended with both nondeterministic and probabilistic choice. Nondeterminism (by default) is demonic, with respect to the postcondition of a program: A demonic choice is always assumed to be taken so as to minimise the likelihood of the postcondition being satisfied.

The Expectation-Transformer Model

As described, the intuitive interpretation of a program, its operational semantics, is as a probabilistic automaton: from a given state, the program chooses the next randomly. The program is thus a *forward* transformer, taking a distribution on initial states to one on final states.

For mechanisation, we are more interested in the alternative interpretation of a program: as a *reverse* transformer. Here, the program maps a function

on the final state to one on the initial state: a 'real-valued predicate'. These generalised predicates are the *expectations* of Section 4.1, and are the bounded, non-negative functions from the state space σ, to $\mathbb{R}$:

$$\text{bounded\_by } b\ P = \forall s.\ P\ s \leqslant b \qquad \text{bounded } P = \exists b.\ \text{bounded\_by } b\ P$$
$$\text{nneg } P = \forall s.\ 0 \leqslant P\ s \qquad \text{sound } P = \text{bounded } P \wedge \text{nneg } P$$

The strict (wp), and liberal (wlp) interpretations are given in Figure 4.1. The liberal interpretation differs only for **abort** and μ: the distinction is explained in Section 4.4. That the forward and reverse interpretations are equivalent has been established [McIver and Morgan, 2004], although we have not mechanised the proof.

To see that this model provides the probabilistic weakest precondition demanded in Section 4.1, note that the expectation «R» : $\sigma \to \mathbb{R}_{\geqslant 0}$ gives, trivially, the probability that the predicate R holds (the state is of type σ) . This interpretation is preserved under transformation: wp prog R s_{initial} is the probability that R will hold in the final state, if prog executes from s_{initial}. Equivalently, it is the expected value of the predicate after executing, minimised over all demonic choices: $\sum_{s_{\text{final}}} P(s_{\text{final}}|s_{\text{initial}}) \times$ «R» s_{final}, hence the term expectation.

A program is modelled as a function from expectation to expectation: $(\sigma \to \mathbb{R}) \to \sigma \to \mathbb{R}$. This maps a post-expectation to its weakest pre-expectation. For a *standard* post-expectation, the embedding of a predicate (e.g. «P»), this is the greatest lower bound on the likelihood of it holding in the final state.

Not all transformers (functions of type $(\sigma \to \mathbb{R}) \to \sigma \to \mathbb{R}$) are valid. We impose a number of *healthiness* conditions (slightly weaker than the standard versions given by McIver and Morgan [2001]), that define well-behaved transformers. We combine the treatment of strict transformers (weakest precondition, giving total correctness) and liberal transformers (weakest *liberal* precondition, giving partial correctness) by working in the union of their domains. This basic healthiness is defined as the combination of *feasibility*, *monotonicity* and *weak scaling*:

$$\text{feasible } t = \forall P.\ \text{bounded\_by } b\ P \wedge \text{nneg } P \to \text{bounded\_by } b\ (t\ P) \wedge \text{nneg } (t\ P)$$
$$\text{mono\_trans } t = \forall P\ Q.\ (\text{sound } P \wedge \text{sound } Q \wedge P \vDash Q) \to t\ P \vDash t\ Q$$
$$\text{scaling } t = \forall P\ s.\ (\text{sound } P \wedge 0 < c) \to c \times t\ P\ s = t\ (\lambda s.\ c \times P\ s)\ s$$

$$
\begin{aligned}
\text{hide} &= \mathrm{P} := 1 \sqcap \mathrm{P} := 2 \sqcap \mathrm{P} := 3 \\
\text{guess} &= \mathrm{G} := 1 \ {}_{\lambda s.\ 1/3}\!\oplus (\mathrm{G} := 2 \ {}_{\lambda s.\ 1/2}\!\oplus \ \mathrm{G} := 3) \\
\text{reveal} &= \mathrm{C} :\in \big(\lambda s.\ \{1,2,3\} - \{\mathrm{P}\ s, \mathrm{G}\ s\}\big) \\
\text{switch} &= \mathrm{G} :\in \big(\lambda s.\ \{1,2,3\} - \{\mathrm{C}\ s, \mathrm{G}\ s\}\big)
\end{aligned}
$$

$$
\text{monty } \textit{switch} = \text{hide} ; ; \text{guess} ; ; \text{reveal} ; ; \text{if } \textit{switch} \text{ then switch else } \mathbf{skip}
$$

Figure 4.2: The Monty Hall game in pGCL.

Stronger results are established on-the-fly by appealing to one of several supplied rule sets.

A well-defined program has healthy strict and liberal interpretations, related appropriately:

$$
\begin{aligned}
\text{well\_def } a = \text{healthy } (\text{wp } a) \wedge \text{healthy } (\text{wlp } a) \wedge \\
(\forall P.\ \text{sound } P \to \text{wp } a \vDash \text{wlp } a)
\end{aligned}
$$

Reasoning with pGCL

The rules in Figure 4.1 evaluate the weakest pre-expectation of non-recursive program fragments structurally. If the resulting term is not too large, the simplifier can calculate it exactly. We also support two other approaches: Modular reasoning by structural decomposition, and the verification condition generator (VCG). Examples are given in Isabelle proof script.

Consider Figure 4.2, a model of the Monty Hall problem [Selvin, 1975; Hurd et al., 2005]. The scenario is a game show: A prize is hidden behind one of three doors (hide), of which the contestant then guesses one (guess). The host then opens a *different* door, (reveal), showing that it does not hide the prize. The contestant now chooses: to switch to the unopened door (switch), or to stick to the original (**skip**). Is the contestant is better off switching?

The victory condition, and hence the probability of the contestant winning and from a given starting state s, is given by the weakest precondition:

$$
\text{win } g = (\mathrm{G}\ g = \mathrm{P}\ g) \qquad \mathrm{P}(\text{win}) = \text{wp } (\text{monty } \textit{switch}) \ \text{«win»}\ s
$$

Proof by unfolding If *switch* is false, we can solve by explicitly unfolding the rules in Figure 4.1. This approach was demonstrated by Hurd et al. [2005] As expected, the contestant has a 1/3 chance of success:

```
lemma wp_monty_switch: "λs. 1/3 ⊨ wp monty false «wins»"
unfolding monty_def hide_def guess_def reveal_def switch_def
  by(simp add:wp_eval insert_Diff_if)
```

Proof by decomposition If *switch* is true the state space grows dramatically, and such a straightforward proof rapidly becomes infeasible (though it is still just possible in this case). Modular reasoning lets us scale further. Luckily, the weakest precondition semantics of pGCL admit familiar composition rules:

WP_STRENGTHEN_POST

$$\frac{P \vDash \mathrm{wp}\ p\ Q \quad Q \vDash R \quad \mathrm{healthy}\ (\mathrm{wp}\, p) \quad \mathrm{sound}\ Q \quad \mathrm{sound}\ R}{P \vDash \mathrm{wp}\ p\ R}$$

WP_SEQ

$$\frac{Q \vDash \mathrm{wp}\ b\ R \qquad P \vDash \mathrm{wp}\ a\ Q \quad \mathrm{healthy}\ (\mathrm{wp}\ a) \quad \mathrm{healthy}\ (\mathrm{wp}\ b) \quad \mathrm{sound}\ Q \quad \mathrm{sound}\ R}{P \vDash \mathrm{wp}\ (a\,;;\,b)\ R}$$

The healthiness and soundness obligations result from the shallow embedding.

To integrate with Isabelle's calculational reasoner, we define probabilistic Hoare[5] triples:

WP_VALIDI

$$\frac{P \vDash \mathrm{wp}\ a\ Q}{\{P\}\ a\ \{Q\}}$$

WP_VALIDD

$$\frac{\{P\}\ a\ \{Q\}}{P \vDash \mathrm{wp}\ a\ Q}$$

VALID_SEQ

$$\frac{\{P\}\ a\ \{Q\} \qquad \{Q\}\ b\ \{R\} \quad \mathrm{healthy}\ (\mathrm{wp}\ a) \quad \mathrm{healthy}\ (\mathrm{wp}\ b) \quad \mathrm{sound}\ Q \quad \mathrm{sound}\ R}{\{P\}\ a\,;;\,b\ \{R\}}$$

Note that VALID_SEQ is simply the composition of WP_COMPOSE with WP_VALIDI and WP_VALIDD.

[5]The classical Hoare triple $\{P\}\ a\ \{Q\}$ states that if P holds initially then after executing a, Q is guaranteed to hold.

We need one final rule, peculiar to pGCL and its real-valued expectations:

$$\frac{\text{WP\_SCALE}}{P \vDash \text{wp } a\ Q \quad \text{healthy } (\text{wp } a) \quad \text{sound } Q \quad 0 < c}{(\lambda s.\ c \times P\ s) \vDash \text{wp } a\ (\lambda s.\ c \times Q\ s)}$$

This follows from the healthiness of the transformer, and allows us to scale the pre- and post-expectations such that the latter 'fits under' some target. To illustrate, consider the 'obvious' specification of hide:

$$\lambda s.\ 1 \vDash \text{wp hide } «p \in \{1,2,3\}» \tag{4.2}$$

This states that with probability 1, the prize ends up behind door 1, 2 or 3. In evaluating our preconditions stepwise, however, we find that the weakest precondition of the remainder of the program is in fact:

$$\lambda s.\ 2/3 \times «p \in \{1,2,3\}»\ s$$

Applying rule WP_SCALE to Equation 4.2 we derive a scaled rule:

$$\lambda s.\ 2/3 \vDash \text{wp hide } (\lambda s.\ 2/3 \times «p \in \{1,2,3\}»\ s)$$

Finally the probability of success if the contestant switches is at least[6] 2/3:

```
declare valid_Seq[trans]
lemma wp_monty_switch_modular: "λs. 2/3 ⊨ wp monty true «wins»"
proof(rule wp_validD)
    note wp_validI[OF wp_scale, OF wp_hide, simplified]
    also note wp_validI[OF wp_guess]
    also note wp_validI[OF wp_reveal]
    also note wp_validI[OF wp_switch]
    finally show "λs. 2/3 ⊨ wp monty true «wins»"
      unfolding monty_def
      by(simp add:healthy_intros sound_intros monty_healthy)
  qed
```

Here, we take advantage of the calculational reasoning facility of Isabelle, as described by Bauer and Wenzel [2001]. The intermediate Hoare triples are automatically derived, by applying VALID_SEQ to the previous relation and the supplied specification. Finally, we again discharge all side conditions using the simplifier.

[6] In fact it is exactly 2/3, but our object is to demonstrate an entailment proof. In more complicated situations, calculating the exact pre-expectation is impractical.

Proof by VCG Alternatively, we can pass the component specifications to our verification condition generator (VCG), which follows a similar strategy to the above, automatically matching the appropriate rule to the goal. The VCG leaves behind an inequality between the target pre-expectation and that calculated internally (generally not the weakest). In this case, the final goal is trivial enough to be discharged internally.

```
lemmas scaled_hide = wp_scale[OF wp_hide, simplified]
declare scaled_hide[wp] wp_guess[wp] wp_reveal[wp] wp_guess[wp]
declare healthy_wp_hide[health] healthy_wp_guess[health]
        healthy_wp_reveal[health] healthy_wp_switch[health]
lemma wp_monty_switch_vcg: "λs. 2/3 ⊨ wp monty true «wins»"
  unfolding monty_def by(simp,pvcg)
```

The above proofs are available in `pGCL/Examples/Monty.thy`.

Loops and Recursion

We cannot unfold loops syntactically, as we would simply recurse endlessly. The treatment of recursion in pGCL is well developed, and we incorporate some of this work, specifically regarding loops. This area is still under development, but we already provide several useful rules, including this, which is a specialisation of lemma 7.3.1 of McIver and Morgan [2004]. This gives the correctness condition for standard post-expectations on loops which terminate[7] with probability 1:

WP_LOOP

$$\frac{\begin{array}{c}\text{well\_def}\ (\text{wp}\ \mathit{body}) \\ \text{sub\_distrib}\ (\mathbf{do}\ \text{G} \to \mathit{body}) \quad (\lambda\text{s.}\ \text{«G»}\ \text{s} \times \text{«I»}\ \text{s}) \vDash \text{wp}\ \mathit{body}\ \text{«I»}\end{array}}{\text{«I»} \mathbin{\&\&} \text{wp}\ (\mathbf{do}\ \text{G} \to \mathit{body})\ (\lambda\text{s.}\ 1) \vDash \text{wp}\ (\mathbf{do}\ \text{G} \to \mathit{body})\ (\lambda\text{s.}\ \text{«}\mathcal{N}\text{G»} \times \text{«I»})}$$

Here, $\mathcal{N}$G is the negation of the guard: λs. $\neg$G s.

4.4 Implementation and Extensions

We now expand on some of the more interesting details of the implementation, and how it can be extended with new primitives. A significant amount

[7] This is a weaker condition than terminating along all paths: Non-terminating traces with probability 0 are acceptable. Imagine flipping a coin until it shows heads.

$$
\begin{aligned}
a\ ;;\ b &= \lambda a\,b.\ (a\ a\,b) \circ (b\ a\,b) \\
\textbf{abort} &= \lambda a\,b.\ \text{if } a\,b \text{ then } \lambda s.\ 0 \text{ else } \lambda s.\ \text{bound\_of } P \\
\text{Embed } f &= \lambda a\,b.\ f \\
\mu x.\ \text{prog } x &= \lambda a\,b.\ \text{if } a\,b \text{ then lfp\_trans } (\lambda t.\ \text{prog } (\text{Embed } t)\ a\,b) \\
&\qquad\qquad\qquad\quad \text{else gfp\_trans } (\lambda t.\ \text{prog } (\text{Embed } t)\ a\,b) \\
\text{wp } a &= a\ \text{True} \\
\text{wlp } a &= a\ \text{False}
\end{aligned}
$$

Figure 4.3: The underlying definitions of selected pGCL primitives.

of legwork was unfortunately necessary, as the existing infrastructure (the fixed-point theory) could not be straightforwardly applied to our semantic structures, as we will shortly see.

Implementing wp and wlp

Figure 4.3 details the implementation of several primitives, together with the definitions of wp and wlp. Programs are represented as their associated transformer, parameterised by the treatment of **abort** (the parameter $a\,b$), giving either strict or liberal semantics. Only **abort** and μ change their behaviour between wp and wlp: The former gives either failure ($\lambda P\,s.\ 0$) or success ($\lambda P\,s.$ bound_of P), whereas the latter is the least or the greatest fixed point, respectively. All others, as for ($;;$), simply pass $a\,b$ inward.

Consequences of a Shallow Embedding

The very shallow embedding used has two important consequences, the first of which is negative. The healthiness of transformers, and soundness of expectations, must be explicitly carried as assumptions. A deeper embedding, such as that of Hurd et al. [2005] could restrict to the type of healthy transformers, in which case these would be satisfied by the type axioms.

We avoid such an embedding to reuse as much of the mechanisation within Isabelle/HOL as possible. Reasoning within a fresh type requires lifting (and modifying) all necessary rules. The recent integration of the lifting package into Isabelle makes doing this simpler, but there is still a large body of existing automation that would need to be reinvented. The burden of discharging

```
type_synonym (σ, α) nondet_monad =
      σ ⇒ (α × σ) set × bool
{P} f {Q} = ∀s. P s → (∀(r, s′) ∈ fst (f s). Q r s′)
no_fail P m = ∀s. P s → ¬(snd (m s))

Exec :: (σ, α) nondet_monad ⇒ σ prog
Exec M = λa b R s.
  let (SA, f) = M s in                                  Run the monad
    if f then abort a b R s                             Fail is Abort
        else if SA = {} then (bound_of R)               Stuck is Success
                        else let S = snd ‴ SA in        Ignore result
                             glb (R ‴ S)                Infimum over states
```

Figure 4.4: The L4.verified non-deterministic monad in Isabelle.

our side conditions is, moreover, not high. For any primitively constructed program, healthiness follows by invoking the simplifier with the appropriate lemmas.

The positive consequence of an extremely shallow embedding is the ease with which it can be extended. We have already seen an example: the definition of demonic choice from a set (following a standard abbreviation [McIver and Morgan, 2004]). To do so, one need only supply weakest-precondition (and weakest-liberal-precondition) rules, rules to infer healthiness and (optionally) rules for proof decomposition.

It is not necessary to show that the new primitive is sound, that is, produces a healthy transformer for all inputs. It is merely necessary that the supplied rules show healthiness for just those cases in which it is actually used. Set demonic choice is just such a partially sound primitive: Healthiness does not generally hold for infinite sets. Applied to finite sets, as it is here, the supplied rules establish healthiness.

As a further example, we embed the non-deterministic monad at the heart of the L4.verified proof. The existing definition, Figure 4.4, is as a function from states (σ) to a set of result (α), state pairs. The extra result is the failure flag, used to explicitly signal failure. This was added to ensure that termination is preserved under refinement, as described elsewhere [Cock et al., 2008].

We embed as follows: Stuck (no successor states) is success, for compatibility with our infimum-over-alternatives interpretation. Explicit failure is **abort**, which is in turn either success or failure under wp or wlp, respectively. We lift results as follows:

WP_EXEC
$$\frac{\{P\}\ prog\ \{\lambda r\ s.\ Q\ s\} \quad \text{no\_fail}\ P\ prog \quad \exists s.\ P\ s}{\text{«P»} \vdash \text{wp}\ prog\ \text{«Q»}}$$

WLP_EXEC
$$\frac{\{P\}\ prog\ \{\lambda r\ s.\ Q\ s\} \quad \exists s.\ P\ s}{\text{«P»} \vdash \text{wlp}\ prog\ \text{«Q»}}$$

Note that the difference between wp and wlp is simply termination.

Induction and the Lattices of Expectations and Transformers

Handling recursion means reasoning about fixed points. In this case, we need both least and greatest, on expectations and transformers. Due to the shallow embedding, we cannot appeal to the existing fixed point results, which are phrased on a complete lattice. Neither the underlying type for expectations ($\alpha \to \mathbb{R}$) or for transformers ($(\alpha \to \mathbb{R}) \to \alpha \to \mathbb{R}$) can be so instantiated, due to the lack of both top and bottom elements. The solution in each case is different.

Sound expectations have an obvious bottom element, $\lambda s.\ 0$, but there is no universal upper bound. We only require that there exists a bound for any given expectation. There need not exist any bound on an arbitrary set of sound expectations. For example, with $\alpha = \mathbb{N}$, consider the set

$$\{\lambda s.\ \text{if } s = n \text{ then } n \text{ else } 0\ :\ n \in \mathbb{N}\}.$$

Each expectation is bounded (by n) and non-negative, yet the least upper bound, $\lambda s.\ s$ is itself unbounded.

We need a surrogate for the top element. To illustrate, take our definition for greatest fixed point:

$$\text{gfp\_in}\ f\ S = \text{if}\ \exists x \in S.\ x \leqslant f\ x\ \text{then lub}\ \{x \in S.\ x \leqslant f\ x\}\ \text{else lowerbound}\ S$$

A lower bound is easy ($\lambda s.\ 0$), but in order to find the *least* upper bound, we need *some* upper bound. In a complete lattice, this is the top element. Instead, we appeal to feasibility:

$$\text{bound\_of}\ ((\mu x.\ f\ x)\ Q) \leqslant \text{bound\_of}\ Q$$

and thus

$$(\mu x.\ f\ x)\ Q \leqslant \lambda s.\ \text{bound\_of}\ Q\ .$$

It is therefore sufficient to consider fixed points that are *weakly bounded* by Q:

$$\text{weakly\_bounded\_by}\ Q = \{R.\ \text{sound}\ R \wedge \text{bound\_of}\ R \leqslant \text{bound\_of}\ Q\}$$

Finally, we establish the standard fixed-point results parameterised by Q, for example:

GFP_IN_UNFOLD

$$\frac{\text{healthy}\ t}{\text{gfp\_in}\ t\ (\text{weakly\_bounded\_by}\ P) = t\ (\text{gfp\_in}\ t\ (\text{weakly\_bounded\_by}\ P))}$$

The case of transformers is simpler, again due to feasibility:

$$t\ P\ s \leqslant \text{bound\_of}\ P \quad \text{and thus} \quad t \leqslant \lambda P\ s.\ \text{bound\_of}\ P$$

Thus we have a top element, and establish a complete lattice by means of a quotient [Huffman, 2012].

$$\text{le\_trans}\ t\ u = \forall P.\ \text{sound}\ P \to t\ P \leqslant u\ P$$

$$\text{equiv\_trans}\ t\ u = \text{le\_trans}\ t\ u \wedge \text{le\_trans}\ u\ t$$

$$\text{htrans\_real}\ t\ u = \text{healthy}\ t \wedge \text{healthy}\ u \wedge \text{equiv\_trans}\ t\ u$$

$$\textbf{quotient\_type}\ \sigma\ \text{trans} = (\sigma \to \mathbb{R}) \to \sigma \to \mathbb{R}\ /\ \text{partial} : \text{htrans\_rel}$$

Using the induced homomorphism, we draw back the standard results:

GFP_TRANS_UNFOLD

$$\frac{\begin{array}{c}\wedge t.\ \text{healthy}\ t \vdash \text{healthy}\ (T\ t) \\ \wedge t\ u.\ [\text{healthy}\ t; \text{healthy}\ u; \text{le\_trans}\ t\ u] \vdash \text{le\_trans}\ (T\ t)\ (T\ u)\end{array}}{\text{equiv\_trans}\ (\text{gfp\_trans}\ T)\ (T\ (\text{gfp\_trans}\ T))}$$

The Verification Condition Generator

The VCG (tactic `pvcg`), is simple but nonetheless capable handling Figure 4.2. It alternates applying an entailment rule, and attempting to discharge side-goals using internal and user-supplied rules.

The user supplies specifications as proved entailment rules, tagged with `[wp]`, and healthiness rules, tagged with `[health]`. Internal rules are tagged `[wp_core]`. The VCG selects rules in this order:

1. User-supplied rules, as written.
2. User-supplied rules, with a strengthened postcondition: `[wp_strengthen_post[OF rule]]`. This leaves a side-goal that the postcondition of the supplied rule entails the strengthened version.
3. Internal rules.

A user-supplied rule will override an internal rule, and may refer directly to a compound structure e.g. $P \vDash \mathrm{wp}\ (a\ ;;\ b)\ Q$. The given rule will be used rather than unfolding the composition. If no user rule is found, the VCG will proceed using its internal rules, calculating the exact weakest precondition by unfolding.

4.5 Related Work

The pGCL language of McIver and Morgan [2004] extends the treatment of conventional nondeterministic programs, particularly that of Dijkstra [1975], from which the core syntax is derived. Treatments of the semantics of probabilistic programs had previously appeared, e.g. Kozen [1985], although these tended to abandon classical nondeterministic choice in favour of probabilistic choice. The work of McIver et. al. was the first to uniformly treat both forms of nondeterminism, in a standard verification framework.

Mathematically, calculating the weakest pre-expectation of program in pGCL is known to be equivalent [Gretz et al., 2014] to finding the expected reward for an appropriately-constructed Markov decision process (MDP) [White and White, 1989]. There now exists a formal treatment of Markov models (including rewards) in Isabelle, due to Hölzl and Nipkow [2012]. We plan to unify this with our formalisation of pGCL, to provide it with an operational semantics, and a formal connection to Isabelle's existing probability theory.

Interactive theorem provers, such as Isabelle, already host many existing models of programming language semantics [Nipkow, 2002; Mossakowski et al., 2010; Cock et al., 2008; Harrison and Kieburtz, 2005]. Relative to these, the novelty here is that pGCL allows us to treat probabilistic properties and programs.

A previous formalisation of pGCL, in the HOL4 theorem prover, was presented by Hurd et al. [2005]. The principal difference to this work is the use of a deep, rather than shallow embedding. As we have described, our shallow

embedding allows us to take advantage of the powerful tool and theory support that Isabelle offers. The second advantage is the ease with which we are able to integrate other shallowly-embedded models, as demonstrated with the nondeterministic monad. Our worked example, Monty Hall, is adapted from theirs.

Coble [2010] investigated the problem of formally verifying anonymity in a theorem prover (HOL4). While the specific motivation is different (anonymity versus secrecy), the problems are quite close, and Coble also treats probability at length, providing a substantial library of formalised probability theory. Rather than taking such a ground-up approach, we have adopted an existing logic (pGCL), and tackle probability in a comparatively simplistic manner, which has nevertheless proved adequate. Instead, we have invested more effort in developing a framework that should scale to larger, more realistic systems, and in evaluating (and verifying) leakage measures other than Shannon entropy.

While we approach this area from the perspective of system security, others have arrived at similar approaches in the field of cryptography. Barthe et al. [2009] presented an automated theorem prover, CertiCrypt, tailored to verifying provably-correct cryptographic algorithms. This tool has been superseded by the authors' more recent EasyCrypt [Barthe et al., 2012b].

The language in which algorithms are expressed in these tools is quite similar to pGCL, incorporating true probabilistic choice alongside recursion. Although expressed in different terms, the language of Barthe et. al. incorporates a notion of refinement: the *monotone transformations*. These are transformations of a system model (here a game) that never decrease the probability of certain events. For game G and event A, a transformation h is monotone if (in the author's syntax) $\mathsf{Pr}_G[A] \leqslant h(\mathsf{Pr}_{G'}[A']$, or the probability of the transformed equivalent of event A, A', in the transformed game G', is at least as high as that of A in G. This is exactly the notion of refinement introduced in Section 4.1 for pGCL—refinement *increases* the probability of events.

The treatment of classical nondeterminism in CertiCrypt and EasyCrypt is not as thorough as that of pGCL. Nondeterminism (expressed by distributions that sum to less than one) is only introduced by non-termination or (for EasyCrypt) by assertions. In contrast, pGCL also provides nondeterministic choice as a language primitive, which is essential in order to model classical specification–implementation refinement. It would probably be possible to

unify the two languages, although the EasyCrypt semantics would need to be modified. In particular, the transformers of Barthe et al. [2011] are assumed to be *additive* i.e. $\mu(f + g) = \mu f + \mu g$. Transformers in pGCL, however, are only *sub-additive*: $\mu f + \mu g \leqslant \mu(f + g)$, and it is precisely nondeterministic choice that introduces *strictly* sub-additive transformers (recall that a nondeterministic choice gives the least probability among the alternatives). The other assumptions (monotonicity, multiplicative linearity, continuity) map precisely onto the healthiness conditions of pGCL. Unifying the two approaches would be an interesting and potentially valuable project.

4.6 Summary

In this chapter, we stepped back from the low-level approach of Chapter 3, to consider how we might reason formally about the sorts of stochastic behaviour that we see in real covert and side channels. In particular, we need to find an approach that fits with the refinement-driven correctness proof of seL4, which forms the testbed for our more practical work.

Our approach is to take the existing language pGCL, and mechanise its logic in the Isabelle/HOL theorem prover, as used in the L4.verified project. We have demonstrated a useful degree of automation, and the integration of the nondeterministic state monad that underlies the seL4 specification. We return to this specific detail at more length in the following chapter, where we demonstrate that our mechanisation can be used to prove probabilistic security properties for real systems.

5 Case Study — Lattice Scheduling

This chapter draws on work first presented in the following paper:
David Cock. Practical probability: Applying pGCL to lattice scheduling. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, pages 1–16, Rennes, France, July 2013. Springer.
doi:10.1007/978-3-642-39634-2_23

As a more substantial example than that of Figure 4.3, we attack a much larger, and more obviously security-focused problem. Our goal is to demonstrate that mechanically verifying realistic probabilistic software, with probabilistic properties, is feasible. Our 'realistic probabilistic program' is a hybrid probabilistic lattice-lottery scheduler, designed to minimise the number of flushes required to completely close the cache channel (described in Section 3.2), while guaranteeing fairness and remaining simple and efficient.

The probabilistic properties of our program are: stochastic fairness — that the probability of starvation for any domain is zero, and non-leakage — that the distribution of observable outputs is independent of hidden inputs. Finally, we make our argument for feasibility by showing that we can incorporate the existing seL4 results in a probabilistic setting. We are able to restrict probabilistic reasoning to small regions, allowing the remainder of the proof to proceed in a traditional manner.

We begin with an abstract, nondeterministic specification, which we refine iteratively. Our first refinement is to a probabilistic version, and then to a practical implementation based on lottery scheduling. We demonstrate that this refinement could be continued using the L4.verified results. Finally, we

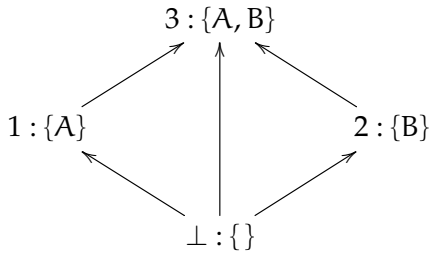

Figure 5.1: The classification/clearance lattice.

attach a hardware model, allowing us to demonstrate that we do in fact eliminate leakage through the cache.

All lemmas in this chapter have been formally verified in Isabelle.

5.1 Security Policies and Covert Channels

Consider a hierarchically partitioned system, as depicted in Figure 5.1. Here, all data is classified with one (or both) of the labels A and B. An agent (program) may be cleared to process one, both, or neither of these, giving rise to 4 *clearance domains*: 1 for A only, 2 for B only, 3 for both and $\bot$ for neither. Our goal is to ensure that information derived from labelled data can only flow into a domain cleared to process it. Enforcing such access policies, encompassing explicit channels, is a well-studied problem [Denning, 1976]. We are interested in eliminating flows through covert and side channels, specifically the cache.

We again consider the cache contention channel introduced in Section 3.2. Recapping briefly, if two processes in distinct clearance domains (say 2 and 3) are executed on the same processor, they may be able to use cache contention to communicate in violation of the security policy, even if all explicit channels are removed. Exactly the same mechanism that we investigated previously would apply: the higher-clearance process (domain 3) can deliberately vary its working set, to evict some fraction of the lines belonging to the low-clearance process (domain 2). As the low process can detect cache misses (as already demonstrated), this forms a channel.

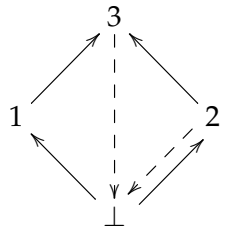

Figure 5.2: The scheduling graph: S.

5.2 Countermeasures through Refinement

In this instance, we consider a different countermeasure to those already described (colouring and instruction-based scheduling)—we flush the entire cache on a context switch. This will eliminate any interference, but it will not be cheap: The Intel Xeon E7-8870, a high-end modern processor, has a 30MiB L3 cache, taking 2.5×10^6 cycles to refill (at the peak theoretical bandwidth of the memory subsystem), or 89% of the 2.8×10^6 cycles per preemption interval at 1000Hz. Nevertheless, the high degree of isolation offered by flushing might make it worthwhile for a sufficiently sensitive system, especially if we can amortise the cost.

A simple optimisation, due to Hu [1992a], is to exploit the security policy to avoid most cache flushes. In our partitioned system, it is acceptable to permit leakage from a domain to any higher domain—it is only necessary to flush when decreasing clearance level. This is the essence of *lattice scheduling*: Transition upward in the classification lattice for as long as possible, before finally starting again at the bottom, employing countermeasures (cache flushing) to protect the downward transition.

To implement this, we construct the *scheduling graph*, S, Figure 5.2, from the classification graph in Figure 5.1. The scheduling graph gives valid domain transitions for the system, and contains only edges from the classification graph, or transitions to the *downgrader*[1], $\bot$. Downward transitions have dashed arrows. In the implementation, the cache is flushed on entering the downgrader. We omit the edges from $\bot$ to 3 and from 1 to $\bot$ to emphasise that not all edges need be included.

[1]The downgrader could just as easily be $\top$ in the lattice: it is trusted both to read from a high domain, and to write to a low domain, as it does nothing but flush the cache.

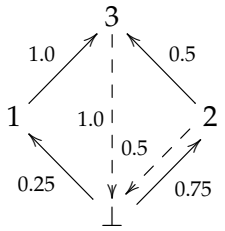

Figure 5.3: The transition graph: T.

The conditions on the scheduling graph (modelled as a relation) are captured as assumptions on S (encapsulated within an Isabelle locale), with the most important being downgrading:

Lemma 5 (Downgrading)**:** If S allows a downward transition, it is to the downgrader, $\bot$:

$$\frac{(c, n) \in S \quad \text{clearance } c \not\subseteq \text{clearance } n}{n = \bot}$$

We specify the scheduler nondeterministically over the valid transitions from the current domain, using the unconstrained demonic choice operator, $:\in$.

record stateA = current_domain :: *dom_id*
scheduleS =
  c is current_domain in
  current_domain $:\in$ (λ_. {n. (c, n) $\in$ S})

The statement 'x is y in (z x)' binds the name x to the current value of the expression y, before evaluating z, while 'y $:\in$ S' updates the variable y nondeterministically, with some value from the set S.

A Randomised Scheduler

This classically nondeterministic specification, together with the downgrading property, captures the requirement that all downward transitions pass through the downgrader. As a practical specification however, it has a problem: it allows starvation. A refinement of this specification is free to follow any trace within the graph, for example $(\bot, 2, \bot, 2, \dots)$, never scheduling domain 3.

We could extend the specification to guarantee starvation freeness, by dictating its behaviour over traces in a modal logic. This would risk obscuring its present simplicity, and would require a more complex implementation.

Randomisation provides an elegant alternative: By assigning a probability to each edge in Figure 5.2, we produce the *transition graph*, T, in Figure 5.3. The outgoing probabilities from each node sum to 1, and any transition with non-zero probability must appear as an edge in Figure 5.2. Implementation is simple: We simply choose the next state randomly, according to the transition probabilities. More importantly, with appropriate transition probabilities the *probability* of starvation is zero. We specify the new scheduler using the probabilistic choice operator:

$$
\begin{array}{l}
\mathsf{scheduleT} = \\
\quad \mathsf{c}\ \mathsf{is\ current\_domain\ in} \\
\quad \mathsf{current\_domain} :\in (\lambda\_.\ \{\bot, 1, 2, 3\}\ \mathsf{at}\ (\lambda\_\ \mathfrak{n}.\ \mathsf{T}\ (\mathsf{c}, \mathfrak{n}))
\end{array}
$$

The statement '$\mathfrak{y} :\in S$ at P' updates the variable $\mathfrak{y}$ *probabilistically*, with some value $x \in S$, with probability $\mathsf{P}\ x$.

This scheduler is a Markov process, with T giving its transition rule. Under the appropriate conditions (strong-connectedness, or positive recurrence interval for all states), there exists an asymptotic equilibrium distribution. These conditions are satisfied by the graph of Figure 5.3, and thus in addition to avoiding starvation, it guarantees statistical fairness.

Program Refinement and Starvation Freedom

In order to show non-leakage, we need to demonstrate that the downgrading property is also shared by scheduleT. We do so by establishing that scheduleT is a *probabilistic refinement* of scheduleS.

Recall the definition of refinement in Equation 4.1: program $\mathfrak{b}$ *refines* program $\mathfrak{a}$, written $\mathfrak{a} \sqsubseteq \mathfrak{b}$, exactly when all expectation-entailments on $\mathfrak{a}$ also hold on $\mathfrak{b}$:

$$\frac{\mathsf{P} \vDash \mathsf{wp}\ \mathfrak{a}\ \mathsf{Q}}{\mathsf{P} \vDash \mathsf{wp}\ \mathfrak{b}\ \mathsf{Q}}$$

Lemma 6: The transition scheduler refines the lattice scheduler:

$$\text{scheduleS} \sqsubseteq \text{scheduleT}$$

Proof. See theorem `refines_ST` in `'lattice_sched/LatticeSched.thy'`, in the attached material. □

Note that, in the terminology of pGCL, the specification of scheduleT is completely "deterministic". Here we are referring to the absence of demonic nondeterminism. This terminology makes sense in light of the refinement order: Demonic nondeterminism can be restricted by refinement, whereas probabilistic choice cannot. Once a specification is fully probabilistic, it is maximal in the refinement lattice, and one can take it no further. This implies that any further refinement is, in fact, semantic equivalence. We make use of this fact shortly, as a shortcut to establishing program correspondence.

Having fixed transition probabilities, we formally establish non-starvation. Proceeding in stages, we first show that starting in *any* domain, the probability of ending in domain $\bot$ after 4 steps is at least $1/64$:

$$(\text{in\_dom } d_i) \,\&\&\, \left(\lambda\_.\ \frac{1}{64}\right) \vDash \text{wp } \text{scheduleT}^4\ (\text{in\_dom } \bot)$$

where

$$\text{in\_dom } d = «\lambda s.\ \text{current\_domain } s = d»$$

We further establish that from domain $\bot$, after a further 4 steps, there is a non-zero probability of ending in any desired final domain:

$$(\text{in\_dom } \bot) \,\&\&\, \left(\lambda\_.\ \frac{1}{64}\right) \vDash \text{wp } \text{scheduleT}^4\ (\text{in\_dom } d_f)$$

Combining these, we have:

$$\left(\lambda\_.\ \frac{1}{4096}\right) \vDash \text{wp } \text{scheduleT}^8\ (\text{in\_dom } d_f) \tag{5.1}$$

Finally:

Lemma 7 (Non-starvation)**:** Taking at least 8 steps from *any* initial domain, we reach *any* final domain with non-zero probability:

$$\forall s.\ 0 < \text{wp } \text{scheduleT}^{\,8+n}\ (\text{in\_dom } d_f)\ s$$

Figure 5.4: First refinement diagram.

Proof. By induction on n. Equation 5.1 establishes the result for $n = 0$. By inspection of Figure 5.3, we see that every domain is reachable in one step, and with non-zero probability, from at least one other, and thus if all domains are reachable after n steps then all are reachable after $n + 1$.

See also lemma `nostarvation` in `'lattice_sched/Fairness.thy'`. □

Note that this result is strictly stronger than the standard form of nonstarvation, "Domain d_f is eventually scheduled.": We have established that not only is d_f *eventually* scheduled, it must always be scheduled within 8 steps, with some nonzero probability.

Figure 5.4 summarises these results. We have downgrading for scheduleS and non-starvation for the probabilistic scheduleT, as indicated by the dotted arrows. Refinement is depicted as a solid arrow. The arrow directions summarise the compositionality of results: composing with refinement, downgrading also holds for scheduleT, but non-starvation does not hold for scheduleS.

Data Refinement and the Lottery Scheduler

It is not sufficient to have an elegant specification, unless that specification can be practically implemented. Therefore we implement our randomised lattice scheduler as a *lottery scheduler* [Waldspurger and Weihl, 1994]. We then only need the assumption of randomness for a single operation: drawing a ticket.

We extend the abstract state with a *lottery* for each domain. Every possible successor domain holds a certain set of tickets, given by the function 'lottery'. To transition, the scheduler draws a ticket (a 32 bit word) and consults the table to choose a successor. To emphasise that the probabilistic component can be isolated, and to demonstrate compatibility with our existing framework, we divide the implementation into a core, in the nondeterministic state monad [Cock et al., 2008], which is then lifted into pGCL using the Exec operator,

allowing us to employ probabilistic choice. Both scheduleC and scheduleM operate on the same state space: stateC. The syntax $r(\!|x := y|\!)$ is an Isabelle record update, assigning value y to field x of record r.

$$\begin{aligned}
\textbf{record}\ \text{domain} &= \text{lottery} :: 32\ \text{word} \Rightarrow \textit{dom_id} \\
\textbf{record}\ \text{stateC} &= \text{current_domain} :: \textit{dom_id} \\
&\quad\ \text{domains} :: \textit{dom_id} \Rightarrow \text{domain} \\
\text{scheduleM}\ t &= \text{do}\ c \leftarrow \text{gets current_domain} \\
&\qquad\ dl \leftarrow \text{gets domains} \\
&\qquad\ \text{let}\ n = \text{lottery}\ (dl\ c)\ t\ \text{in} \\
&\qquad\ \text{modify}\ (\lambda s.\ s(\!|\text{current_domain} := n|\!)) \\
&\quad\ \text{od} \\
\text{scheduleC} &= t\ \text{from}\ (\lambda s.\ \text{UNIV})\ \text{at}\ 2^{-32}\ \text{in} \\
&\qquad\ \text{Exec}\ (\text{scheduleM}\ t)
\end{aligned}$$

The statement 'x from S at P in $(z\ x)$' binds the name x probabilistically from the set S (at probability $P\ x$), before evaluating z.

Having moved to a new state space, we cannot have direct program refinement between scheduleT and scheduleC. Noting, however, that the abstract state can be recovered from the concrete by projection, we instead have *(projective) probabilistic data refinement*:

Definition 4 (Probabilistic Data Refinement)**:** Program b, on state type σ, refines program a, state τ, given precondition $G : \sigma \to \text{Bool}$ and under projection $\theta : \sigma \to \tau$, written $a \sqsubseteq_{G,\theta} b$, exactly when any expectation entailment on a implies the same for b, on the projected state and with a guarded pre-expectation:

$$\frac{P \vDash \text{wp}\ a\ Q}{\text{«}G\text{»}\ \&\&\ (P \circ \theta) \vDash \text{wp}\ b\ (Q \circ \theta)}$$

Lemma 8: Let $\|S\|$ be the cardinal measure (element count) of set S. Under condition LR, that the lottery reflects the transition matrix,

$$T\ (c, n) = 2^{-32} \|\{t.\ \text{lottery}\ (\text{domains}\ s\ c)\ t = n\}\|$$

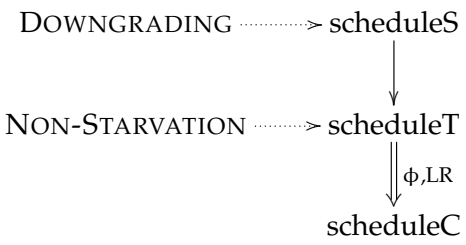

Figure 5.5: Second refinement diagram.

then under projection ϕ, which preserves the current domain,

$$\text{current\_domain}\ (\phi\ s) = \text{current\_domain}\ s$$

scheduleC is a data refinement of scheduleT:

$$\text{scheduleT} \sqsubseteq_{\text{LR},\phi} \text{scheduleC}$$

Probabilistic Correspondence

As already mentioned, scheduleT is maximal in the refinement order, and thus any refinement is an equivalence. This is *probabilistic correspondence*:

Definition 5 (Probabilistic Correspondence)**:** Programs a and b are said to be in *probabilistic correspondence*, pcorres θ G a b, given condition G and under projection θ if, for any post-expectation Q, the guarded pre-expectations coincide:

$$\text{«G»}\ \&\&\ (\text{wp}\ a\ Q \circ \theta) = \text{«G»}\ \&\&\ \text{wp}\ b\ (Q \circ \theta)$$

Probabilistic correspondence is guarded equality on distributions: From an initial state satisfying G, a and b establish Q with equal probability. The advantage of detouring via refinement, rather than directly showing correspondence, is that the proof is simpler: the next result follows directly from Lemma 8:

Lemma 9: The specifications scheduleT and scheduleC correspond given condition LR and under projection ϕ:

$$\text{pcorres}\ \phi\ \text{LR scheduleT scheduleC}$$

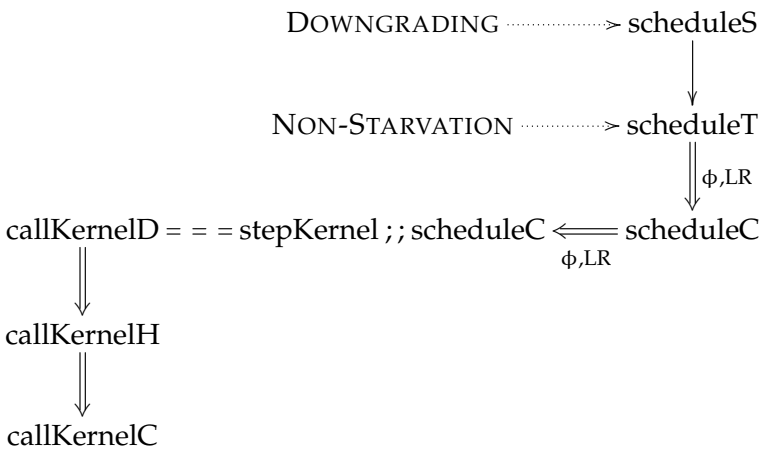

Figure 5.6: Composed refinement diagram.

This extends Figure 5.4 to Figure 5.5, with correspondence indicated by the double arrow. As correspondence implies refinement, both downgrading and non-starvation hold for scheduleC, as implied by the arrows. Properties represented by a single dotted arrow (e.g. downgrading), are preserved by both refinement (single arrow) and correspondence (double arrow).

Proof Reuse: Composing with seL4

Our argument for the feasibility of this approach rests on the compatibility of probabilistic correspondence with the non-probabilistic equivalent at the heart of the L4.verified proof. In Chapter 4, we demonstrated that monadic specifications, in the style of seL4, can be re-used in a probabilistic setting, automatically lifting Hoare triples to probabilistic predicate entailment relations. With the following result we go further, and lift the bulk of the refinement stack. The predicate corres_underlying in the following lemma is the fundamental definition which underlies the refinement results at all levels of the L4.verified proof. Here, we need only note that this is the form of the top-level theorem[2].

[2]Briefly, corres_underlying *srel nf rrel* G G′ m m′ is defined as:

$$\forall(s, s') \in \mathit{srel}.\ G\ s \wedge G'\ s' \rightarrow (\forall(r', t') \in \mathrm{fst}\ (m'\ s').\ \exists(r, t) \in \mathrm{fst}\ (m\ s).\ (t, t') \in \mathit{srel} \wedge \mathrm{rrel}\ r\ r' \wedge (\mathit{nf} \rightarrow \neg \mathrm{snd}\ (m'\ s')))$$

Lemma 10 (Lifting Correspondence)**:** Given correspondence between monadic programs M and M′, with precondition G and projective state relation ϕ,

$$\text{corres\_underlying}\ \{(s, s').\ s = \phi\ s'\}\ \text{True}\ \mathrm{rrel}\ G\ (G \circ \phi)\ M\ M'$$

where M does not fail given G,

$$\text{no\_fail}\ G\ M$$

and neither diverges without failing,

$$\text{empty\_fail}\ M \quad \text{empty\_fail}\ M'$$

and that M is deterministic on the image of the projection,

$$\forall s.\ \exists (r, s').\ M\ (\phi\ s) = \{(\text{False}, (r, s'))\}$$

then we have *probabilistic* correspondence between their lifted counterparts:

$$\text{pcorres}\ \phi\ (G \circ \phi)\ (\text{Exec}\ M)\ (\text{Exec}\ M')$$

Note that the final assumption is exactly the determinism[3] condition that we previously established for scheduleT, restricted to the components of interest. M is free to behave nondeterministically on components which are masked by the projection.

We can thus compose our probabilistic results with the deterministic levels of the L4.verified proof (the executable, or more recent deterministic abstract specification [Matichuk and Murray, 2012]). For the problem at hand, it is only necessary to make a few assumptions on the kernel:

Where guards G and G′ hold on initial states s and s′ satisfying state relation *srel*, for any pair of (result, final state) obtained by executing m', there exists a corresponding pair obtainable by executing m. If the non-failure flag, *nf* is set, then the predicate additionally asserts that m' does not fail.

We use the predicate with a projective relation derived from ϕ, no failure, an arbitrary result relation, and a concrete guard which is the anti-projection of the abstract guard $(G \circ \phi)$.

[3] Determinism gives us correspondence, rather than just refinement. Consider monads A and A′, and variable $x \in \mathbb{N}$, preserved by projection ϕ_A. Let A s be nondeterministic, giving either $s(\!|x := x\ s + 1|\!)$ or $s(\!|x := x\ s + 2|\!)$, while A′ s is deterministic, giving $s(\!|x := x\ s + 2|\!)$. All behaviours of A′ are included in A, and thus corres_underlying holds. However, wp A $x = \lambda s.\ x\ s + 1$ whereas wp A′ $(x \circ \phi_A) = \lambda s.\ x\ s + 2$: a refinement, but not correspondence. As previously mentioned, if A were deterministic then by maximality, this refinement would be correspondence.

Lemma 11: If the kernel preserves the lottery relation,

$$\{\mathrm{LR}\}\ \mathrm{stepKernel}\ \{\lambda\_.\ \mathrm{LR}\}$$

and the current domain,

$$\{\lambda s.\ \mathrm{CD}\ s = d\}\ \mathrm{stepKernel}\ \{\lambda\_\ s.\ \mathrm{CD}\ s = d\}$$

and is total,

$$\mathrm{no\_fail}\ \top\ \mathrm{stepKernel} \quad \mathrm{empty\_fail}\ \mathrm{stepKernel}$$

then with the concrete scheduler, it refines the transition scheduler:

$$\mathrm{scheduleT} \sqsubseteq_{\mathrm{LR},\phi} \mathrm{stepKernel}\ ;;\ \mathrm{scheduleC}$$

With this (again using refinement to show correspondence), Figure 5.5 becomes Figure 5.6, now including the lifted kernel. The L4.verified refinement stack is depicted on the left to indicate how the results would compose, to take our result down to the real, executable kernel. Here callKernelD is the deterministic refinement of original abstract specification of seL4, callKernelH is the executable model derived from the Haskell prototype, and callKernelC is the concrete model, comprising the final C and assembly language implementation

So far, we have only shown that our results are *compatible*: we do not yet have a mechanised proof. The remaining results are the first two assumptions of Lemma 11, which will hold by construction as the existing kernel clearly cannot modify the additional scheduler state, and the fact that the state relation is projective: that is, that the abstract state is uniquely recoverable from the concrete state. This is the intended behaviour of the state relation, and we have no reason to suspect that this will not hold.

Non-leakage with a Concrete Machine Model

Our ultimate goal is to show the absence of information leakage via shared state (specifically the processor cache), and so we extend our scheduler with a simple hardware model. We model a private state per domain (memory), and a single shared state (cache):

$$\begin{aligned}\textbf{record}\ (\mathtt{sh}, \mathtt{pr})\ \mathrm{machine} = {} & \mathrm{private} :: \mathit{dom\_id} \Rightarrow \mathtt{pr} \\ & \mathrm{shared} :: \mathtt{sh}\end{aligned}$$

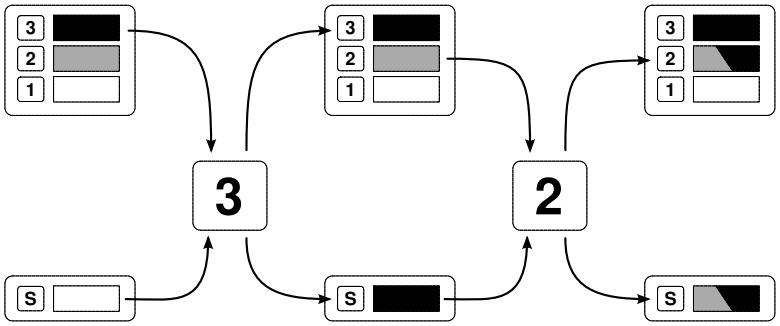

Figure 5.7: A schematic depiction of flow from between domains, via shared state S.

The action of a domain is modelled by the underspecified function runDom :: $\mathtt{sh} \times \mathtt{pr} \Rightarrow \mathtt{sh} \times \mathtt{pr}$, acting on both the current domain's private state and the shared state. Only the action of domain $\bot$ is specified, and then only on the shared state, resetting it.

The model exposes the essential information-flow characteristics of the cache channel, as illustrated by Figure 5.7. Initially, the states associated with domain 3 (black) and 2 (grey) are isolated. After a single step, domain 3's influence propagates to the cache (S), but as yet no other private state has been affected. It is only after the second step that influence propagates to 2's private state, it and the cache now being influenced by both 2 and 3's initial states. As this mixing of private states cannot occur in less than 2 steps, and may take an unbounded time (2's state cannot be influenced until 2 is scheduled), we cannot formulate a one-step security property. Instead we have a trace property, enforcing that after any number of steps, the distribution of outcomes visible to a low observer is independent of any initial high state.

Lemma 12 (Non-leakage)**:** Define functions mask $\mathtt{d}$, which sets the private state of all domains with clearance not less than that of domain $\mathtt{d}$ to some constant value, and replace $\mathtt{d}\ \mathtt{s}$, which overwrites the private state of $\mathtt{d}$ with the supplied value, $\mathtt{s}$.

If the clearance of domain $\mathtt{h}$ is not entirely contained within that of $\mathtt{l}$,

$$\text{clearance}\ \mathtt{h} \not\subseteq \text{clearance}\ \mathtt{l}$$

then any function of the state after execution, which depends only on elements within $\mathtt{l}$'s clearance,

$$\mathrm{Q} \circ \text{mask}\ \mathtt{l}$$

is invariant under modifications to h's private state (as represented by replace):

$$\begin{aligned}\text{wp}\ (\text{runDom} ; ; \text{scheduleT})^n\ (Q \circ \text{mask}\ l) = \\ (\text{wp}\ (\text{runDom} ; ; \text{scheduleT})^n\ (Q \circ \text{mask}\ l)) \circ (\text{replace}\ h\ p)\end{aligned}$$

This result implies that the distribution of any variable o, visible to l, is unaffected by changing any variable s, in the private state of h. In the terminology of Chapter 3, the channel matrix constructed by plotting the conditional distribution $P(o|s)$ is horizontally uniform. Thus, the Shannon leakage and the min-leakage are both zero.

This is a form of *probabilistic noninterference*. Traditional noninterference [Goguen and Meseguer, 1982], is a security property that states (informally) that the state visible at a low security level is not affected by actions taken at a high level (they can replaced by do-nothing operations without changing the output). This notion of security has been extended to probabilistic systems, for example by Gray [1990]. What we have shown is very similar to Gray's definition of P-restrictiveness, modulo the fact that we are considering only states of the system, and not transitions, and therefore only condition 2 of Gray's definition (Theorem 2) is applicable (presented here using Gray's original notation):

$$\sigma_1 \approx \sigma_2 \Longrightarrow P(\sigma_1, x, \sigma_1') = P(\sigma_2, x, \sigma_1')$$

This says that any two states (σ_1,σ_2), which are indistinguishable to a low observer, transition (x) to any other visible state σ_1' with the same probability. If we define the equivalence relation $\approx$ such that it is respected by 'replace' (i.e. modifications to any state above l's clearance are indistinguishable to l), and ignore the transition label, x (our transitions are not visible), then we have an equivalent statement.

We also have correspondence between scheduleT and runDom; ;scheduleC:

Lemma 13: Assuming that the lottery relation LR holds, then under projection ψ, which drops the machine state, we have the correspondence:

$$\text{pcorres LR}\ \psi\ \text{scheduleT}\ (\text{runDom} ; ; \text{scheduleT})$$

and thus by compositionality,

$$\text{pcorres LR}\ (\psi \circ \phi)\ \text{scheduleT}\ (\text{runDom} ; ; \text{scheduleC})$$

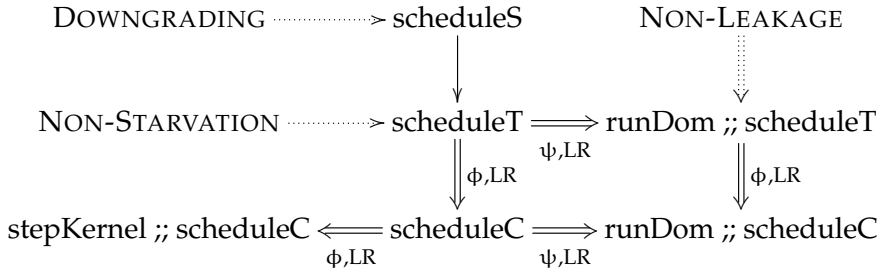

Figure 5.8: The complete refinement diagram.

Therefore, finally, we have all three results: downgrading, non-starvation and non-leakage, on the concrete lottery scheduler composed with the hardware model, as depicted in Figure 5.8. Here, non-leakage is shown using a double dotted arrow to emphasise that it is only preserved by correspondence, and not by refinement.

5.3 Ongoing & Future Work

We have established non-starvation in Lemma 7 as a property of finite traces (of length at least 8). While weaker than this, it would be nice to derive the standard formulation of non-starvation: that any given domain will eventually be scheduled, or $\forall \mathrm{d}.\ \Diamond(\text{current\_domain} = \mathrm{d})$ in the syntax of a boolean modal logic. In our case, of course, the result must necessarily be probabilistic: that it is 'almost certain' that any domain is eventually scheduled. We have already partially mechanised the quantitative temporal logic qTL, of Morgan and McIver [1999], which allows us to express this result as $\forall \mathrm{d}.\ \Diamond(\text{current\_domain} = \mathrm{d})1$, with boolean predicates generalised to real-valued expectations, as for pGCL. We have so far managed to feed our unmodified pGCL results into qTL, and anticipate that these results will appear in a future work.

We have not evaluated the performance impact of lattice scheduling in a real system, which would be a prerequisite for applying this technique in practice. The costs should be dominated by the effect of cache flushing, which is reduced relative to a full flush on every context switch. It would be

interesting to see whether this technique could be used to reduce the cost of the inter-partition L1 cache flush introduced to support cache colouring, as described in Section 3.3.

Progress on the assumptions of Lemma 11, required for the connection to seL4, is ongoing. In a separate work, Daum et al. have shown that the seL4 state relation is indeed projective, satisfying our implicit assumption. Formally proving the explicit assumptions (lottery relation and current domain preservation) presents no theoretical challenges, simply requiring a large but trivial proof. Integrating the projectivity result should be similarly straightforward. The more interesting question is what form that the final top-level statement should take to cleanly integrate the probabilistic and classical properties of seL4, and is the subject of ongoing research.

5.4 Related Work

Lattice-based security models are a longstanding idea, motivated largely by institutional classification policies [DoD], and formally treated by authors including Denning [1976]. Lattice scheduling, which exploits this structure to minimise context flushing was presented by Hu [1992a], and was implemented (as with fuzzy time) in the VAX VMM [Karger et al., 1991].

Lottery scheduling is an established technique for efficient hierarchical allocation of execution time, introduced by Waldspurger and Weihl [1994]. We use it as an elegant way to implement probabilistic scheduling: we do not take advantage of hierarchical resource distribution.

Barthe et al. [2012a] present a machine-checked proof (in the Coq theorem prover) that cache flushing on a context switch eliminates information leakage via the cache, with respect to a particular cache model. Our result goes further in two ways: by showing that the number of flushes can be minimised (by lattice scheduling), and that we can guarantee liveness (by lottery scheduling). Our hardware model is, however, more simplistic. We also demonstrate that we can integrate the results of a large existing verification effort [Klein et al., 2009].

Many authors have analysed the leakage properties of scheduling algorithms themselves [Chen and Malacaria, 2007; Gong et al., 2011; Huisman and Ngo, 2012], some employing mechanical proof. Most existing analyses focus on leakage due to the actions of the scheduler itself, or due to the order of

updates to shared variables. We are specifically concerned with mitigating a side channel, outside any explicitly shared state. The absence of unintended channels through explicit mechanisms in seL4, including the scheduler, has already been established [Murray et al., 2012, 2013].

Large-scale probabilistic verification efforts are still rare, although Fidge and Shankland [2003] have previously applied pGCL to verifying termination properties for the IEEE 1394 (Firewire) election protocol. Other authors, such as Baier and Kwiatkowska [1997] approach the problem of asymptotic fairness by model-checking formulae in probabilistic temporal logics. By integrating qTL, we hope to achieve a similar level of automation.

5.5 Summary

We present a hybrid probabilistic lattice-lottery scheduler, which allows efficient mitigation of the cache channel while simultaneously guaranteeing non-starvation. Working in pGCL, our system is produced by iterative refinement, supplemented by mechanical proof.

This demonstrates that given adequate tool support (namely Isabelle/HOL and our mechanisation of pGCL), refinement-driven development and verification of realistic *probabilistic* systems software is no more difficult than the existing non-probabilistic case. We have shown that our refinement framework is compatible with that of the L4.verified project, and set out the steps necessary to combine this work with a system such as seL4, giving a mechanical proof down to a real system of probabilistic top-level properties. Above all, we argue that verifying probabilistic security properties on realistic systems software is entirely feasible with current technology.

6 Formal Leakage Models

This chapter presents joint work with my former supervisor Will Uther[1]. The formal proof of Equation 6.4 is the subject of the following paper:
David Cock. From probabilistic operational semantics to information theory; side channels in pGCL with isabelle. In *Proceedings of the 5th International Conference on Interactive Theorem Proving*, pages 1–15, Vienna, Austria, July 2014. Springer. doi:10.1007/978-3-319-08970-6_12

In this final chapter, we demonstrate that given a high-level probabilistic model of a system in pGCL, we can formally abstract from operational details, and reason purely in information-theoretic or probabilistic terms. This provides a link back to Chapter 2, where we calculated such bounds based on an intuitive understanding of the system. The purpose of this chapter is to formalise that intuition.

Specifically, we formally derive leakage bounds for the guessing attack introduced in Section 2.2. We model the attack as a loop in pGCL with one iteration per guess, terminating once the secret is found. The principal result of this chapter, the connection of the concrete attack model to the general vulnerability bound of Equation 6.4, is fully machine-checked in Isabelle/HOL.

We continue to take a black-box approach to leakage—We do not assume the ability to modify (or even inspect) the source code of sensitive components. We want to provide system-level security guarantees given only a model of a component's behaviour. In contrast to the analytic approach of Chapter 2,

[1]Now at Google Australia

and the empirical approach of Chapter 3, in this chapter we achieve this with a refinement-driven approach to specification: results are shown on simple models, which components are then shown to refine.

We return again to the example password authentication service of Chapter 2, as our canonical example of a system vulnerable to a guessing attack. Here the secret is the stored password (or hash), and the correct behaviour is to permit access exactly when the supplied password matches the stored one. As established, this specification permits *intrinsic* leakage: Every response either confirms or eliminates a hypothesis for the adversary. This leakage is unavoidable: the system cannot do its job unless it is able to deny incorrect passwords. Again, as established in Section 2.2, an implementation might furthermore have a response time which depends on the similarity of the supplied and stored passwords. Here we abstract from the exact nature of this leakage, and consider it only as a conditional distribution of observation given secret.

Adaptive Countermeasures We aim to establish measures of leakage to guide the application and evaluation of *adaptive countermeasures*. By this, we mean a system with a variable degree of protection against side-channel leakage, guided by some adaptive strategy. We assume that our countermeasure can completely eliminate leakage over some specific channel, albeit at some cost. Our model for this type of countermeasure is the scheduled message delivery system presented in Section 3.7.

In the password example, response time is correlated with the secret (the stored password). As a countermeasure, we might quantise the response time by delaying any response until the next time satisfying $t = k\Delta t$, for integer k. If the response time is bounded above by t_{MAX}, this limits the space of possible observations per response to $n = \lceil \frac{t_{MAX}}{\Delta t} \rceil$. If $\Delta t \geqslant t_{MAX}$ then $n = 1$ and the adversary learns nothing through the channel. This protection comes at a cost: The system's latency must increase, as a system's worst-case execution time is often orders of magnitude larger than its average.

Given this tradeoff, we have a choice as to how aggressively the countermeasure is applied. It is important to note that for a secret selected from a finite set, the chance of compromise is never zero; the adversary always has a chance of simply guessing the secret, however small. We consider secrets to be drawn from such a finite (but large) set.

If a system has some upper bound on its current vulnerability, then it can ensure security by switching to a completely safe countermeasure ($\Delta t = t_{MAX}$ in the above example) as soon as its vulnerability approaches an acceptable threshold. Ideally, it should be able to apply the countermeasure at partial strength to achieve any desired level of security. The purpose of these measures is to guide the system in negotiating this tradeoff.

As a vulnerability measure, we take the k-guess vulnerability of Definition 2, and show that we can formally derive it from a concrete attack model. We also formally establish the optimality of the Bayesian attacker introduced in Section 2.4.

6.1 An Informal Model

As described in Chapter 2, we distinguish between intrinsic leakage, which occurs unavoidably through the correct behaviour of the system, and side-channel leakage, which comprises any information that is not required to be leaked, but is leaked by a specific implementation.

To understand the effect of the side-channel leakage, we compare the system under consideration to an ideal implementation, and consider the likelihood of compromise of each. This approach is practically relevant under our assumptions for adaptive countermeasures, as any system can be transformed into an ideal system by fully applying them.

As we have established (for example, in Chapter 3), side-channel leakage is generally stochastic. At any point, we assume that the adversary has accumulated some list of side-channel observations, $o_1, \ldots, o_t \in O$, correlated with the secret, $s \in S$. The prior distribution on secrets, $P(s)$, and the conditional distribution, $P(o|s)$, of observations given the secret are assumed to be known to the attacker. We further assume that the observations are conditionally independent, given the secret. We assume that these distributions are irreducible, that there exist no observable variables permitting factorisation of either. To the adversary then, the secret is distributed according to the conditional probability given by Bayes' rule (see Equation 2.8 and the associated discussion), which we restate here in expanded form:

$$P(s|o_1, \ldots, o_t) = \frac{P(o_1, \ldots, o_t|s)P(s)}{\sum_s P(o_1, \ldots, o_t|s)P(s)}$$

We write $\Delta(S)$ for the set of distributions on S. We write $P(s = s')$, or simply $P(s')$, for the probability that variable s has value s'. Where probabilities (or likelihoods) appear as functions, we disambiguate with subscripts: therefore $P(s) = P_s\ s$ and $P(o|s) = P_{o|s}\ s\ o$.

In our authentication model, the adversary's interaction with the system is limited to guessing possible secrets one at a time, eliciting a yes-or-no answer. The attacker's prior knowledge regarding the key is summarised by the distribution $P(s)$. This may represent the combination of initial knowledge (e.g. a known bias in key selection, such as the Markov model used in Section 2.4) and previous side-channel observations. The attacker's model for the side channel's dependence on the key is given by $P(o|s)$. We assume that observations are conditionally independent, that there is no relevant hidden state beyond the secret, and that observations do not depend on the attacker's input (in this respect we have simplified from the `strcmp` example). The attacker's observations are given by the list $\mathtt{ol}$. Initially $\mathtt{ol} = [\,]$. The attacker's only relevant knowledge is the two distributions (which are constant) and $\mathtt{ol}$, and thus its strategy reduces to a function from an observation list to a secret to guess:

$$\sigma :: O^* \to S$$

We treat the attack as a game, played between the attacker and the system, with the environment an impartial participant. The players are distinguished by the treatment of their choices: The system's choices are predetermined, the environment's are random and the attacker's are demonically nondeterministic (always assumed to be taken to minimise our chance of success, see Section 4.1). The distinction between random and demonic choice is crucial: The attacker is assumed choose the outcome that maximises its chances, whereas the environment's choices are made without consideration of their effect on the relative positions of the players, being only constrained by their distribution.

We assume no computational limits on the attacker. For ourselves, however, while tracking the true security state by iteratively applying Bayes rule to the distribution $P(s|o_1, \dots)$ gives the most precise vulnerability measure, doing so is intractable. Instead, we look for a summary measure for the distribution, from which we can calculate a safe upper bound. For example, we might track the Shannon entropy remaining in $P(s)$, $H(s|o_1, \dots)$, hoping to keep it above some threshold. Likewise, we might summarise $P(o|s)$ by the

conditional entropy, $H(o|s)$ (in some instances the min-entropy $H_\infty(s)$ may be a better measure, however). To model this, we allow the functions P_s and $P_{o|s}$ to range over the sets Q_s and $Q_{o|s}$, respectively. We then ask for the worst-case vulnerability (strictly, the supremum), noting that as long as these *summary sets* contain the true distribution, the vulnerability we calculate is a safe upper bound. We consider the selection from the summary sets to be made by the adversary, as it is done to maximise vulnerability. The applicability of both Shannon and min-entropy to guessing attacks was established in Chapter 2.

Finally, we establish our vulnerability measure by selecting a positive integer, n. The system is considered vulnerable if, at any point, the attacker has produced the correct secret in no more than n trials. This is, of course, V_n, per Definition 2.

The game proceeds in stages:

1. The system is instantiated: the adversary selects $P(s)$ and $P(o|s)$ satisfying $P_s \in Q_s$ and $P_{o|s} \in Q_{o|s}$. Knowing the system model, the adversary commits to a strategy σ.

2. The environment randomly selects s according to $P(s)$.

3. The game is played, with the system generating observations, and the attacker guessing at the secret.

That the adversary commits to a strategy (which secret it will guess for which set of observations) in step 1, *before* s is chosen, is crucial, and captures the restriction that the adversary knows only the distribution on s, and not its value. Were the adversary's choice made afterwards, nondeterminism would allow it to select a strategy with $\sigma\,[\,] = s$. That the game must take this form, with demonic choice resolved before random choice, is a consequence of the fact that the two notions of nondeterminism do not commute.

We have informally defined *posterior* vulnerability for a specific system trace, under the following assumptions:

- The adversary's knowledge of the system is given exactly by the distributions $P(s)$ and $P(o|s)$, and the observation list ol.

- Observations are conditionally independent, both of each other and of the attacker's actions, given the secret.

We now formalise this model, and introduce machinery to allows us to abstract systematically over both notions of nondeterminism, to transform this posterior vulnerability to *prior* vulnerability, and thus to bound the likelihood of compromise for the system.

6.2 A Formal Model

We formalise the above game as a process in pGCL, as presented in Chapter 4. We first briefly cover the rules for loop verification, which we have so far avoided, and justify our use of the *liberal* semantics (wlp).

More pGCL

We first need to cover a few more details of pGCL. As before, the interested reader is directed to McIver and Morgan [2004] for a fuller treatment.

The following identities for embedded predicates ($\{0,1\}$-valued expectations) will be useful, and follow from the fact that $«P \wedge Q» = «P» * «Q»$:

$$«P» * «P» = «P» \qquad \text{idempotence}$$
$$«P» * «\neg P» = 0 \qquad \text{cancellation}$$

Under the usual (strict) interpretation, the reverse map (expectation transformer) is wp, or *weakest precondition*. On predicates, this gives the weakest predicate which, if it holds on the initial state, implies that the given postcondition will hold on the final state. For a post-expectation given as the embedding of a boolean postcondition, this is the least probability (over all demonic choices) that the postcondition will hold in the final state, for a given initial state.

So far this suits us, but a strict interpretation has a problem. Recursion is defined using the *least* fixed point. If we attempt to calculate the probability of success for the defender (that the system remains secure), wp will assign the least expectation, $\lambda s.\ 0$, to a non-terminating program, whereas we should treat non-termination as success (the attacker in this case never produces the correct secret). The attacker is certainly free to choose such a strategy, and under demonic nondeterminism can be assumed to do so.

We might try instead to calculate the probability of success for the attacker (that the system is compromised), which would give the correct answer for a

non-terminating attack, but in this case demonic choice will act in the wrong direction. Demonic choice will minimise this pre-expectation, whereas if we conflate the demon and the attacker, it should maximise it.

The answer to our dilemma is to instead use the weakest *liberal* precondition (wlp). This interpretation models partial correctness (correct if terminating), and differs from wp only in its treatment of aborting or otherwise non-terminating programs, which are considered successful. The definition of recursion here is as the *greatest* rather than least fixed point.

There is no syntactic rule for recursion, and of course, no general decision procedure exists. For our needs, however, the following verification condition suffices:

Lemma 14 (Partial Correctness for Loops)**:** Define a while loop as follows:

$$\textbf{do } G \to \mathit{body} \textbf{ od} \triangleq \mu X.\ (\mathit{body}\ ;; X) \ {}_{«G»}\oplus\ \textbf{skip}$$

If I is a *probabilistic invariant*, such that

$$«G» * I \vDash \mathrm{wlp}\ \mathit{body}\ I$$

then

$$I \vDash \mathrm{wlp}\ \textbf{do } G \to \mathit{body} \textbf{ od}\ («\mathcal{N}G» * I) \tag{6.1}$$

establishes partial correctness for the loop.

Proof. See McIver and Morgan [2004] Lemma 7.2.2. □

Finally, we recall the definition for extended demonic choice:

Definition 6 (Extended Demonic Choice)**:** For countable S, write $x :\in S$ for $x := s_1 \sqcap x := s_2 \sqcap \dots$, with the following syntactic interpretation:

$$\mathrm{wp}\ (x :\in S)\ R = \inf_{s \in S} R\left[{}^{s}/_{x}\right]$$

If R, considered as a function $S \to \mathbb{R}$, is nonnegative and bounded, the existence of this infimum follows from the completeness of the reals.

Noting that this argument does not rely on the countability of S, we extend our definition to uncountable sets using the same interpretation. There is no corresponding syntactic intuition in this case. [2]

[2]This definition fails to guarantee continuity, but as we only apply it outside of recursion, this does not cause difficulty.

Deriving a Bound on Vulnerability

We now formally define vulnerability, using the victory condition in the above game. Let ol be the attacker's current list of observations. We write $|l|$ for the length of list l, $[\,]$ for the empty list, and $l[a..b]$ for the contiguous subsequence of l from index b to a inclusive, counting from the end of the list n.b. for $a < b$, $l[a..b] = [\,]$ and for $|l| \leqslant a$ and $b \leqslant 0$, $l[a..b] = l$.

The attacker wins if its strategy has produced the correct secret, s, after no more than n trials (observations):

$$V = \exists 0 \leqslant i \leqslant n.\ \sigma\ o[i..1] = s$$

The complement gives the security predicate:

$$\neg V = \forall 0 \leqslant i \leqslant n.\ \sigma\ o[i..1] \neq s$$

We embed V as «V»: a $\{0,1\}$-valued function on final states. We interpret this as the *probability* that this system has been compromised by a given trace.

We next consider the final step of the game: the attack loop, in which the attacker collects observations and attempts to guess the secret:

$ol := [\,]$;
do $\sigma.ol \neq s \rightarrow$
  o from UNIV at $P(o|s)$;
  $ol := o{:}ol$

In order to evaluate the weakest pre-expectation of our security predicate under this attack, we first show the following:

Lemma 15: [3] The expectation,

$$I = \prod_{i=0}^{n} «\sigma\ ol[i..1] \neq s» * \sum_{\substack{o_n, \\ \ldots, \\ o_{|ol|+1}}} \prod_{i=|ol|+1}^{n} P(o_i|s) «\sigma\ (o_i{:}\ldots{:}ol) \neq s»$$

is a probabilistic wlp invariant for loop *guess*.

[3] See also lemma `invariant` in `guessing_attack/Guessing_Attack.thy`.

Proof. Unfolding wlp applications, we have:

$$\begin{aligned}\text{wlp } \textit{body}\ I &= \text{wlp } (o \text{ from UNIV at } P(o|s); ol := o{:}ol)\ I \\ &= \sum_o P(o|s) I\left[{}^{o:ol}/_{ol}\right]\end{aligned}$$

If $n \leqslant |ol|$ then

$$I = \prod_{i=0}^{n} \text{«}\sigma\ ol[i..1] \neq s\text{»} \quad \text{and thus}$$

$$\text{wlp } \textit{body}\ I = \sum_o P(o|s) \prod_{i=0}^{n} \text{«}\sigma\ (o{:}ol)[i..1] \neq s\text{»} = I$$

Otherwise, $|ol| < n$ and,

$$\begin{aligned}&\text{wlp } \textit{body}\ I \\ &= \sum_o P(o|s) \left(\prod_{i=0}^{|ol|+1} \text{«}\sigma\ (o{:}ol)[i..1] \neq s\text{»} * \right. \\ &\qquad \left. \sum_{\substack{o_{|ol|+2}, \\ \dots, \\ o_n}} \prod_{i=|ol|+2}^{n} P(o_i|s) \text{«}\sigma\ (o_i{:}\dots{:}o{:}ol) \neq s\text{»} \right) \\ &= \sum_o \left(\prod_{i=0}^{|ol|} \text{«}\sigma\ ol[i..1] \neq s\text{»} * P(o|s) \text{«}\sigma\ (o{:}ol) \neq s\text{»} * \right. \\ &\qquad \left. \sum_{\substack{o_{|ol|+2}, \\ \dots, \\ o_n}} \prod_{i=|ol|+2}^{n} P(o_i|s) \text{«}\sigma\ (o_i{:}\dots{:}o{:}ol) \neq s\text{»} \right) \\ &= \prod_{i=0}^{|ol|} \text{«}\sigma\ ol[i..1] \neq s\text{»} \\ &\quad * \sum_o \sum_{\substack{o_{|ol|+2}, \\ \dots, \\ o_n}} P(o|s) \text{«}\sigma\ (o{:}ol) \neq s\text{»} \prod_{i=|ol|+2}^{n} P(o_i|s) \text{«}\sigma\ (o_i{:}\dots{:}o{:}ol) \neq s\text{»} \\ &= \prod_{i=0}^{|ol|} \text{«}\sigma\ ol[i..1] \neq s\text{»} * \sum_{\substack{o_{|ol|+1}, \\ \dots, \\ o_n}} \prod_{i=|ol|+1}^{n} P(o_i|s) \text{«}\sigma\ (o_i{:}\dots{:}o_{|ol|+1}{:}ol) \neq s\text{»} \\ &= I\end{aligned}$$

also,

$$\begin{aligned}
\text{«G»} * I &= \text{«}\sigma\ ol \neq s\text{»} * \prod_{i=0}^{|ol|} \text{«}\sigma\ ol[i..1] \neq s\text{»} * \dots \\
&= \text{«}\sigma\ ol \neq s\text{»} * \text{«}\sigma\ ol \neq s\text{»} * \prod_{i=0}^{|ol|-1} \dots \\
&= I && \text{by idempotence}
\end{aligned}$$

Thus

$$\text{«G»} * I = \text{wlp}\ \mathit{body}\ I$$

□

Applying Equation 6.1 therefore, we have that:

$$I \vDash \text{wlp} \left(\begin{array}{l} \textbf{do}\ \sigma\ ol \neq s \rightarrow \\ \quad o \text{ from UNIV at } P(o|s); \\ \quad ol := o{:}ol \end{array} \right) \Big(\text{«}\sigma\ ol = s\text{»} * I \Big)$$

Unfolding I in the post-expectation we have:

$$\begin{aligned}
\text{«}\sigma\ ol = s\text{»} * I = \text{«}\sigma\ ol = s\text{»} * \prod_{i=0}^{n} \text{«}\sigma\ ol[i..1] \neq s\text{»} \\
* \sum_{\substack{o_{|ol|+1}, \\ \dots, \\ o_n}} \prod_{i=|ol|+1}^{n} P(o_i|s) \text{«}\sigma\ (o_i{:}\dots{:}ol) \neq s\text{»}
\end{aligned}$$

Either $|ol| \leqslant n$, in which case the left product collapses:

$$\begin{aligned}
\text{«}\sigma\ ol = s\text{»} * I &= \text{«}\sigma\ ol = s\text{»} * \text{«}\sigma\ ol \neq s\text{»} * \dots \\
&= 0 && \text{by cancellation}
\end{aligned}$$

or $n < |ol|$, in which case the right sum is empty and

$$\begin{aligned}
\text{«}\sigma\ ol = s\text{»} * I &= \text{«}\sigma\ ol = s\text{»} * \prod_{i=1}^{n} \text{«}\sigma\ o[i..1] \neq s\text{»} \\
&= \text{«}\sigma\ ol = s\text{»} * \text{«}\forall 1 \leqslant i \leqslant n.\ \sigma\ ol[i..1] \neq s\text{»} \\
&\vDash \text{«}\neg V\text{»}
\end{aligned}$$

By the monotonicity of wlp, and prepending the initialisation step, we have:

$$I\left[[\,]/_{ol}\right] \vDash \text{wlp} \left(\begin{array}{l} ol := [\,]; \\ \textbf{do } \sigma\ ol \neq s \rightarrow \\ \quad o \text{ from UNIV at } P(o|s); \\ \quad ol := o{:}ol \end{array} \right) \langle\!\langle \neg V \rangle\!\rangle$$

Applying the system's probabilistic choice of secret:

$$\sum_s \left(P(s) * I\left[[\,]/_{ol}\right] \right) = \sum_{s \in S} \left(P_s(s) * \sum_{\substack{o_1, \\ \ldots, \\ o_n}} \prod_{i=1}^{n} P(o_i|s) \langle\!\langle \sigma\ o[i..1] \neq s \rangle\!\rangle \right)$$

$$\vDash \text{wlp} \left(\begin{array}{l} s \text{ from UNIV at } P(s); \\ ol := [\,]; \\ \textbf{do } \sigma\ ol \neq s \rightarrow \\ o \text{ from UNIV at } P(o|s); \\ \quad ol := o{:}ol \end{array} \right) \langle\!\langle \neg V \rangle\!\rangle \qquad (6.2)$$

We thus have a lower bound on the likelihood of compromise, from a starting state specified by the distribution $P(s)$, likelihood function $P(o|s)$, and the attacker's strategy, σ. To get to a truly pessimistic bound, we need to abstract over the latter, considering the worst case over all possible strategies. We first rewrite Equation 6.2 as follows:

$$\text{wlp} \left(\begin{array}{l} s \text{ from UNIV at } P(s); \\ ol := [\,]; \\ \textbf{do } \sigma\ ol \neq s \rightarrow \\ \quad o \text{ from UNIV at } P(o|s); \\ \quad ol := o{:}ol \end{array} \right) \langle\!\langle \neg V \rangle\!\rangle$$

$$\dashv \sum_{s \in S} \left(P(s) * \sum_{\substack{o_1, \\ \ldots, \\ o_n}} \prod_{i=0}^{n} P(o_i|s) \langle\!\langle \sigma\ ol[i..1] \neq s \rangle\!\rangle \right)$$

$$= \sum_{s \in S} \left(P(s) * \sum_{\substack{o_1, \\ \ldots, \\ o_n}} P(o_1 \ldots o_n|s) \prod_{i=0}^{n} \langle\!\langle \sigma\ ol[i..1] \neq s \rangle\!\rangle \right) \quad \text{by independence}$$

which, as «P» $*$ «Q» $= P \wedge Q$ and $P(s)P(o|s) = P(o)P(s|o)$,

$$
\begin{aligned}
&= \sum_{\substack{o_1,\\ \ldots,\\ o_n}} \left(P(o_1 \ldots o_n) * \sum_{s \in S : \bigwedge_{i=0}^{n} o_l[\![i..1]\!] \neq s} P(s|o_1 \ldots o_n) \right) \\
&= \sum_{\substack{o_1,\\ \ldots,\\ o_n}} \left(P(o_1 \ldots o_n) * \left(1 - \sum_{s \in S : \bigvee_{i=0}^{n} \sigma\ o_l[\![i..1]\!] = s} P(s|o_1 \ldots o_n) \right) \right) \\
&= 1 - \sum_{\substack{o_1,\\ \ldots,\\ o_n}} \left(P(o_1 \ldots o_n) * \overbrace{\sum_{s \in S : \bigvee_{i=0}^{n} \sigma\ o_l[\![i..1]\!] = s} P(s|o_1 \ldots o_n)}^{X} \right) \qquad (6.3)
\end{aligned}
$$

We note that if $\sigma\ o_l[\![i..1]\!] = \sigma\ o_l[\![j..1]\!]$ for $i \neq j$ (the strategy repeats), and $n \leqslant |S|$, then a non-repeating strategy may be constructed by iteratively replacing all repeated guesses with fresh elements of S. Each replacement increases the number of summands, and since they are non-negative, X least as large as before. Thus, for any repeating strategy, there exists a non-repeating strategy that is at least as good. We therefore restrict our attention to non-repeating strategies.

Given no repeats, the union in Equation 6.3 is disjoint, which thus becomes

$$
\begin{aligned}
&1 - \sum_{\substack{o_1,\\ \ldots,\\ o_n}} \sum_{i=0}^{n} P(\sigma\ o_l[\![i..1]\!], o_l[\![n..1]\!]) \\
&= 1 - \sum_{i=0}^{n} \sum_{\substack{o_1,\\ \ldots,\\ o_i}} P(\sigma\ o_l[\![i..1]\!], o_l[\![i..1]\!]) \quad \text{by marginalisation}
\end{aligned}
$$

Which is minimised when σ selects according to the incremental maximum-a-posteriori (MAP) criterion: that is, maximising $P(\sigma\ o_l[\![i..1]\!]|o_1 \ldots o_i)$. Take one such strategy, labelled $\hat{\sigma}$. Maximising vulnerability over strategies is equivalent to allowing the adversary a demonic choice over strategies, and then calculating the pre-expectation of the combined process. Together with

choices over distribution and likelihood, we have:

$$V_{\text{prior}} = 1 - \text{wlp} \left(\begin{array}{l} P_s \in Q_s; \\ P_{o|s} \in Q_{o|s}; \\ \sigma \in O^* \to S; \\ s \text{ from UNIV at } P(s); \\ ol := [\,]; \\ \textbf{do } \sigma\ ol \neq s \to \\ o \text{ from UNIV at } P(o|s); \\ \quad ol := o{:}ol \end{array} \right) [\neg V]$$

$$\begin{aligned} &= 1 - \inf_{\substack{P_s \in Q_s \\ P_{o|s} \in Q_{o|s}}} 1 - \sum_{i=0}^{n} \sum_{ol[\![i..1]\!]} P(\hat{\sigma}\ ol[\![i..1]\!], ol[\![i..1]\!]) \\ &= \sup_{\substack{P_s \in Q_s \\ P_{o|s} \in Q_{o|s}}} \sum_{i=0}^{n} \sum_{ol[\![i..1]\!]} P(\hat{\sigma}\ ol[\![i..1]\!], ol[\![i..1]\!]) \end{aligned} \tag{6.4}$$

The above result is formally verified (for a fixed P_s and $P_{o|s}$). See theorem `V_prior` of `guessing_attack/Guessing_Attack.thy` in the attached Isabelle sources.

Note that the optimal attack strategy is the one that maximises $P(s, ol)$, or the joint probability of the hypothesis (that s is the real secret) and the observations. This is the MAP (maximum a posteriori) criterion for a Bayesian attacker. This is the strategy taken by the attacker of Section 2.4, which we have now shown to be optimal.

6.3 Simpler Bounds using Refinement

The bound in Equation 6.4 is tight, but impractical to apply, as it is essentially the simulation of an optimal strategy. It serves as the basis, however, for looser bounds that are more easily computed. We discover a family of bounds, by exploring the refinement order on attack models.

Recall the definition of refinement from Section 4.1: The standard refinement order in pGCL is induced by pointwise comparison of weakest pre-expectations. Specifically, a program a is refined by a program b, written $a \sqsubseteq b$, if for all post-expectations E and initial states s, $\text{wlp}\ a\ E\ s \leqslant \text{wlp}\ b\ E\ s$. For *standard* post-expectations, those constructed by embedding a predicate,

e.g. «P», a refinement establishes the postcondition with at least as high a probability as the original program, by definition.

Our post-expectation is standard, being the embedding of the predicate 'the system is secure'; Therefore, refinement can only decrease vulnerability, and abstraction (its converse) can only increase it. The power of this approach is that we can construct such refinements and abstractions structurally, rather than reasoning from first principles.

The structural refinement rules that we exploit are:

$$\textsc{Refine\_Seq}\ \frac{a \sqsubseteq b}{a\,;;\,c \sqsubseteq b\,;;\,c} \qquad \textsc{Refine\_Choice}\ \frac{S \subseteq T}{x :\in T \sqsubseteq x :\in S}$$

We apply REFINE_CHOICE to the parameters Q_s and $Q_{o|s}$, to obtain a parameterised family of attack models, ordered by refinement in the reverse of the order of inclusion of the set parameters. The resulting ordering on vulnerability (lower for smaller sets) matches the intuition that reducing the attacker's freedom can never increase the worst-case vulnerability.

For convenience, we dispense with the subscripted V_{prior}, and simply write V for vulnerability, understanding it to mean prior vulnerability, in the sense of Equation 6.4. We must, however, introduce new syntax to distinguish our vulnerability bounds, according to the specialisations on which they rely. We thus write, for example, $V_m \Big|_{\ s}$ to mean V_{prior}, where $Q_s = S$ and $n = m$, or $V_m \Big|_{\ \mathcal{P}(P_s)}$ where $Q_s = \{P' : \mathcal{P}(P')\}$ i.e. the set of distributions satisfying predicate $\mathcal{P}$.

The attack models (and hence bounds) form a complete lattice under the refinement order. The bottom element is the degenerate class of models (equivalent under refinement), where either Q_s or $Q_{o|s}$ is the empty set: In this case, the relevant supremum in Equation 6.4 is 0. Thus:

$$V_n \Big|_{\ Q_s=\{\} \vee Q_{o|s}=\{\}} = 0$$

The top element is again a class, where Q_s is the universal set. For any $s \in S$, this includes the delta distribution assigning all mass to s, whence the attacker is free to choose a strategy where $\sigma\,[\,] = s$, and thus:

$$V_n \Big|_{\ Q_s=\text{UNIV}} = 1$$

The next simplest case is a non-leaking system, where observations are uncorrelated with secrets. That is, $\forall s\ o.\ P(s,o) = P(s)P(o)$. This would be the

case where no further observations are permitted, for example in the offline brute-force phase of an attack. We have, writing NL for the independence condition:

$$\begin{aligned} V_n \Big|_{P(o,s)=P(o)P(s)} &= \sup_{\substack{P_s \in Q_s \\ P_{o|s} \in NL}} \sum_{\substack{o_1, \\ \dots, \\ o_i}} \sum_{i=0}^{n} P(\hat{s}_i, o_1 \dots o_i) \\ &= \sup_{\substack{P_s \in Q_s \\ P_{o|s} \in NL}} \sum_{i=0}^{n} \sum_{\substack{o_1, \\ \dots, \\ o_i}} P(o_1 \dots o_i) P(\hat{s}_i) \\ &= \sup_{P_s \in Q_s} \sum_{i=0}^{n} P(\hat{s}_i) \end{aligned}$$

Setting $Q_s = \{P'_s\}$, to consider a known prior distribution, we have,

$$V_n \Big|_{\substack{P(o,s)=P(o)P(s) \\ \{P(s)\}}} = \sum_{i=0}^{n} P'(\hat{s}_i) \ . \tag{6.5}$$

The probability of compromise by an optimal attacker is simply the sum of the prior probabilities of the n most likely secrets. The belief distribution does not change given observations, given that they are independent of the secret.

For $n = 0$ we have, even more simply (appealing to the optimality of $\hat{\sigma}$),

$$V_0 \Big|_{\substack{P(o,s)=P(o)P(s) \\ \{P(s)\}}} = \boxed{\max_s P'(s)} \tag{6.6}$$

This is exactly the one-guess vulnerability introduced in Equation 2.2.

If we are more ambitious, we can make a less restrictive assumption regarding the prior distribution by choosing a larger set for Q_s, while retaining the assumption of no leakage. We might ask: What is the weakest assumption (the largest set) that admits the simplification to Equation 6.6? The answer, $Q_s = \{P \in \Delta(S) : \max_s P(s) \leqslant X\}$, for some $X \in [0, 1]$, is practically significant in its connection to the min entropy.

Recall (from Equation 2.11), the definition of min entropy:

$$H_\infty(P) \triangleq -\min_s \log_2 P(s)$$

or equivalently,

$$\max_s P(s) = 2^{-H_\infty(P)}$$

We can thus rewrite our set equivalently as,

$$Q_s = \{P' \in \Delta(S) : H_\infty(P') \leqslant H_\infty(P)\} \tag{6.7}$$

whence Equation 6.6 becomes

$$V_0 \left| \begin{smallmatrix} P'(o,s)=P'(o)P'(s) \\ H_\infty(P')\leqslant H_\infty(P) \end{smallmatrix} \right. = 2^{-H_\infty(P)}$$

and for multiple guesses, appealing again to the optimality of $\hat{\sigma}$, Equation 6.5 becomes, for $n \leqslant 2^{H_\infty(P)}$,

$$V_n \left| \begin{smallmatrix} P'(o,s)=P'(o)P'(s) \\ H_\infty(P')\leqslant H_\infty(P) \end{smallmatrix} \right. = \boxed{(n+1)2^{-H_\infty(P)}}$$

which, as noted by Smith [2009], allows us to bound n-guess vulnerability using one-guess vulnerability, which is generally more tractable. Further, the bound is tight for $n \leqslant 2^{H_\infty(P)}$, being attained by any distribution assigning maximal individual probability $2^{-H_\infty P}$ to at least n elements.

The above bounds, assuming zero leakage, are useful in that they give us a gold standard: The minimum vulnerability that could be attained given a certain prior distribution (or class of distributions). We wish, however, to treat systems *with* leakage, in order to quantify how far the depart from this ideal. We thus proceed to relax our restriction on $P_{o|s}$, to account for possible leakage.

Bounds given leakage will only diverge from the no-leakage case after the first observation, thus we begin with the next simplest case: $n = 1$, two guesses and one observation. Given the distributions $P(s)$ and $P(o|s)$, we specialise Equation 6.4 to:

$$V_1 \left| \begin{smallmatrix} \{P_{o|s}\} \\ \{P_s\} \end{smallmatrix} \right. = P(\hat{s}_0) + \sum_o P(\hat{s}_1, o)$$

We may safely bound this from above by lifting the requirement that $\hat{s}_0$ and $\hat{s}_1$ are distinct, giving:

$$V_1 \left| \begin{smallmatrix} \{P_{o|s}\} \\ \{P_s\} \end{smallmatrix} \right. \leqslant \max_s P(s) + \sum_o \max_s P(o, s)$$

The equivalent bounds will, unfortunately, become more and more complex, and the approximation coarser, as n increases. The number of summands will increase exponentially (as $|O|^n$), quickly becoming intractable. What we need is an incremental approach that, given a measure of vulnerability in the current state, gives a safe (and hopefully tight) bound in the next. What we really want to bound, therefore, is the change in vulnerability between steps n and $n+1$. What we need is a notion of *leakage* that is compatible with our notion of vulnerability.

First, recall our notation, $V_0(k)$, introduced in Section 2.3, for the probability of compromise *on* the k^{th} guess. For $k = 1$ we have:

$$\begin{aligned} V_0 \left| \begin{smallmatrix} \{P_{o|s}\} \\ \{P_s\} \end{smallmatrix} \right. \left(1\right) &= V_1 \left| \begin{smallmatrix} \{P_{o|s}\} \\ \{P_s\} \end{smallmatrix} \right. - V_0 \left| \begin{smallmatrix} \{P_{o|s}\} \\ \{P_s\} \end{smallmatrix} \right. \\ &\leqslant \sum_o \max_s P(o,s) \\ &= \boxed{\sum_o \max_s P(o|s)P(s)} \end{aligned} \tag{6.8}$$

This bound is tight for any distribution that assigns maximal prior probability to at least 2 secrets.

This is Smith's bound [Smith, 2009] on the vulnerability of a probabilistic program (with $s = h$ and $o = l$). This bound is tight, but relies on knowing the full prior distribution, $P(s)$. We obtain more easily applied bounds if we follow Equation 6.7, and make a more liberal restriction on the distributions. Again characterising $P(s)$ by its min entropy, we have:

$$V_0 \left| \begin{smallmatrix} \{P_{o|s}\} \\ H_\infty(P) \leqslant H \end{smallmatrix} \right. \left(1\right) = \sup_{H_\infty(P_s) \leqslant H} \sum_o \max_s P(o|s)P(s)$$

If there exists an o' such that $P(o')$ is maximal[4], defining P_s by

$$P(s) = \begin{cases} 2^{-H} & s = \operatorname{argmax}_s P(o'|s) \\ \log_2 \frac{1-2^{-H}}{|S|-1} & \text{otherwise} \end{cases}$$

[4]This is certainly true for observations drawn from a finite set. This bound holds also for any complete set, bounded above e.g. a real interval. To show this, take a series of distributions, assigning maximum probability 2^{-H} to consecutive elements of a series of o whose conditional probability $P(o|s)$ converges on the supremum.

maximises the sum, giving:

$$V_0\left|\begin{array}{c}\{P_{o|s}\}\\ H_\infty(P)\leqslant H\end{array}\right.\left(1\right) = \sum_o \max_s P(o|s)2^{-H}$$
$$= \left(V_0\left|\begin{array}{c}\{P_{o|s}\}\\ H_\infty(P)\leqslant H\end{array}\right.\left(1\right)\right)\sum_o \max_s P(o|s)$$

or

$$\frac{V_0(1)}{V_0(0)} = \sum_o \max_s P(o|s)$$

To apply this result for $n > 1$, we first note that if guess $n-1$ was unsuccessful, and the chance that guess n succeeds is $V_0(n)$, then the min-entropy in the distribution $P(s|o_1 \ldots o_{n-1})$ is, by definition, $-\log_2 V_0(n)$ and thus that by summarising the (stepwise) prior by its min-entropy, and maximising over all distributions sharing it, we certainly have an upper bound on $V_0(n)$. Thus,

$$\frac{V_0(n)}{V_0(n-1)} \leqslant \sum_o \max_s P(o|s)$$

We thus arrive at a multiplicative (log-additive) definition of leakage, the *min leakage* of Köpf and Smith [2010]:

$$\mathcal{ML} = \boxed{\log_2 \sum_o \max_s P(o|s)} \tag{6.9}$$

We may now bound min-entropy iteratively by:

$$H_\infty(n+1) \geqslant H_\infty(n) - \log_2 \sum_o \max_s P(o|s)$$

We can further simplify the bound (at the cost of reducing precision), by summarising the likelihood $P(o|s)$ by $\max_o P(o|s)$. This may be practically useful when $|O|$ is large.

$$\begin{aligned}\mathcal{ML} &= \log_2 \sum_o \max_s(\max_o P(o|s))\\ &= \log_2|O| \max_{o,s} P(o|s)\\ &= \log_2|O| + \log_2 \max_{o,s} P(o|s)\\ &\leqslant \log_2|O|\end{aligned} \tag{6.10}$$

Equation 6.10 further implies that in the simple case that $P(s)$ is uniformly $1/|S|$,

$$V_{=n+1} \leqslant \boxed{\frac{|O|^n}{|S|}}$$

We have thus derived a number of formal bounds on vulnerability, including min leakage, from a concrete operational of a guessing attack. This gives us a formal link back to the bounds we analysed in Chapter 2. We will now see that once we consider refinement, we can also link together the remaining bounds, particularly those based on Shannon entropy.

Bounds under Refinement

Defining bounds in this way: prior vulnerability in terms of P_s and change in vulnerability (leakage) in terms of $P_{o|s}$ is useful both for dynamic evaluation, and for modularity. Firstly and most simply, given independence of observations, the security state of an uncompromised system can be summarised by the distribution $P(s|o_1, \ldots, o_i)$: the attacker's belief function. A safe approximation therefore, is to simply maintain the current value of $H_\infty(s|o_1, \ldots, o_i)$, and update it as every observation occurs.

Secondly, these two distributions act as a specification for the component to be mitigated. Consider an event-driven system, with two actions: **setup** and **respond**, where **setup** is responsible for choosing the secret, and **respond** for responding to the attacker's queries, incidentally generating observations. Then, statements of the form '**setup** chooses secret **s** with probability at least p' and 'given secret **s**, **respond** generates observation **o** with probability at least p' are both preserved by (probabilistic) refinement. To see that these imply our predicates Q_s and $Q_{o|s}$ (for a finite secret space), note that for $p \leqslant 1/|S|$:

$$\begin{aligned} \forall s.\, p \leqslant P(s) \quad &\Rightarrow \quad \forall s.\, P(s) \leqslant 1 - p(|S| - 1) \\ &\Rightarrow \quad -\log_2(1 - p(|S| - 1)) \leqslant H_\infty(P_s) \end{aligned}$$

Thus we have a bound on min entropy preserved by refinement. A similar argument gives a bound for $\mathcal{ML}$. Therefore to give an overall bound on vulnerability, it is necessary only to specify a component by an upper bound on H_∞ and $\mathcal{ML}$.

Finally, this approach applies to any sets Q_s and $Q_{o|s}$. In particular, we can derive vulnerability bounds for systems (and states) classified by

their Shannon (H_1) or Renyi (H_α) entropies. Recall the definition of Shannon entropy in Equation 2.10: $-\sum_s P(s) \log_2 P(s)$. Here, the appropriate definition of leakage is the mutual information: $L_1 = I(S; O)$.

The Renyi entropy is a generalisation of both the Shannon and min entropies, and is defined as: $1/(1-\alpha) \sum_s \log_2 P(s)^\alpha$. The limit as $\alpha \to 1$ gives H_1 and as $\alpha \to \infty$ gives H_∞. We define the leakage, L_α, as the expected change in H_α given an observation: $L_\alpha = H_\alpha(S) - \sum_o P(o) H_\alpha(S|o)$.

We bound vulnerability, for sets with bounded H_1 or H_α entropy, by calculating the greatest H_∞ entropy for any distribution in the set, and then proceeding as above. We presented this vulnerability correction (for Shannon entropy) in Section 2.5. An equivalent relation holds for the class of Renyi entropies, with the details presented in Appendix A. Even without appealing to the exact form of these bounds, we can nonetheless place them within the lattice together with the above min-entropy bounds, ordering the sets by inclusion, as Figure 6.1 shows for $n \leqslant 1$.

This lattice represents the ordering on bounds given by *safety*: $f \leqslant g$ if $\forall x.\ f\ x \leqslant g\ x$ i.e. g is a safe upper bound on f. Note that this is also the definition of refinement in pGCL: this is *also* the lattice of operational models (programs of the form given in Equation 6.4), ordered by refinement. We thus connect the lattice of vulnerability bounds, the *channel*-based view, with the refinement lattice, the *program*-based view. This completes the link from the formal, mechanised proofs of the last three chapters back to the mathematical models on which we based our initial analysis.

6.4 Related Work

The guessing-attack model, and the use of min entropy follow from the discussion in Chapter 2, and the related work discussed there is also relevant here. Capturing malicious (or in our terminology, demonic) behaviour by modelling a system as an adversarial game is a very common approach, particularly in controller synthesis. Harris et al. [2013], for example, use a game-based model to calculate a secure distribution of capabilities in the Capsicum system.

In particular, the simplified bounds that we derive coincide with those presented by Köpf and Smith [2010]. We arrive at similar results by different means: Köpf and Smith present their results in terms of a channel matrices (as we used in both Chapter 2 and Chapter 3), whereas the derivation just

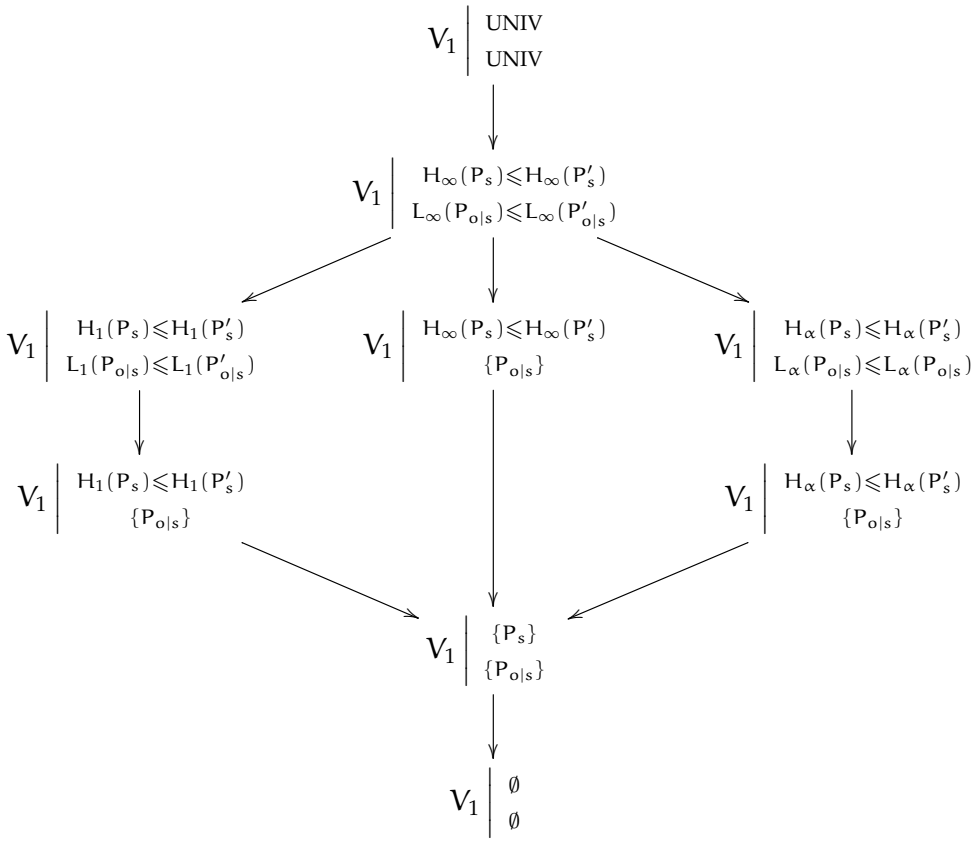

Figure 6.1: The lattice of bounds.

presented is in terms of refinements (or abstractions) of a concrete *process model*. Relating the two allows us to connect the concrete attack model with the information-theoretic view.

The basic countermeasure model that we analyse is *quantisation*, which is consistent with our established principle of reducing the signal (here, the number of distinct values), rather than increasing noise (as, for example in fuzzy time [Hu, 1991, 1992b]). A quantisation approach such as this, or that advocated by [Zhang et al., 2011] is ideally implemented by a lightweight mechanism such as scheduled delivery, as presented in Chapter 3.

Finally, in contrast with language or security type-based approaches [Zhang et al., 2012], we treat component-based systems, where a leaking component is considered only as a black box.

6.5 Summary

In this final chapter, we have taken one further step from the details of practical countermeasures, and from system verification, and investigated the *kind* of results that we might expect to be able to verify of a practical system. Remaining within the framework of pGCL, we formalised an operational, or programmatic model of the guessing attack of Chapter 2, and demonstrated that from it we can derive the principal bounds that we previously discussed in information-theoretic terms. This approach thus hints at a bridge between the process-oriented, semantic, view of leakage, and the channel-oriented, information-theoretic view. We took advantage of the fact that the guessing-attack model is not trace-dependent: security can be defined as a predicate on states, and thus is preserved by refinement, as noted in Section 4.1.

Of particular interest is our rederivation of the notion of min leakage in Equation 6.9, and the connection between the lattice on bounds, and the refinement order in pGCL, as expressed in Figure 6.1. The focus of this chapter was the establishment of sound information-theoretic bounds on leakage, that will survive refinement via the L4.verified proof stack, down to the level of real countermeasures, as discussed in Chapter 3 and Chapter 5.

In the preceding three chapters, we have demonstrated that we can build a chain of machine-checked proofs from a concrete probabilistic model (incorporating a large classical specification, such as seL4), all the way to abstract information-theoretic properties of the system modelled as a noisy channel. We thus confirm that the high-level approach we present in Chapter 2 can be linked to a concrete specification, which in turn may apply the countermeasures discussed in Chapter 3, or a more context-dependent approach, such as the lattice scheduler verified in Chapter 5.

7 Conclusion

We have surveyed the challenges of provable protection against information leakage in component-based systems, and contributed both to the understanding of the low-level mechanisms of leakage, and to the verification of high-level probabilistic properties and countermeasures.

Recognising that while min entropy gives a safe bound on vulnerability, we have demonstrated that Shannon entropy can lead to simpler, more tractable models (Section 2.5). Our proof (Lemma 4) of the maximum vulnerability given Shannon entropy allows such models to be used to safely bound vulnerability, through a pessimistic correction.

The extensive empirical analysis of Chapter 3 covered two local channels (cache and bus contention), and one remote (the lucky thirteen attack on DTLS), and demonstrates the widely varying effectiveness of three mitigation strategies: cache colouring, instruction-based scheduling and scheduled delivery.

Cache colouring is generally effective against the cache channel, although the capacity of residual channels is increasing on modern hardware, and a more careful implementation is required to avoid compromising artefacts, many of which are only apparent given a large amount of data.

Instruction-based scheduling, which seeks to prevent the use of the preemption tick as a clock, has decreased dramatically in effectiveness on recent hardware. On older (circa 2005) ARM processors it is essentially perfect, while on more recent ARM processors, and on more complex x86 processors it is largely ineffective, with large residual channels apparent. Although applicable to a wide range of channels (e.g. bus contention), its performance does not justify the difficulty of attempting to remove all visible clocks.

The Achilles' heel of both local-channel countermeasures is their reliance

on unspecified low-level hardware behaviour, which manufacturers are likely to modify without notice in pursuit of greater performance. It appears that for genuinely covert-channel-secure systems, partitioned hardware will remain necessary, although effective mitigation of side-channel leakage, particularly through the cache appears to be practical (via colouring).

For the remote channel (lucky thirteen), OS-level techniques allow us to efficiently and effectively mitigate the channel, without requiring any modification of the vulnerable code in OpenSSL. In fact, we outperform the constant-time implementation in the latest version of the library, both for security and performance. There is no reason that a more careful implementation should not be able to achieve perfect mitigation at negligible performance cost.

The work presented in Chapter 4 through Chapter 6, demonstrates that the refinement-style technique used in L4.verified can be extended to handle probabilistic properties and probabilistic systems, while incorporating existing proofs. Our formalisation of pGCL in Isabelle/HOL allows us to verify the lattice-lottery scheduler, as an asymptotically fair countermeasure against the cache channel. Finally, we formally derive (with machine-checked proof) the information-theoretic bounds introduced in Chapter 2 from a concrete threat model: the guessing attack.

We conclude that a significant amount of work remains before we will be able to formally verify the total absence of timing channels in a system implemented on commodity hardware. Nevertheless, we have demonstrated that effective mitigations do exist for some channels, and that there is no barrier in principle to verifying such stochastic high-level properties.

A Detailed Proofs

This appendix presents the detailed proofs of several results from Chapter 2, including the extension of the vulnerability divergence result, Lemma 4 of Section 2.5, to the full class of Rényi entropies [Rényi, 1961], of which the Shannon entropy is a special case.

Proof of Lemma 1

Let X be a random variable, ranging over the integers $[1, n]$. For any fixed $y \in \mathbb{R}$, then over all distributions where $E(X) = y$, $E(1/X)$ is maximised when all probability mass is assigned either to 1 or to n.

The result is trivial for $n \leqslant 2$, therefore assume $3 \leqslant n$. Fix a distribution P such that $E_P(X) = y$, and assume that $E_P(1/X)$ is maximal among such distributions. Further assume that for some $x \in [2, n-1]$, $P(x) > 0$. Let:

$$\alpha = \frac{n-x}{n-1} \qquad \beta = \frac{x-1}{n-1}$$
$$\text{n.b.} \quad \alpha + n\beta = x \tag{A.1}$$

Construct a modified distribution P' where:

$$P'(z) = \begin{cases} P(z) + \alpha P(x) & z = 1 \\ 0 & z = x \\ P(z) + \beta P(x) & z = n \\ P(z) & \text{otherwise} \end{cases}$$

We have redistributed the probability that had been assigned to x between 1

and n. Calculating $E_{P'}(X)$ we see that

$$\begin{aligned}
E_{P'}(X) &= \sum_z zP'(z) \\
&= [P(1) + \alpha P(x)] + 0 + [n(P(n) + \beta P(x))] + \sum_{z\in[2,n-1]-x} zP(z) \\
&= \alpha P(x) + n\beta P(x) + \sum_{z\in[1,n]-x} zP(z) \\
&= xP(x) + \sum_{z\in[1,n]-x} zP(z) \qquad \text{By Equation A.1} \\
&= E_P(X)
\end{aligned}$$

Also,

$$\begin{aligned}
E_{P'}(1/X) &= \sum_z \frac{P'(z)}{z} \\
&= [P(1) + \alpha P(x)] + 0 + \left[\frac{P(n) + \beta P(x)}{n}\right] + \sum_{z\in[2,n-1]-x} \frac{P(z)}{z} \\
&= (\alpha + \frac{\beta}{n})P(x) + \sum_{z\in[1,n]-x} \frac{P(z)}{z} \\
&> E_P(1/X) \quad \text{if } \alpha + \frac{\beta}{n} > \frac{1}{x} \qquad \text{(A.2)}
\end{aligned}$$

It remains only to show that the inequality in Equation A.2 holds i.e. that we have strictly increased the value of $E_{P'}(1/X)$:

$$\begin{aligned}
n + x < nx \qquad & \text{as } 3 \leqslant n, 2 \leqslant x \\
\therefore (n+x)(n-x) < nx(n-x) \qquad & \text{as } x < n \\
\therefore (n+x)(n-x) < nx(n-x) + n - x \qquad & \text{"} \\
\therefore 0 < n^2x - nx^2 - n^2 + x^2 + n - x &
\end{aligned}$$

also

$$\begin{aligned}
n < n^2 \qquad & \text{as } 3 \leqslant n \\
\therefore 0 < n^2x - nx \qquad & \text{as } 2 \leqslant x
\end{aligned}$$

and thus

$$\begin{aligned} 0 &< \frac{n^2x - nx^2 + x^2 - x + n^2 - n}{n^2x - nx} \\ &= \frac{n-x}{n-1} + \frac{x-1}{n^2-n} - \frac{1}{x} \\ &= \alpha + \frac{\beta}{n} - \frac{1}{x} \end{aligned}$$

finally therefore

$$\alpha + \frac{\beta}{n} > \frac{1}{x}$$

This contradicts the assumption of maximality. Therefore, any distribution that maximises $E(1/X)$ only assigns probability to 1 and n.

Proof of Lemma 4

Let

$$Q = \{P : H_1(P) = H\}$$

be the set of distributions over X with Shannon entropy H, and for each $P \in Q$, take any disjoint partition of X as $Y \cup Z$ such that

$$\forall y \in Y, z \in Z.\ y \geqslant z \wedge |Y| = n$$

Then, for $H \geqslant \log_2|Y|$, $P(Y)$ is maximised over Q at the 'corner point', where $\forall y.\ P(y) = 1/|Y|$ and $\forall z.\ P(z) = 1/|Z|$.

For $H < \log_2|Y|$, a solution exists with $P(Y) = 1$ and $P(X) = 0$.

The partition of X induces a partition of P:

$$P(x) = \begin{cases} P(Y) * P_Y(x) & x \in Y \\ (1 - P(Y)) * P_Z(x) & x \in Z \end{cases} \qquad \text{where } P_S(x) = \frac{P(x)}{P(S)}$$

Let $N = |X|$, hence $|Z| = N - n$. Also, let $p = P(Y)$.

The question is: for a given H and n, what is the largest possible p, over all choices for P? Or, what is the largest probability we can assign to the n most likely events, and still achieve overall entropy H?

Note that:

$$H_1(P_X) = h(p) + pH_1(P_Y) + (1-p)H_1(P_Z) \tag{A.3}$$

where

$$h(p) = -p\log_2(p) - (1-p)\log_2(1-p) \qquad \text{and } 0\log_2 0 = 0$$

and that

$$\begin{aligned} 0 \leqslant H_1(P_X) &\leqslant \log_2 N \\ 0 \leqslant H_1(P_Y) &\leqslant \log_2 n \\ \text{and} \quad 0 \leqslant H_1(P_Z) &\leqslant \log_2(N-n) \end{aligned}$$

If $H \leqslant \log_2 n$, a solution exists with $H(Y) = H$, $p = 1$. Assume now that:

$$H > \log_2 n \tag{A.4}$$

We rewrite equation Equation A.3 as:

$$H(Z) = \frac{H - h(p)}{1-p} - \frac{p}{1-p}H(Y) \tag{A.5}$$

For a given p, the solutions form a line in the $H(Y)$–$H(Z)$ plane with slope between 0 at $p = 0$, and $-\infty$ at $p = 1$. What is the greatest p such that this line intersects the rectangle $(0,0)$–$(\log_2 n, \log_2(N-n))$?

As the slope is uniformly non-positive, a solution exists exactly when one exists on the line:

$$\frac{H(Y)}{\log_2 n} = \frac{H(Z)}{\log_2(N-n)} \tag{A.6}$$

Reparameterising and substituting into Equation A.3:

$$H = h(p) + pk\log_2 n + (1-p)k\log_2(N-n) \tag{A.7}$$

$$k(p) = k = \frac{H - h(p)}{p\log_2 n + (1-p)\log_2(N-n)} \tag{A.8}$$

We now need the greatest p such that $k(p) \in [0,1]$. Note that

$$k(1) = \frac{H}{\log_2(n)} \tag{A.9}$$

$$> 1 \qquad \text{by Equation A.4} \tag{A.10}$$

Assume that for some $q \in [0,1]$, $k(q) \leqslant 1$. As $k(1) > 1$, by the continuity of k, there exist finitely many $q' \in [q,1]$ with $k(q') = 1$. Let r be the greatest such. Again by continuity, $\nexists q' > q.\ k(q') \leqslant 1$. Thus the largest value of p (r) is found when $k(p) = 1$ i.e. when $H(Y)$ and $H(Z)$ attain their maximum values: $\log_2 n$ and $\log_2(N-n)$. This occurs only for the uniform distributions $P_X(x) = \frac{1}{n}$ and $P_Y(y) = \frac{1}{N-n}$.

Vulnerability given Rènyi Entropy

Definition 7: The Rényi α-entropy of the distribution P over the set X is:

$$H_\alpha(P) = \lim_{a \to \alpha} \left(\frac{1}{1-a} \log_2 \sum_{x \in X} P(x)^a \right)$$

This subsumes both the Shannon entropy, when $\alpha = 1$, and the min-entropy, when $\alpha = \infty$. Thus the divergence of H_1 is covered by Lemma 4, and that of H_∞ is trivial.

Lemma 16: For $\alpha \neq 1$, let

$$Q = \{P : H_\alpha(P) = H\}$$

be the set of distributions over X with Renyi α-entropy H, and for each $P \in Q$, partition X disjointly as $Y + Z$ such that

$$\forall y \in Y, z \in Z.\ y \geqslant z \ .$$

For $H \geqslant \log_2|Y|$, $P(Y)$ is maximised over Q at the 'corner point', where $\forall y.\ P(y) = 1/|Y|$ and $\forall z.\ P(z) = 1/|Z|$.

For $H < \log_2|Y|$, a solution exists with $P(Y) = 1$ and $P(X) = 0$.

Proof. The proof is analogous to that of Lemma 4.

The partition of X induces a partition of P:

$$P(x) = \begin{cases} p * P_Y(x) & x \in Y \\ (1-p) * P_Z(x) & x \in Z \end{cases} \qquad \text{where } P_S(x) = \frac{P(x)}{P(S)}$$

Let $n = |Y|$, $N = |X|$; hence $|Z| = N - n$.

Let:

$$H_\alpha(X) = \frac{1}{1-\alpha} \log_2 \sum_{x \in X} P(x)^\alpha \tag{A.11}$$

and define $H_\alpha(Y)$ and $H_\alpha(Z)$ analogously.

Note that

$$0 \leqslant H_\alpha(X) \leqslant \log_2 N$$
$$0 \leqslant H_\alpha(Y) \leqslant \log_2 n$$
$$0 \leqslant H_\alpha(Z) \leqslant \log_2(N-n)$$

Expanding Equation A.11 and rearranging:

$$\begin{aligned}
H_\alpha(X) &= \frac{1}{1-\alpha}\log_2\left[\sum_{y\in Y}[pP_Y(y)]^\alpha + \sum_{z\in Z}[(1-p)P_Z(z)]^\alpha\right] \\
&= \frac{1}{1-\alpha}\log_2\left[p^\alpha\sum_{y\in Y}P_Y(y)^\alpha + (1-p)^\alpha\sum_{z\in Z}P_Z(z)^\alpha\right] \\
&= \frac{1}{1-\alpha}\log_2\left[p^\alpha 2^{(1-\alpha)H_\alpha(Y)} + (1-p)^\alpha 2^{(1-\alpha)H_\alpha(Z)}\right] \\
2^{(1-\alpha)H_\alpha(X)} &= p^\alpha 2^{(1-\alpha)H_\alpha(Y)} + (1-p)^\alpha 2^{(1-\alpha)H_\alpha(Z)}
\end{aligned}$$

As for Shannon entropy, if $H_\alpha(X) \leqslant \log_2 n$, we find a solution with $p = 1$ and $H_\alpha(Y) = H_\alpha(X)$. Therefore, assume:

$$H_\alpha(X) > \log_2 n \tag{A.12}$$

Let:

$$A = 2^{(1-\alpha)H_\alpha(X)}$$

$$B = 2^{(1-\alpha)H_\alpha(Y)} \tag{A.13}$$

$$C = 2^{(1-\alpha)H_\alpha(Z)} \tag{A.14}$$

For $\alpha \neq 1$, equations Equation A.13 and Equation A.14 give a continuous, invertible change of variables. Thus, it is sufficient to solve the following:

$$A = p^\alpha B + (1-p)^\alpha C \tag{A.15}$$

As in Equation A.5, we again have a line of negative slope, this time in the $B - C$ plane.

There are two cases to consider: for $\alpha < 1$,

$$\begin{aligned}
n^{1-\alpha} \leqslant A \leqslant N^{1-\alpha} \\
1 \leqslant B \leqslant n^{1-\alpha} \\
1 \leqslant C \leqslant (N-n)^{1-\alpha}
\end{aligned}$$

and for $\alpha > 1$,

$$\begin{aligned}
N^{1-\alpha} \leqslant A \leqslant n^{1-\alpha} \\
n^{1-\alpha} \leqslant B \leqslant 1 \\
(N-n)^{1-\alpha} \leqslant C \leqslant 1
\end{aligned}$$

Again we reparameterise, looking for solutions on the diagonal:

$$C = p^{\alpha}(1 + k(n^{1-\alpha} - 1)) + (1-p)^{\alpha}(1 + k((N-n)^{1-\alpha} - 1))$$
$$k(p) = \frac{C - p^{\alpha} - (1-p)^{\alpha}}{p^{\alpha}(n^{1-\alpha} - 1) + (1-p)^{\alpha}((N-n)^{1-\alpha} - 1)} \tag{A.16}$$

Equation A.16 is continuous for $p \in [0,1]$ and $n \in [0,N]$ and $N > 2$. Also:

$$\begin{aligned} k(1) &= \frac{C-1}{n^{1-\alpha} - 1} \\ &= \frac{2^{(1-\alpha)H_{\alpha}(X)} - 1}{2^{(1-\alpha)\log_2 n} - 1} \\ &> 1 \qquad \text{by ass. A.12} \end{aligned}$$

Thus the solution that maximises p will again be found at the extreme point of the rectangle, namely $B = n^{1-\alpha}$, $C = (N-n)^{1-\alpha}$. Substituting into Equation A.15 gives:

$$2^{(1-\alpha)H_{\alpha}(X)} = p^{\alpha}n^{1-\alpha} + (1-p)^{\alpha}(N-n)^{1-\alpha}$$
$$H_{\alpha}(X) = \frac{1}{1-\alpha}\log_2\left[p^{\alpha}n^{1-\alpha} + (1-p)^{\alpha}(N-n)^{1-\alpha}\right]$$

□

Bibliography

Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 526–540, San Francisco, CA, May 2013. IEEE. doi:10.1109/SP.2013.42.

Mário S. Alvim, Kostas Chatzikokolakis, Catuscia Palamidessi, and Geoffrey Smith. Measuring information leakage using generalized gain functions. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 265–279. IEEE, 2012. doi:10.1109/CSF.2012.26.

Miguel E. Andrés, Catuscia Palamidessi, Peter Rossum, and Geoffrey Smith. Computing the leakage of information-hiding systems. In Javier Esparza and Rupak Majumdar, editors, *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 6015 of *Lecture Notes in Computer Science*, pages 373–389. Springer, 2010. doi:10.1007/978-3-642-12002-2_32.

Suguru Arimoto. An algorithm for computing the capacity of arbitrary discrete memoryless channels. *IEEE Transactions on Information Theory*, 18(1):14–20, 1972. doi:10.1109/TIT.1972.1054753.

Cortex A9 TRM. *Cortex-A9 (revision r3p0) Technical Reference Manual*. ARM Ltd., July 2011.

Ross Arnold and Tim Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the 1997 Data Compression Conference*, pages 201–210. IEEE, 1997. doi:10.1109/DCC.1997.582019.

Aslan Askarov, Andrew C. Myers, and Danfeng Zhang. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 520–538, Chicago, Illinois, USA, 2010. ACM. doi:10.1145/1866307.1866341.

Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security*, pages 103–108, Chicago, Illinois, USA, 2010a. ACM. doi:10.1145/1866835.1866854.

Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, 2010b. USENIX.

Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, and Caroline Sporleder. Acoustic side-channel attacks on printers. In *Proceedings of the 19th USENIX Security Symposium*, pages 1–16, Washington, DC, 2010. USENIX.

Christel Baier and Marta Kwiatkowska. Automatic verification of liveness properties of randomized systems. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, page 295. ACM, 1997. doi:10.1145/259380.259527.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 90–101, Savannah, GA, USA, 2009. ACM. doi:10.1145/1480881.1480894. URL http://doi.acm.org/10.1145/1480881.1480894.

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22791-2. doi:10.1007/978-3-642-22792-9_5. URL http://dx.doi.org/10.1007/978-3-642-22792-9_5.

Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 186–197. IEEE, 2012a. doi:10.1109/CSF.2012.17.

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the*

39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 97–110, Philadelphia, PA, USA, 2012b. ACM. doi:10.1145/2103656.2103670. URL http://doi.acm.org/10.1145/2103656.2103670.

Gertrud Bauer and Markus Wenzel. Calculational reasoning revisited – an Isabelle/Isar experience. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, pages 75–90. Springer, 2001.

Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, Vancouver, BC, Canada, 2010. USENIX.

Richard E. Blahut. Computation of channel capacity and rate-distortion functions. *IEEE Transactions on Information Theory*, 18:460–473, 1972. doi:10.1.1.133.7174.

Christelle Braun, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Quantitative notions of leakage for one-try attacks. *Electronic Notes in Theoretical Computer Science*, 249:75–91, August 2009. doi:10.1016/j.entcs.2009.07.085.

David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, Washington, DC, USA, 2003. USENIX. doi:10.1016/j.comnet.2005.01.010.

Lewis Carroll. *Alice's Adventures in Wonderland*. Macmillan, November 1865.

Han Chen and Pasquale Malacaria. Quantitative analysis of leakage for multi-threaded programs. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 31–40, San Diego, California, USA, 2007. ACM. doi:10.1145/1255329.1255335.

Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 31–45. IEEE, June 2005. doi:10.1109/CSFW.2005.10.

Aaron R. Coble. *Anonymity, information, and machine-assisted proof*. PhD thesis, University of Cambridge, Computer Laboratory, 2010.

David Cock. Bitfields and tagged unions in C: Verification through automatic generation. In Bernhard Beckert and Gerwin Klein, editors, *Proceedings of the 5th International Verification Workshop*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, August 2008.

David Cock. Lyrebird – assigning meanings to machines. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Proceedings of the 5th Systems Software Verification*, pages 1–9, Vancouver, Canada, October 2010. USENIX.

David Cock. Exploitation as an inference problem. In *Proceedings of the 4th ACM Workshop on Artificial Intelligence and Security*, pages 105–106, Chicago, IL, USA, October 2011. ACM. doi:10.1145/2046684.2046702.

David Cock. Verifying probabilistic correctness in Isabelle with pGCL. In *Proceedings of the 7th Systems Software Verification*, pages 1–10, Sydney, Australia, November 2012. Electronic Proceedings in Theoretical Computer Science. doi:10.4204/EPTCS.102.15.

David Cock. Practical probability: Applying pGCL to lattice scheduling. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, pages 1–16, Rennes, France, July 2013. Springer. doi:10.1007/978-3-642-39634-2_23.

David Cock. From probabilistic operational semantics to information theory; side channels in pGCL with isabelle. In *Proceedings of the 5th International Conference on Interactive Theorem Proving*, pages 1–15, Vienna, Austria, July 2014. Springer. doi:10.1007/978-3-319-08970-6_12.

David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Mu noz, and Sofiène Tahar, editors, *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 167–182, Montreal, Canada, August 2008. Springer. doi:10.1007/978-3-540-71067-7_16.

David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile; an empirical study of timing channels on sel4. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 1–12, Scottsdale, USA, November 2014. ACM. doi:10.1145/2660267.2660294. (to appear).

Cryptome. How old is TEMPEST?, 2002. URL `http://cryptome.org/tempest-old.htm`. Accessed 11 March 2014.

Matthias Daum, Nelson Billing, and Gerwin Klein. Concerned with the unprivileged: User programs in kernel refinement. *Formal Aspects of Computing*. `doi:10.1007/s00165-014-0296-9`. (to appear).

Matteo Dell'Amico, Pietro Michiardi, and Yves Roudier. Password strength: An empirical analysis. In *Proceedings of the 30th IEEE INFOCOM*, pages 1–9. IEEE, 2010. `doi:10.1109/INFCOM.2010.5461951`.

Dorothy. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–242, 1976. `doi:10.1145/360051.360056`.

Philip Derrin, Kevin Elphinstone, Gerwin Klein, David Cock, and Manuel M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, Portland, OR, USA, September 2006. ACM. `doi:10.1145/1159842.1159850`.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975. `doi:10.1145/360933.360975`.

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

Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified protection model of the seL4 microkernel. In Jim Woodcock and Natarajan Shankar, editors, *Proceedings of Verified Software: Theories, Tools and Experiments 2008*, volume 5295 of *Lecture Notes in Computer Science*, pages 99–114, Toronto, Canada, October 2008. Springer.

Barbara Espinoza and Geoffrey Smith. Min-entropy leakage of channels in cascade. In Gilles Barthe, Anupam Datta, and Sandro Etalle, editors, *Proceedings of the 10th International Workshop on Formal Aspect of Security and Trust (FAST)*, volume 7140 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2012. `doi:10.1007/978-3-642-29420-4_5`.

Barbara Espinoza and Geoffrey Smith. Min-entropy as a resource. *Information and Computation*, 226:57–75, 2013. doi:10.1016/j.ic.2013.03.005.

Colin Fidge and Carron Shankland. But what if i don't want to wait forever? *Formal Aspects of Computing*, 14:281–294, 2003. doi:10.1007/s001650300006.

Bryan Ford. Plugging side-channel leaks with timing information flow control. In *Proceedings of the 4th USENIX Workschop on Hot Topics in Cloud Computing*, pages 1–5, Boston, MA, 2012. USENIX.

Richard Gay, Heiko Mantel, and Henning Sudbrock. Empirical bandwidth analysis of interrupt-related covert channels. In *Proceedings of the 2nd International Workshop on Quantitative Aspects in Security Assurance*, London, September 2013.

Michael Godfrey. On the prevention of cache-based side-channel attacks in a cloud environment. Master's thesis, Queen's University, Ontario, Canada, September 2013. doi:1974/8320.

Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, USA, April 1982. IEEE Computer Society.

Xun Gong, Negar Kiyavash, and Parv Venkitasubramaniam. Information theoretic analysis of side channel information leakage in FCFS schedulers. In *Proceedings of the 2011 IEEE International Symposium on Information Theory*, pages 1255–1259. IEEE, August 2011. doi:10.1109/ISIT.2011.6033737.

James W. Gray. Probabilistic interference. In *Proceedings of the 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 170–179, May 1990. doi:10.1109/RISP.1990.63848.

James W. Gray. On introducing noise into the bus-contention channel. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 90–98. IEEE Computer Society, 1993. doi:10.1109/RISP.1993.287640.

James W. Gray. Countermeasures and tradeoffs for a class of covert timing channels. Technical Report HKUST-CS94-18, Hong Kong University of Science and Technology, 1994. doi:1783.1/25.

Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Performance Evaluation*, 73(0):110–132, 2014. doi:10.1016/j.peva.2013.11.004.

David Greve and Matthew Wilding. Evaluatable, high-assurance microprocessors. In *Proceedings of the 2nd Annual High-Confidence Software and Systems Conference*, Annapolis, MD, USA, 2002.

W.R. Harris, S. Jha, T. Reps, J. Anderson, and R.N.M. Watson. Declarative, temporal, and practical programming with capabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 18–32, May 2013. doi:10.1109/SP.2013.11.

William L. Harrison and Richard B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(6):837–891, November 2005. doi:10.1017/S0956796805005666.

Johannes Hölzl and Tobias Nipkow. Verifying pCTL model checking. In Cormac Flanagan and Barbara König, editors, *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *Lecture Notes in Computer Science*, pages 347–361. Springer, 2012. doi:10.1007/978-3-642-28756-5_24.

Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20. IEEE Computer Society, 1991. doi:10.1109/RISP.1991.130768.

Wei-Ming Hu. Lattice scheduling and covert channels. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–61. IEEE, 1992a. doi:10.1109/RISP.1992.213271.

Wei-Ming Hu. Reducing timing channels with fuzzy time. *Journal of Computer Security*, 1(3–4), January 1992b. doi:10.3233/JCS-1992-13-404.

Brian Huffman. Formal verification of monad transformers. In *Proceedings of the 17th International Conference on Functional Programming*, pages 15–16, Copenhagen, Denmark, 2012. ACM. doi:10.1145/2364527.2364532.

Marieke Huisman and Tri Minh Ngo. Scheduler-specific confidentiality for multi-threaded programs and its logic-based verification. In *Proceedings of the 2011 International Conference on Formal Verification of Object-Oriented Software*, pages 178–195, Turin, Italy, 2012. Springer. doi:10.1007/978-3-642-31762-0_12.

Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in HOL. *Theoretical Computer Science*, 346(1):96 – 112, 2005. doi:10.1016/j.tcs.2005.08.005.

Intel 64 & IA-32 AORM. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, April 2012.

Paul A. Karger and John C. Wray. Storage channels in disk arm optimization. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 52–61. IEEE Computer Society, May 1991. doi:10.1109/RISP.1991.130771.

Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering*, 17(11):1147–1165, November 1991. doi:10.1109/32.106971.

Richard A. Kemmerer. Shared resource matrix methodology: an approach to identifying storage and timing channels. *ACM Transactions on Computer Systems*, 1(3):256–277, August 1983. doi:10.1145/357369.357374.

Richard A. Kemmerer. A practical approach to identifying storage and timing channels: twenty years later. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC)*, pages 109–118. IEEE, December 2002. doi:10.1109/CSAC.2002.1176284.

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

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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM. doi:10.1145/1629575.1629596.

Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, June 2010. doi:10.1145/1743546.1743574.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1): 2:1–2:70, February 2014. doi:10.1145/2560537.

Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999. doi:10.1007/3-540-48405-1_25.

Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 286–296, Alexandria, Virginia, USA, 2007. ACM. doi:10.1145/1315245.1315282.

Boris Köpf and Geoffrey Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, pages 44–56, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/CSF.2010.11.

Dexter Kozen. A probabilistic PDL. *Journal of Computer and System Sciences*, 30 (2):162–178, 1985. doi:10.1016/0022-0000(85)90012-1.

Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973. doi:10.1145/362375.362389.

Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, Montreal, Canada, June 1997. IEEE. doi:10.1109/RTTAS.1997.601360.

Steven. B. Lipner. A comment on the confinement problem. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles*, pages 192–196. ACM, 1975. doi:10.1145/800213.806537.

David Malone and Kevin Maher. Investigating the distribution of password choices. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 301–310, Lyon, France, 2012. ACM. doi:10.1145/2187836.2187878.

James L. Massey. Guessing and entropy. In *Proceedings of the 1994 IEEE International Symposium on Information Theory*, page 204. IEEE, June 1994. doi:10.1109/ISIT.1994.394764.

Daniel Matichuk and Toby Murray. Extensible specifications for automatic re-use of specifications and proofs. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, pages 1–8, Thessaloniki, Greece, December 2012. Springer. doi:10.1007/978-3-642-33826-7_23.

Annabelle McIver and Carroll Morgan. Partial correctness for probabilistic demonic programs. *Theoretical Computer Science*, 266(1-2):513 – 541, 2001. doi:10.1016/S0304-3975(00)00208-5.

Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2004. ISBN 978-0-387-40115-7. doi:10.1007/b138392.

Carroll Morgan and Annabelle K. McIver. An expectation-based model for probabilistic temporal logic. *Logic Journal of the IGPL*, 7:779–804, 1999. doi:10.1093/jigpal/7.6.779.

Carroll Morgan, Annabelle McIver, Geoffrey Smith, Barbara Espinoza, and Larisa Meinicke. Abstract channels and their robust information-leakage ordering. In *Principles of Security and Trust (ETAPS)*, pages 83–102, Grenoble, France, April 2014.

Till Mossakowski, Lutz Schröder, and Sergey Goncharov. A generic complete dynamic logic for reasoning about purity and effects. *Formal Aspects of Computing*, 22(3-4):363–384, May 2010. doi:10.1007/s00165-010-0153-4.

Steven J. Murdoch. Hot or not: revealing hidden services by their clock skew. In *Proceedings of the 13th ACM Conference on Computer and Communications*

Security, pages 27–36, Alexandria, Virginia, USA, 2006. ACM. doi:10.1145/1180405.1180410.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In Chris Hawblitzel and Dale Miller, editors, *Proceedings of the 2nd International Conference on Certified Programs and Proofs*, volume 7679 of *Lecture Notes in Computer Science*, pages 126–142, Kyoto, Japan, December 2012. Springer. doi:10.1007/978-3-642-35308-6_12.

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 *Proceedings of the IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, USA, May 2013. IEEE. doi:10.1109/SP.2013.35.

Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002. doi:10.1007/978-94-010-0413-8_11.

Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 978-3-540-43376-7. doi:10.1007/3-540-45949-9.

NSA. TEMPEST: A signal problem. *Cryptologic Spectrum*, September 1972. Declassified 2007.

Alfréd Rényi. On measures of entropy and information. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability*, pages 547–561, 1961.

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 *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 199–212, Chicago, IL, USA, 2009. ACM. doi:10.1145/1653662.1653687.

Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978. doi:10.1145/359340.359342.

Marvin Schaefer, Barry Gold, Richard Linde, and John Scheid. Program confinement in KVM/370. In *Proceedings of the 5th ACM Computer Science Conference*, pages 404–410, Atlanta, GA, USA, 1977. ACM. doi:10.1145/800179.1124633.

Steve Selvin. A problem in probability (letter to the editor). *American Statistician*, 29(1):67, February 1975.

Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 1948. doi:10.1145/584091.584093. Reprinted in SIGMOBILE Mobile Computing and Communications Review, 5(1):3–55, 2001.

Olin Sibert, Phillip A Porras, and Robert Lindell. The Intel 80x86 processor architecture: Pitfalls for secure systems. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 1995. doi:10.1.1.17.5041.

Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, pages 288–302, York, UK, 2009. Springer. doi:10.1007/978-3-642-00596-1_21.

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 *Proceedings of the 18th European Symposium On Research in Computer Security*, pages 718–735, Egham, UK, September 2013. Springer. doi:10.1007/978-3-642-40203-6_40.

David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared L2 caches on multicore systems in software. In *Proceedings of the 3rd Workshop on the Interaction between Operating Systems and Computer Architecture*, San Diego, CA, USA, June 2007.

Jonathan T. Trostle. Modelling a fuzzy time system. In *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 82–89. IEEE Computer Society, 1993. doi:10.1109/RISP.1993.287641.

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

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

Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, USA, November 1994. USENIX.

Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 157–167, Paris, France, July 2013. `doi:10.1109/ECRTS.2013.26`.

Chelsea C. White and Douglas J. White. Markov decision processes. *European Journal of Operational Research*, 39(1):1–16, 1989. `doi:10.1016/0377-2217(89)90348-2`.

Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 500–515, Munich, Germany, August 2009. Springer. `doi:10.1007/978-3-642-03359-9_34`.

John C. Wray. An analysis of covert timing channels. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–7. IEEE, May 1991. `doi:10.1109/RISP.1991.130767`.

Yaming Yu. Squeezing the Arimoto-Blahut algorithm for faster convergence. *IEEE Transactions on Information Theory*, 56(7):3149–3157, 2010. `doi:10.1109/TIT.2010.2048452`.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 563–574, Chicago, IL, USA, 2011. ACM. `doi:10.1145/2046707.2046772`.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 2012 ACM*

SIGPLAN Conference on Programming Language Design and Implementation, pages 99–110, Beijing, China, 2012. ACM. doi:10.1145/2254064.2254078.

Kehuan Zhang, Zhou Li, Rui Wang, XiaoFeng Wang, and Shuo Chen. Sidebuster: Automated detection and quantification of side-channel leaks in web application development. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 595–606, New York, NY, USA, 2010. ACM. doi:10.1145/1866307.1866374.