On Inexpensive Methods for Improving Security of Embedded Systems

Kostengünstige Maßnahmen zur Erhöhung
der Sicherheit eingebetteter Systeme

Der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Johannes Bauer aus Lichtenfels

Als Dissertation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Tag der mündlichen Prüfung:	25. November 2016
Vorsitzender des Promotionsorgans:	Prof. Dr.-Ing. Reinhard Lerch
Gutachter:	Prof. Dr.-Ing. Felix Freiling
	Prof. Dr.-Ing. habil. Falko Dressler

Contents

List of Figures

List of Tables

Listings

Abstract

We usually scrutinize security of embedded systems under an extraordinarily sophisticated attacker model: the adversary has physical possession of the target and unlimited time to break it. For the defensive side, this forms an exceptionally challenging scenario. This thesis studies fortification of systems against such adversaries. The principal contributions lie in the field of embedded security, where we explore methods of building secure systems in a resource-efficient manner. This allows implementation of our countermeasures on resource-constrained microcontrollers. While these have a detrimental effect on runtime performance, the cost of the hardware itself remains unaffected, thereby providing an attractive and inexpensive alternative to hardware countermeasures. Next, we will briefly outline our contributions.

Attacks such as *Differential Power Analysis* (DPA) enable adversaries to exploit even the most minute differences in data dependent energy consumption. To make it more difficult for attackers to gain access to secrets within a chip, effective countermeasures need to be employed. One technique, implemented using only software, is described by us as a first contribution. We use binary recompilation to achieve binary code polymorphism. This causes different characteristic emission patterns for each call of a protected cryptographic primitive. Due to extensive and sophisticated pre-calculations which we perform at compile time, execution is extremely fast during runtime.

Since not only power consumption but also timing differences are something that attackers can exploit with great accuracy, we studied detection of timing leaks. Considering the architecture of today's increasingly complex microcontrollers, manual estimation of runtime has become virtually infeasible. Therefore, as a second contribution, we developed a behavioral Cortex-M core emulator which permits cycle-accurate simulation. We show how to incorporate such an emulator in a semi-automatic vetting process. After compilation, all security-relevant routines within the code are analyzed and checked for timing discrepancies.

The complexity of modern microcontroller units (MCUs) is shown from a different angle when considering attackers who can manipulate firmware. Since the reduction of electromagnetic interference (EMI) is an important goal of system designers, many recent MCUs already include software-tunable EMI countermeasures. In our third contribution, we show how these anti-EMI peripherals can be abused to construct covert channels. Unfortunately for the defensive side, these channels operate in the radio frequency domain and thus could be used for wireless transmission of data — even when the benign application was never intended to perform such communication. We describe how changes in parasitic electromagnetic emission can be used to encode data and what hardware is necessary to recover this data.

To increase the resistance of embedded systems against physical attacks, it is common to use special semiconductors which employ hardware countermeasures. The downside of such integration is that the specialized device usually dictates the exact cryptographic construction. How such hardware can be used nevertheless to augment general-purpose microcontrollers is something we focus on with our fourth contribution. As a demonstration, we incorporate a hardware security module in the handshake of the transport layer security (TLS) protocol. We do so without the need to create a custom cipher suite and without modifying the TLS handshake itself; instead, we use a generic approach by relying on implementation-specific protocol invariants and therefore get around the limitations which would be imposed by nonstandard protocol modifications.

When processors make use of external peripherals, such as dynamic random access memory (DRAM), another attack vector arises: Due to parasitic effects of the physical construction of modern high-density RAM, it is possible that the hardware cannot guarantee data integrity for all bit patterns. To counteract this, a technique commonly used by memory controllers is the scrambling of data to gain an effectively bias-free bitstream on the RAM chip. With our fifth contribution, we show how one such scrambling scheme by Intel works in-depth and how scrambled memory can be descrambled to reveal the original memory content. In the field of forensics, this is highly relevant: When physical memory acquisition, for example by cold-boot attacks, is used to capture a memory image, descrambling of that image is required before it can be analyzed meaningfully. We furthermore discuss how knowledge about scrambler-internal workings may open up possibilities for an attacker to deliberately cause disturbances in RAM.

Zusammenfassung

Die Sicherheit eingebetteter Systeme wird üblicherweise unter einem äußerst starken Angreifermodell evaluiert: Es wird ein Angreifer angenommen, der sowohl physischen Zugang zu der Hardware und darüber hinaus ein unbegrenztes Kontingent an Zeit hat, um das System zu brechen. Solch ein Szenario stellt für die defensive Seite eine große Herausforderung dar. Diese Arbeit studiert die Härtung von eingebetteten Systemen gegen solche Angreifer. Sie kombiniert mehrere Beiträge aus dem Bereich Sicherheit eingebetteter Systeme; im Kernteil untersuchen wir ressourceneffiziente Methoden, um die Resistenz dieser Systeme gegen physische Angriffe zu härten. Diese sind auch auf stark ressourcenbeschränkten Mikrocontrollern einsetzbar und beeinträchtigen in der Regel das Laufzeitverhalten nur minimal, erhöhen aber nicht die Kosten der Hardware selbst.

Angriffe, wie die *Differential Power Analysis* erlauben es Angreifern, durch statistische Methoden kleinste datenabhängige Variationen im Stromverbrauch von Mikrocontrollern auszunutzen. Um solche Angriffe zu erschweren, müssen geeignete Gegenmaßnahmen angewendet werden. Diese Arbeit stellt eine solche Technik, bei der es sich um eine reine Softwaregegenmaßnahme handelt, vor. Durch Rekompilierung von Binärcode zur Laufzeit erreichen wir Polymorphie der unvermeidlichen Seitenkanalemissionen. Dieser Ansatz ist üblicherweise auf kleinsten Controllern schwierig implementierbar, da er ausgesprochen rechenzeitintensiv ist. Durch geschickte Vorausberechnung bestimmter Teilaspekte erreichen wir dennoch hohe Laufzeitperformance und können so auch für kleinste Mikrocontroller wirksame DPA-Gegenmaßnahmen anbieten.

Nicht nur variable Stromaufnahme, sondern auch winzige Unterschiede in der Laufzeit selbst können von Angreifern mit einfachen Mitteln ausgenutzt werden. Da der angenommene Angreifer physische Kontrolle über das Ziel besitzt, ist es ihm möglich, taktzyklenakkurate Messungen des Controllers vorzunehmen. Code, der zur Laufzeit datenabhängig um nur einen einzigen Taktzyklus variiert, kann bereits ein Informationsleck bedeuten und zum völligen Verlust der Sicherheit führen. Durch die heutigen, immer komplexer werdenden CPUs, ist eine händische Laufzeitanalyse nicht mehr vernünftig möglich; wir stellen daher in unserem zweiten Beitrag einen zyklenakkuraten Cortex-M Prozessorkern-Emulator vor, der solche Seitenkanalangriffe früh im Softwarelebenszyklus detektieren kann. Er kann benutzt werden, um sicherheitskritische Funktionen von Firmware direkt nach jedem Kompiliervorgang auf datenabhängige Laufzeitunterschiede zu testen und dem Entwickler frühzeitig Diagnoseinformationen zur Problembehebung bereitzustellen.

Wie komplex moderne Mikrocontroller geworden sind, zeigen wir in unserem dritten Beitrag, der die Sicherheit eingebetteter Systeme aus einem anderen Blickwinkel betrachtet: Wir zeigen, wie Peripherie, die mittlerweile als gängige Dreingabe auf Mikrocontrollern präsent ist, missbraucht werden kann. Konkret geht es um Funktionalität, die dazu gedacht ist unerwünschte, parasitäre, elektromagnetische Emissionen zu unterdrücken. Wir verwenden ebendiese Einheit jedoch nicht, um Emissionen zu unterdrücken, sondern modulieren die Abstrahlung und erzeugen damit einen verdeckten Kanal. Dieser verfügt zwar über eine relativ geringe Bandbreite, kann aber von einem Angreifer dazu verwendet werden, kurze Nachrichten wie beispielsweise kryptografische Schlüssel zu übermitteln.

Ein gängiger Ratschlag, um eine Härtung gegen physische Angriffe zu erreichen, ist der Einsatz von Hardwaresicherheitsmodulen (HSM). Diese bringen bereits in Hardware implementierte Gegenmaßnahmen gegen gängige physische Angriffe mit. In der Literatur wird leider häufig nicht behandelt, dass die von diesen HSMs angebotenen Schnittstellen hochgradig proprietär sind. Insofern ist eine Integration in bereits bestehende, spezifizierte Protokolle, oft schwierig bis unmöglich. Wie eine solche Integration doch gelingen kann, zeigen wir in unserem vierten Beitrag. Hierbei integrieren wir verschiedene proprietäre HSMs in den standardisierten Handshake des bekannten *Transport Layer Security* (TLS) Protokolls. Dies erreichen wir ohne inkompatible Veränderungen an TLS selbst, sondern durch geschicktes Nutzen bestimmter Freiheitsgrade, die von TLS angeboten werden.

Zuletzt betrachten wir externe Peripherie, welche oft an eingebettete Systeme angeschlossen wird. Hierfür nehmen wir DRAM als Beispiel; bestimmte RAM-Generationen haben durch die hohe Integrationsdichte das Problem, dass nicht alle Bitmuster zuverlässig gespeichert werden können. Deshalb kommt beispielsweise bei Intel-Prozessoren innerhalb des DDR3 Speichercontrollers ein so genannter *Scrambler* zum Einsatz, mit dem wir uns in unserem fünften Beitrag beschäftigen. Dieser beaufschlagt alle Speichermuster mit einem pseudozufälligen Datenstrom. Für forensische Zwecke ist daher das Speicherabbild, das beispielsweise im Rahmen eines Kaltstartangriffs gewonnen werden kann, unbrauchbar. Wir zeigen wie das spezielle Verfahren von Intel funktioniert, wie ein so zerhacktes Abbild wieder zu seinem Urbild konvertiert werden kann und was die Implikationen solcher zusätzlicher, transparenter Technologie sein können.

Acknowledgment

That I arrived where I am now is largely due to my mentor and supervisor Prof. Felix Freiling, who gave me the guidance and freedom to pursue fascinating topics. Navigating through the shoals of science would have been impossible without the orientation you provided. Thank you for your guidance, support, and patience over the last four years.

The person who, many years ago, sparked my interest in cryptography and security is Prof. Falko Dressler. I had the luck to experience his passionate and inspiring way of teaching as a college student. It fills me with great pride that, after all of this time, that spark is still alive and well and ultimately lead to the creation of this work. Thank you for igniting my passion for cryptography and agreeing to review my work.

In the process of writing this thesis I have met many brilliant people at the Department of Computer Science 1 at FAU. Working together with Felix, Sebastian, Michael, and Andreas on our papers has been both pleasant and productive — it made me appreciate what effort goes into scientific writing. Many others at i1 have been involved with my work in one way or the other, however. I would like to thank especially Andreas, Chris, Johannes G., Johannes S., Lena, Michael, Mykola, Sven and Tilo for peer-reviewing my thesis and giving me invaluable feedback on it. We had great times together not only at work but also in private, and I consider myself lucky to have you as my friends. Meeting such bright, open-minded, welcoming and curious people has broadened my horizon tremendously and has been a truly humbling experience. Thank you all for a great time at i1.

Writing this thesis paved the way for me to get incredible insights into the area of computer science which I find most fascinating. This opportunity would never have been possible without the support of my employer, Bosch Software Innovations, and Bosch Smart Home. I especially thank Thomas Schmidt, who believed that I would be qualified to become a security specialist at Robert Bosch and who therefore recommended me as a Ph.D. candidate. It has been a fantastic and wonderful ride so far. I am proud, happy and grateful to be able to do the things which I love — designing and evaluating security systems — for a living.

Lastly, I would like to thank my wonderful wife Julia for her encouragement, patience, and support over the years. Many close friends also played a significant role in getting to this point; Severin, Andreas, Tobias, Alexander — you all know exactly about the ecstatic ups and crushing downs that the life of a Ph.D. candidate brings with it. Thank you for sharing your unique perspective with me and thank you for being the amazing friends you are.

Chapter 1

Introduction

No different from all other disciplines of engineering, in the subject of IT security we also need to make many reasonable compromises. In the design of systems, security is never the only criterion, but we need to have an eye on other parameters such as cost, performance, complexity and maintainability as well. The approach a security engineer takes to build a *secure* system is no different from that of a building engineer who wants to construct a *stable* structure. Both terms are relatively vague and depend on the exact circumstances in which they appear.

How secure does a *high*-security system need to be? How structurally sound does a *stable* structure need to be? It all depends on the context. While there certainly is consensus for some unambiguous cases, most real-world systems are not easily pigeonholed. A connection over which a bank wire transfer is conducted usually has higher security requirements than a connection which serves a public weather report. The tricky part, however, in this determination of requirements is *quantifying* the security level that is appropriate to fulfill the demands of a particular application.

1.1 Security Quantification

The estimation of the security of algorithms like the *Advanced Encryption Standard* (AES; National Institute of Standards and Technology 2001) or the Rivest, Shamir and Adleman cryptosystem (RSA; Rivest, Shamir, and Adleman 1978) is done scientifically: The measure which is used to indicate the level of security is *bits of security.* What this means is that an algorithm that provides n bits of security is as secure as an ideal block cipher with a key length of n bits. The only feasible attack against such a cipher is an *exhaustive search* of the key space. Because it is ideal, there are no algorithmic shortcuts to speed up the attack by definition. An attacker, therefore, needs on average 2^{n-1} attempts to guess the correct key.

An ideal 128-bit block cipher provides exactly 128 bits of security. The popular AES-128 algorithm comes rather close; given the currently most advanced available cryptanalysis of Bogdanov, Khovratovich, and Rechberger (2011), its security level can be estimated to be around 126 bits. For an asymmetric cryptosystem like RSA, the estimation is slightly more complicated because the algorithms which are used to break the system — for RSA, this means factorization of the public modulus n — are much more sophisticated than simple brute force search over a linear key space. However, this has also been extensively studied, and recommendations which factor in the specific difficulty given the best known algorithms have emerged. As an example, the National Institute of Standards and Technology (NIST) recommends that an RSA modulus n of length 3072 bits would yield the equivalent of about 128 bits of security (Barker 2016).

While this may sound like quantifying security is rather straightforward, this feeling is deceptive. Cryptography in many cases is counter-intuitive, and it is hard to tell apart secure from insecure cryptographic constructions. This difficulty becomes alarmingly obvious when we do not simply look at perfect algorithmic black boxes but observe actual instances of hardware which perform a particular cryptographic computation.

To put this into context, we explore a tangent and take a few steps back in time: About eighty years ago, Alan Turing pioneered the field of computer science with the invention of his Turing machine (Copeland 2010). His brilliant and groundbreaking invention, the so-called *Bombe* computer, was used by the British to decode encrypted transmissions of the Axis powers during World War II. For a device like this machine, the physical side effects were palpable: Since the computer's implementation is of electromechanical nature, it was noisy when performing its decrypting computations. The emitted noises naturally corresponded to specific elements of the individual calculation steps. So while a Turing machine in its perfect mathematical form only performs its intended computation on the given tape, the real-world instance had significant real-world side effects which accompanied every part of the computational journey (ibid.).

Why would this digression be of any relevance to modern computing today? The basic fact of the matter is: even our most sophisticated microcomputers today, built solely on semiconductors and without any mechanical parts at all, are — essentially — glorified Turing machines. Moreover, just as their mechanical ancestors, they also exhibit similar side effects along every performed computation. While we do not typically associate modern computers with acoustic noise emission, other types of externally measurable parameters like consumed power, emitted electromagnetic interference or time of computation are very likely to vary in dependence on the processed data.

These variances are certainly minuscule in magnitude and happen at extraordinarily fast speeds of modern computers. While it, therefore, may seem unlikely that they are exploitable, this is what the field of *Side Channel Analysis* (SCA) has shown to be possible. Different emission patterns during execution of machine code are monitored externally, and the eavesdropping attacker tries to infer secret values from these patterns.

Naturally, when investigation about these side channels began with the work of Kocher (1996), the instruments used to eavesdrop on electronic circuitry were sophisticated, leading edge and expensive. The initial demonstration of feasibility happened in a controlled laboratory setting. An oscilloscope was usually directly coupled to the target circuitry to precisely measure timings or the current consumed by the microchip. Nevertheless, the threat that real-world systems are attacked by SCA should not be underestimated.

Genkin, Pipman, and Tromer (2014) show impressively how far exploitation of side channel leakage has come not even 20 years after the work of Kocher (1996). They describe a remarkable new way of picking up side channel emission without having to rely on wire-bound current measurements. Instead, they exploit the fact that the potential of the computer chassis relative to mains ground shows data dependent fluctuations during a running computation. This insight can greatly influence the attack scenario: An adversary who connects measurement equipment to a target PC immediately and obviously raises suspicion. However, it is common for laptop owners to connect their PC to external video displays such as a video projector during a presentation. The shield of the video port is typically directly connected to chassis ground. If the adversary had therefore prepared the projector beforehand, she could be able to attack private key material that is handled by the victim computer during a cryptographic signing process.

While exploring this fascinating new side channel pickup, Genkin, Pipman, and Tromer (2014) tried to find the limits at which key extraction would not be feasible anymore. For this, they set up one experiment in which they used a human body as a pickup: They measured the electric potential of a person who was touching the victim's computer with one hand. Quite surprisingly, modern SCA makes it possible to extract private key material even in this harsh, noisy environment.

Finally, in light of the brief historical prelude about noisy Turing machines, the work of Genkin, Shamir, and Tromer (2014) shows that indeed even on modern computers, *acoustic* leakage can be sufficiently exploited to gain knowledge of secret keys handled during certain cryptographic computations. The implementer of the cryptographic software would need to take special precautions to avoid such treacherous acoustic leakage. The stunning fact about this is that the audible range, with a maximum frequency at maybe 25 kHz, is magnitudes below the frequencies at which computers operate — typically in the Gigahertz range. It only seems reasonable to assume that the leakage would therefore not be exploitable, but Genkin, Shamir, and Tromer (ibid.) demonstrate that the opposite is true.

In a later work, Genkin, Pachmanov, Pipman, Tromer, and Yarom (2016) show that such a low-bandwidth attack does not only affect the audible frequency spectrum but that their results transfer equally well to low-sampling electromagnetic pickups. For demonstration, they showcase wireless extraction of cryptographic material out of a modern smartphone. Their sampling of the signal is done using a haphazard, self-wound EM probe in combination with a commercial-off-the-shelf sound card.

What this illustrates is that even though it may be possible to estimate the level of security of a basic building block, determining the degree of security of a practical instance remains challenging. We need to scrutinize every operation intensely under the actual hardware constraints to avoid generating unwanted data leakage.

On top of the difficulty of avoiding side channel leakage in the implementation of cryptographic primitives comes the challenge to combine many such primitives into cryptographic protocols. For the popular transport layer security (TLS) protocol, for example, there are dozens of different layers involved: For instance, key exchange is performed using asymmetric cryptography after which a derivation function turns the result into a symmetric master secret. That derivation function, in turn, relies on cryptographic hash functions. Signatures based on asymmetric cryptography are used to authenticate the exchanged keys, and the signed data needs to be pre-processed by padding functions. Transmitted data is authenticated and encrypted. It is not surprising that in such a complex real-world protocol, numerous flaws have been discovered over the years (Adrian et al. 2015; Al Fardan and Paterson 2013; Aviram et al. 2016; Beurdouche et al. 2015; Duong and Rizzo 2011; Möller, Duong, and Kotowicz 2014).

Many of these flaws exploit a combination of unfortunate cryptographic design decisions in conjunction with advances in super scale computing. Both the Logjam attack of Adrian et al. (2015), as well as the most recent DROWN attack of Aviram et al. (2016), use the capabilities of current cloud service providers to compromise the security of TLS in various scenarios. These attacks illustrate what kind of a moving target IT security is. A security estimate that would today be considered reasonable might, a decade in the future, become an actual problem when solving the underlying problem becomes computationally feasible.

Many of these protocol weaknesses likely are attributable to accidental mistakes. Unfortunately, in the world we live in today, it is far from clear whether powerful state-level actors such as the NSA are trying to introduce such weaknesses deliberately. Koblitz and Menezes (2015), two luminaries in the world of cryptography, provide an excellent interpretation of the Edward Snowden revelations. They describe possibilities of how malicious organizations could influence standardization of cryptographic protocols to introduce deliberate weaknesses or back doors.

The task of a security engineer is therefore much more challenging than the sole mathematical analysis of the used cryptography. The effects of real world hardware, the constitution of cryptographic compound protocols and the possibility of adversaries to profoundly influence design decisions of such protocols are factors which complicate the matter significantly.

1.2 Embedded Devices as a Target

We need to consider all factors mentioned above when trying to decide how secure a system needs to be, what quantifiable measure of security is appropriate and against whom we need to defend our system. The number of systems which perform communication tasks is steadily rising because of ubiquitous computing which is emerging in the context of the Internet of Things (IoT). While this term is often used as a buzzword — and, to some extent, certainly is — the devices that belong to that category are currently making their way into the consumer market.

New installed electricity meters are now almost exclusively so-called *smart meters*. Concretely, this means these devices perform communication with a backend to transmit their meter readings wirelessly. These readings need to be protected against modification, which is why asymmetric cryptography is used to sign the measurements.

In the *smart home* market, formerly only of interest to the enthusiasts, we now see devices which are inexpensive enough to attract the average consumer. These devices include switches, outlets and electronic locks, lighting applications as well as indoor and outdoor cameras, smoke detectors and motion detectors. To be able to fulfill reasonable use-cases, many of these devices are not only interconnected among each other, but also perform communication with backend servers. This communication also needs to be secured against eavesdropping and manipulation to avoid adversaries remotely doing electronic lock picking.

Car insurance companies have begun to offer their customers lower rates if they in exchange install black boxes in their vehicles. While this functionality is marketed as a feature that rewards people with defensive driving habits, the insurance company apparently also wants to determine if the insurant was — even partly — at fault as soon as an accident happens. It is only sensible to assume that these black boxes act as velocity and acceleration recorders which can pinpoint the speed of a car at the point of impact. Naturally, these readings would need to be protected against unauthorized modification on one side and encrypted against eavesdropping on the other side.

While all these examples are spread over multiple domains, what they all have in common is their attacker model: All of these embedded systems are small computers which are in permanent possession of the customer — who could potentially also be the adversary. In the area of IT security, this results in the unusually strong attacker model. We try to fend off an adversary in physical possession of the device who has an almost unlimited amount of time to break it.

Having physical control over an embedded system is a tremendous advantage when trying to break into a system. Trivial attacks include attaching a programming adapter to the system and try to force the target into debugging mode. While such development interfaces are usually locked in a production device, there still might be ways to glitch the device into a state in which it enters the debugging mode even though it has officially been disabled. An attacker could probe for such glitches by various means such as applying under- or overvoltage to the chip for extremely short periods of time. Another approach would be to put fast transients on the primary clock source to get the chip to trip up during the instruction decoding phase. These are all possibilities only an attacker in physical proximity of the device could exploit.

Passive physical attacks are not less dangerous, however. The side effects that microcontrollers naturally exhibit may lead to code inadvertently spilling its secrets through side channels such as power emission or electromagnetic emission. An adversary can create perfect laboratory conditions to minimize the effects of noise or jitter on the system and can perform accurate measurements efficiently. Such attacks have become even simpler since O'Flynn and Chen (2014) published the full details and source code of their ChipWhisperer platform. It is a versatile, low-cost device that can perform sophisticated attacks such as *correlation power analysis* (CPA) with daunting precision. We expect therefore the amount of attacks on consumer-grade electronics to see a sharp rise in the future because conducting attacks on such devices has become much less challenging.

1.3 Contributions

For the manufacturer of equipment, fulfilling the requirement of using inexpensive hardware but still get the required level of protection against such sophisticated attacks is challenging. The measures which impose the lowest additional cost on a maybe already developed embedded system is one that can be realized in software. This is why we studied software-only countermeasures against power analysis, and we present our approach in Chap. 2. It extracts entropy from chip-internal sources and uses that to recompile code in a fashion that ultimately leads to the correct result, but under greatly varying emission patterns.

Another point of interest to an attacker is passively monitoring the timing of functions. As you can imagine, an attacker who has physical access to a device can use hardware like oscilloscopes to perform measurements with great accuracy. Therefore, even the slightest differences in timing that a system may exhibit can be exploited by an attacker. This danger is something that many standard implementations today do not even consider since they are optimized for performance and not for constant runtime. A typical example for this is the `memcmp` routine, which compares two chunks of memory. It usually performs a lazy abort as soon as the first difference in the two buffers under examination is encountered. If this `memcmp` call is called by a function which compares given user input to a fixed password, this optimization immediately becomes a security issue: the attacker can then crack the checking function character by character just as a lock picker would pick a lock pin by pin. For each position in the buffer, there is exactly one option that deviates in runtime from the rest — and if only by a single clock cycle — that indicates that the byte was correct, and the attacker can continue guessing the remaining bytes.

While it is generally not possible to automatically create code that performs a given task in constant runtime, we wanted to give the user at least a hands-on tool to be able to *detect* such possibly critical timing leaks. Since the ARM Cortex-M platform is overwhelmingly popular, we chose to develop a Cortex-M core emulator with the single objective of emulating the timing behavior as closely as possible down to clock cycle accuracy. We present this emulator in Chap. 3. It allows a user to include simulation runs in a build pipeline to detect problems early on. Such problems can appear, for example, if a more recent compiler version detects a corner case in which it can apply a particular optimization. With our approach, we can pinpoint such cases to enable the developer to remedy the issue.

The modern microcontrollers which are on the market today have a vast amount of peripherals on board. This feature-packing happens because it is often less costly for the vendor to produce a standard core that all processors of the family have in common. How this can enable new methods of attack for an attacker who can poison the firmware or supply chain is something that we take a look at in Chap. 4. Concretely, we take the offensive side and abuse facilities which are present on modern microcontrollers and which are meant to *reduce* electromagnetic interference to *cause* electromagnetic interference deliberately. The way in which we modulate these hardware peripherals allows us to transmit information. An astounding fact about this is that an adversary may create a covert channel that can be detected in the radio frequency domain — even when the compromised system was never intended to perform any RF communication in the first place. This could give an attacker the possibility of planting back doors or bugs in systems which would otherwise be inaccessible to her.

Since there is market demand for it, not only have general-purpose microcontrollers gained popularity, but specialized security controllers have also become more popular. Such controllers usually bring hardware countermeasures that general purpose microcontrollers do not have. As such, they can be regarded to provide a higher level of security. The problem with integration of such specialized controllers, however, is that they dictate the exact fashion in which a specific cryptographic computation is performed and usually leave little flexibility. For the system designer, this creates a dilemma if the protocol which she wants to implement also dictates a specific API. To create a functional system, high-security companion chips are often discarded because they are deemed incompatible with the pre-existing API definition. This is something we hint to in Chap. 5. However, we show a solution for integration of such proprietary high-security modules into pre-existing protocols. To demonstrate that our approach is feasible, we integrate various hardware security modules into the standard handshake of the TLS protocol without having to alter the structure of TLS itself.

In more sophisticated controller systems, it is not possible to integrate all peripherals on a single System-on-Chip (SoC). Instead, it is common to have a small amount of static RAM on the controller, but have external, dynamic RAM, on separate ICs of the embedded system. DRAM, however, is not without its caveats, as we show in Chap. 6. Concretely, we analyze DDR3 DRAM that is widely used in the personal computer domain today and examine how Intel's memory controllers access this DRAM. Since the bit density in modern DRAM chips is extremely high, the chance of disturbance errors due to current fluctuations grows. Intel compensates for this by applying pseudorandom scrambling in the data stream that is transmitted between the memory controller and DRAM chip itself. The impact of our work has two aspects: On one side, we reiterate that simple scrambling — although it was a major obstacle for forensic image acquisition — is not sufficient for cryptographic purposes. More importantly, however, we demonstrate the danger of using hardware which might only then perform its intended function reliably when the underlying hardware constraints are not exploited.

Chapters 3 (Bauer and Freiling 2016), 4 (Bauer et al. 2016a), 5 (Bauer and Freiling 2015) and 6 (Bauer, Gruhn, and Freiling 2016) are all based on previously published, peer-reviewed work. In all four of these publications, the main contributions and majority of the writing were carried out by Johannes Bauer (clarification in accordance with FPromO Tech, § 10 Abs. 3 Satz 2).

1.4 Outlook

What we want to achieve with our work is twofold: On one hand, we illustrate how susceptible modern general-purpose microcontrollers are to attacks which have come within the reach of even laymen. While power analysis was something that only a few experts could perform, after the openness with which researchers like O'Flynn and Chen (2014, 2015) approached the topic, many of these attacks can now be conducted by literally pointing and clicking. Similar reasoning applies to hardware glitching or timing analysis, which both have become impressively effortless with modern low-cost attack hardware.

The other goal we are trying to reach is to broaden the spectrum of possible countermeasures to these attacks on the low-cost side of the market. While some argue that dedicated hardware can and should be used to protect secrets, we disagree: Especially in the IoT context, many devices store secrets that are particularly relevant for privacy reasons. The temptation of competing vendors to simply omit security features because they are deemed too costly is very real. In the sense of scaling, software countermeasures are inexpensive because once they have been incorporated in code, no additional fee comes with each sold unit. Unfortunately, many companies do not consider the privacy of their customers especially valuable and therefore try to avoid paying an extra price per unit for improved security. In our opinion, it is better to offer these people inexpensive software alternatives; while these certainly do provide a level of security that is less than that of dedicated hardware, they may be the right fit for a particular application nevertheless.

We believe that our work gives interesting insights and ideas which help make embedded systems of the low-cost market more resilient against capable adversaries even in the face of the challenging physical attacker model.

Chapter 2

Power Analysis

Power analysis (PA) has, after around 20 years of research, arrived at a point where it can be conducted not only by experts with intricate knowledge of the subject but also by laymen who know how to operate the available tools properly. This transition severely impacts systems in which typically no countermeasures were applied in the past. Due to the cost-sensitivity of these applications, however, hardware countermeasures are often not a viable option. We present a software-only power analysis countermeasure which achieves Boolean masking by using efficient binary recompilation. The necessary entropy is extracted from MCU-internal processes with no need for a hardware RNG. We implement and evaluate it on the popular ARM Cortex-M0/3/4 architectures, using its specifics like the built-in barrel shifter of the M3/M4 to our advantage. The result is an overall performance penalty factor of 2.2 which reduces the overhead considerably compared to other software-only solutions of which we are aware. Our approach shows that even the smallest Cortex-M series can be hardened effectively against non-invasive physical attacks.

2.1 Introduction

The most basic building block that made the modern computer possible is the transistor. In modern processors, transistors are interconnected to form functional logic units, the *gates*, which itself are interconnected to form core processor components such as the arithmetic-logic unit. Millions of transistors are used in modern processors, and the component density on these devices is extremely high. The third generation Intel Core processors have around 1.4 billion transistors in a chip area of about 160mm^2 (Intel Inc. 2012b). Effectively, this means on average there are almost 9 million transistors per square millimeter of space. These transistors are optimized to produce only a minuscule amount of power dissipation so that such high component density is supportable. In modern devices, this has been achieved remarkably well. Only active components — the ones that may not be idle at a given point in time — on a chip dissipate power, and static losses are reduced to a point where they are virtually negligible.

There are two aspects of looking at computations performed by machines: The first is the algorithmic, mathematical, algebraic representation. Machines viewed like this appear to be perfect black boxes with well-defined inputs and outputs. They perform their duty and only that. When it comes to the real implementation of such a machine, however, the parasitic effects of heavily optimized hardware start to become noticeable: in practice, the machine exhibits side effects such as power fluctuation, noise emission, electromagnetic emission or variable execution time — usually with dependencies on the calculated data. When these side effects cause data leakage to the outside, we refer to them as *side channels*. Cryptographic algorithms have become secure from an algorithmic standpoint; 16 years after its inception, the best known attack against the block cipher AES, presented by Bogdanov, Khovratovich, and Rechberger (2011), only manages to reduce its security by a factor of four. To attack such secure algorithms, side channel analysis, therefore, is often the only viable option. Attackers shift their attention from attacks on the theoretical implementation to attacks on practical instances of such basic building blocks. In unison with the advent of cryptographically secure algorithms, the relevance, and importance of side channels have equally risen as well.

Microcontrollers with impressive computational capabilities have become readily available in the last few years. The popular 32-bit ARM Cortex-M architecture, for example, offers a value line, the Cortex-M0, which is aggressively marketed by one semiconductor manufacturer with the tagline “32 bits for 32 cents” (STMicroelectronics N.V. 2015b). In this segment, these devices offer 32-bit performance while competing with 8-bit microcontrollers price-wise. It goes without saying that these devices are more than capable to run state-of-the-art cryptographic algorithms like AES-128 or SHA-256.

If, however, side channels are present in the implementation of this cryptographic building blocks, *side channel attacks* (SCA) can be used to infer secret bits. One possible instance of SCA is *differential power analysis* (DPA) which exploits the fact that computational hardware exhibits parasitic, data-dependent power emission. When Kocher (1996) pioneered the field of SCA, conducting these attacks was only feasible for attackers with exceptionally sophisticated cross-domain knowledge.

Today, however, the open-source ChipWhisperer platform created by O’Flynn and Chen (2014) is readily available and makes for an affordable, low-noise data acquisition system. Open-source software is included which can perform advanced attacks (O’Flynn and Chen 2015) on the captured data even by non-experts. All hardware design documents such as schematics have been published by them, making custom adaptations and modifications of the hardware easily possible. It can be reasonably expected that such a groundbreaking change leads to an avalanche effect in the attacks we see in the upcoming years. We believe that the full potential of SCA, therefore, is not focused only on expensive hardware but also on low-cost systems which contain secrets of lesser value.

To protect against SCA, both *masking* and *hiding* are common countermeasures. Masking refers to the distortion of intermediate values to obfuscate the device emission characteristic while hiding describes an obfuscation of the control flow of the computed algorithm. Both operations can be realized in either hardware or software. If cost is no issue, hardware solutions are usually preferable, because they can give a higher security margin than software-only masking or hiding techniques. For the low-cost market, however, software-only solutions are often preferred because they only negatively impact runtime performance and code size. In this chapter, we focus on the Cortex-M series and explore the possibilities of concrete software-only DPA countermeasures.

The idea of our approach is to use runtime binary recompilation to randomize invariants of executed instructions. By abstracting away from the actual assembly code and viewing it as a pure bitstream, we make the runtime recompiler instruction-agnostic. This allows for efficient and effective trace randomization. Our trick is to precalculate most operations beforehand during compile time and generate a highly simple bytecode stream which performs the binary code transformations. This bytecode stream is then interpreted at runtime by a minimalistic virtual machine (VM) before every call to the cryptographic routine.

2.1.1 Related Work

When side channel analysis first aimed to break implementations of cryptographic algorithms, the focus was on high-level timing analysis. Kocher (1995, 1996) analyzed the implementation of the modular exponentiation operation which is at the heart of any Diffie-Hellman or RSA computation (Diffie and Hellman 1976; Rivest, Shamir, and Adleman 1983). He measured differences the computation time the system took with specially crafted input data and was able to recover the full private keys from those measurements. Previous to the work of Kocher (1996) it was widely believed that potential leakage would be insufficient for practical attacks.

With his seminal work, Kocher (ibid.) therefore basically founded the field of modern side-channel analysis. Kocher, Jaffe, and Jun (1999) later extended their attacks to include differential power analysis (DPA). They demonstrate their approach to be feasible by successfully attacking concrete implementations of RSA and DES. The attacks they describe are versatile and apply to nearly all computational equipment which exhibits parasitic, data-dependent side effects.

Shortly after, Coron (1999) published multiple algorithmic blinding countermeasures that aimed at protecting against DPA. Clavier, Coron, and Dabbous (2000) published a statistical approach to overcome insertion of the random dummy instructions within a cryptographic computation. This defensive countermeasure is known as *Random Process Interrupts*. They acknowledge that such countermeasures increase the difficulty of an attack considerably, but not to the point of computational infeasibility.

While many published DPA attacks initially used direct measurements of consumed device current, Gandolfi, Mourtel, and Olivier (2001) looked into the feasibility of key recovery by analyzing the electromagnetic emission of a device. They published attacks using *simple electromagnetic analysis* (SEPA) and *differential electromagnetic analysis* (DEPA) — the equivalent of SPA and DPA in the electromagnetic domain. The same year Quisquater and Samyde (2001) also published their work on electromagnetic (EM) side channels; included in their work are also directions on countermeasures that can be used to defend against DEPA. Even though EM-analysis is in some ways more challenging for an attacker to perform, Agrawal et al. (2003) showed that a successful EM analysis attack could be used in certain cases where a DPA attack would be insufficient.

When designing new block ciphers, one important aspect that needs consideration is the general susceptibility of the algorithm to side channel attacks. During the AES competition, Daemen and Rijmen (1999) made an analysis of the proposed candidates and in their analysis distinguished between arithmetic and Boolean operations that are used within ciphers. It is their conclusion that arithmetic operations, in general, are harder to defend against power analysis attacks compared to their Boolean counterparts.

Coron and Goubin (2000) elaborate on the difficulty of effective arithmetic and Boolean masking approaches, in particular, a generalized scheme described by Messerges (2001) to protect AES finalists against SCA. Both papers are helpful in understanding potential SCA countermeasures and how they can fail in practice.

Tillich and Großschädl (2007) present well-designed hardware countermeasures by extending the instruction set of a SPARC LEON2 by instructions which are used for masking. Since a sole hardware solution is expensive regarding the number of used gates, they combine their modification with software countermeasures to significantly reduce the complexity of the device.

A comprehensive reference on how power analysis can be practically conducted is given by Mangard, E. Oswald, and Popp (2007). They describe the theoretical background behind power analysis and also highlight countermeasures that can be used to defend systems.

Eisenbarth et al. (2008) show a real-world application of DPA in which they broke the keyless entry system KeeLoq. It is widespread in the automotive market and utilizes a rolling code which a nonlinear feedback shift register generates. In their work, they developed a specialized hardware attack that required only 30 power traces of the device under test to break the key.

D. Oswald and Paar (2011) performed a successful DPA attack on Mifare DESFire MF3ICD40. DESFire is a popular, contactless smart card with dedicated hardware countermeasures. The effort to break it was quite significant and a lot of data (about 250000 traces) was required to perform key recovery, but they were ultimately successful, underlining how difficult it is to thwart DPA attacks even by sophisticated hardware countermeasures.

Recently, literature also considers sole software countermeasures such as the one we present here. For example, Agosta, Barenghi, and Pelosi (2012) describe an approach that generates semantically equivalent versions of code with different power emanation characteristic to counteract power analysis. They recompile emission-critical code into many semantically equivalent variations called *tiles* and randomly replace these tiles during runtime. This replacement is known as a *morphing action* and has a performance penalty of 90 ms. It is therefore not computationally feasible to morph the code at every call to the encryption routine, so Agosta, Barenghi, and Pelosi (ibid.) only morph their code after 3000 encryption operations, leading to a performance impact factor of 1.20.

Later, Agosta, Barenghi, Pelosi, and Scandale (2015) refined their approach by choosing a precompiled alternative fragment randomly during runtime, rendering it unnecessary to have a writable code segment. The performance impact at runtime for AES-128 encryption with their approach is a factor of about 6.75.

Bayrak et al. (2015) presented an approach for automatic software-only masking countermeasures. It is based on a compiler architecture which detects leaking instructions and masks them to achieve a compromise between the desired security level and runtime performance penalty. While their approach focuses on creating a high-performance statically masked block cipher, we are not aware of any work that uses dynamic, randomized runtime recompilation to achieve the same goal.

Boolean masking is difficult to implement correctly using masking tables, as Tunstall, Whitnall, and E. Oswald (2014) demonstrate. In their work, the authors attack the table generation directly, nullifying their effect during an attack. Such a break of a potential DPA countermeasure underlines the need for a plentiful source of good entropy when attempting to mask power emission. Van Herrewege, van der Leest, et al. (2013) describe means of collecting entropy on devices which do not contain a hardware random number generator. Van Herrewege and Verbauwhede (2014) use the content of uninitialized microcontroller SRAM as seeding entropy to a Keccak-based PRNG. Both works consider the Cortex-M platform.

2.1.2 Contributions

In this chapter, we present a masking technique that uses efficient runtime binary recompilation to randomize invariants of some instruction sequences and randomly schedule the time of their execution. We leverage the offered capabilities of the specific architecture to create efficient, easily recompilable masking countermeasures. One example of such capabilities is the versatile barrel shifter of the Cortex-M3 and M4 variants. Similar to Agosta, Barenghi, and Pelosi (2012), our idea is to pre-calculate which parts of masking transformation opcodes can be randomized and have a minimalistic VM execute these reobfuscations during runtime. More concretely, the contributions we make in this chapter are as follows:

- We present a method of performing runtime binary recompilation which is extremely efficient by doing many time-consuming calculations beforehand at compile time. For an AES-128 encryption operation, our protected variant takes about 2.2 times as long as its unprotected counterpart. This is a substantial improvement to the currently best software-only result we are aware of (factor 6.75 by Agosta, Barenghi, Pelosi, and Scandale (2015)). Note that in contrast to Agosta, Barenghi, and Pelosi (2012), who recommend morphing code every 2000 to 3000 encryption calls and achieve a performance impact factor of 1.2; we recompile our code with *every single invocation* of the encryption operation.
- We utilize specific hardware properties present on the Cortex-M architecture for implementation of Boolean masking transformations. Chip-specific features, namely the internal phase locked loop (PLL) locking times, are used as an entropy source. It can produce 268 bits of unbiased entropy per second, enough to efficiently seed our cryptographically secure pseudo-random number generator (CSPRNG). We

demonstrate that our combined measures yield an effective differential power analysis countermeasure.

2.1.3 Outline

This chapter is structured as follows: In Sect. 2.2 we give the background which is necessary to understand why real-world hardware exhibits data-dependent power emission and how power analysis can exploit this effect. We also highlight some aspects the Cortex-M processor family which we use to strengthen our proposed countermeasures further. Afterward, in Sect. 2.3, we show some concrete instructions or instruction groups which we target within our countermeasures. How we efficiently implement randomization of these instructions at runtime to achieve different emission characteristics is something we show in Sect. 2.4. We finally evaluate our method experimentally in Sect. 2.5 and give concluding remarks in Sect. 2.6.

2.2 Background

To understand the prerequisite that enables power analysis in the first place, we need to take a look at the basic building blocks of modern semiconductors in Sect. 2.2.1. How power analysis makes it possible to infer secret data from power fluctuation measurements is then highlighted in Sect. 2.2.2. Afterward, we show in what ways the Cortex-M family is unique in Sect. 2.2.3 and what possibilities this opens up to use the architecture to our advantage in the implementation of effective PA countermeasures. Then we go into detail about the countermeasures in Sect. 2.3. How we implemented this efficiently on real hardware is something we describe in-depth in Sect. 2.4. To demonstrate that our approach is effective, we give an experimental evaluation in Sect. 2.5 and finally conclude in Sect. 2.6.

2.2.1 Transistors and Switching Loss

In the 1960s manufacturing of field effect transistors (FETs) has become possible on an industrial scale. Since then, they have replaced their bipolar counterparts in almost all semiconductors manufactured today. FETs are electronic devices with three terminals: a *gate*, *source*, and a *drain* connection. A base material, called the *substrate*, is used as a basis for their fabrication. This monocrystalline substrate is *doped*, which means that deliberate impurities are introduced into the substrate to achieve the desired behavior of the semiconductor. Doping is performed to make regions of the substrate either be depleted of electrons (i.e., contain *holes*, known as *p-type* doping) or to have an excess of electrons (i.e., contain *carriers*, also known as *n-type* doping). The physical construction of a transistor is shown in Fig. 2.1. Within the substrate, two islands are cut in which doped material is inserted with a doping type opposite to that of the substrate. These

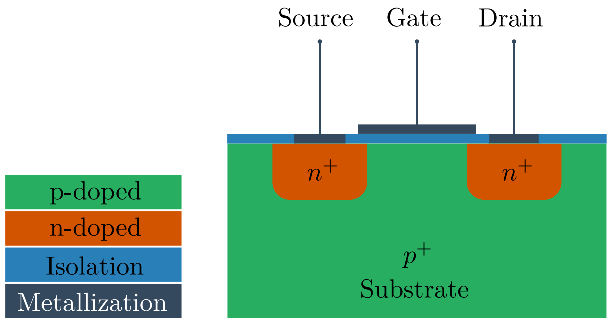


Figure 2.1: N-channel metal-oxide semiconductor field effect transistor

islands form the source and drain. An isolation layer is manufactured on top of both of these two doped islands, and a connection is made on top of this isolation layer. It forms the gate. The exact physical process of how this isolation layer is formed differs across types of FETs. We show the manufacturing process of a MOSFET where an explicit physical isolation layer is used, which usually consists of SiO_2. The transistors which are used in CPUs are constructed differently, but the general principle and resulting consequences stay the same.

As the name suggests and as Shockley (1952) also describe, the mode of operation of FETs utilizes field effects within the transistor. If the substrate is of p-type and source and drain are of n-type the device is called an *N-channel* field effect transistor. Fig. 2.1 shows such an N-channel MOSFET. Characteristic for the N-channel FET is that it becomes conductive across its drain-source region when the potential applied to the gate is positive compared to the potential of the source. Inversely, if the substrate is of n-type and source and drain are of a p-type doped material, the device is called a *P-channel* FET and exhibits the opposite effect. It becomes conductive across its drain-source region when the potential applied to its gate is negative compared to the potential applied to its source.

When a FET is conductive, it is also — analogously to a switch — referred to as being *turned on* while it otherwise is *turned off*. The associated source-drain resistance for the on and off states are called $\mathrm{R}_{\mathrm{DS}_{\mathrm{on}}}$ and $\mathrm{R}_{\mathrm{DS}_{\mathrm{off}}}$. For an ordinary MOSFET such as the popular 2N7002 typical values are $R_{\mathrm{DS}_{\mathrm{on}}} = 2.8\Omega$ and $R_{\mathrm{DS}_{\mathrm{off}}} = 4.8\mathrm{G}\Omega$ — a dynamic range of more than nine orders of magnitude (NXP Semiconductors N.V. 2011).

In complex digital semiconductors such as integrated circuits, FETs are mainly used in a so-called *complementary metal-oxide semiconductor* (CMOS) configuration. Complementary means that every digital output stage is composed of two complementary MOS transistors, one N-channel and one P-channel FET. The gates of both transistors are connected to form the input terminal of the driver stage. Since the switching characteristic of an N-channel and P-channel FET is complementary, this means that exactly one of the two transistors is active in a steady-state with valid digital input voltages applied to the common gate. Valid in this context means that the voltage is outside the so-called *forbidden region*, i.e., the voltage level is either lower than a certain threshold V_{IL} if the

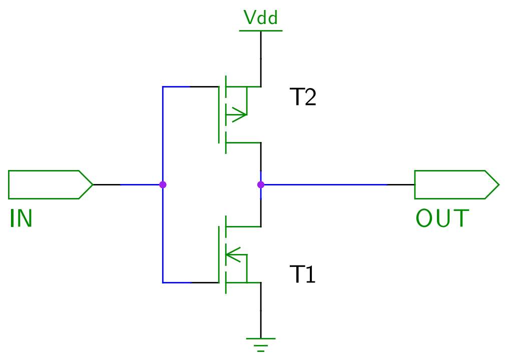


Figure 2.2: Idealized inverting CMOS output stage

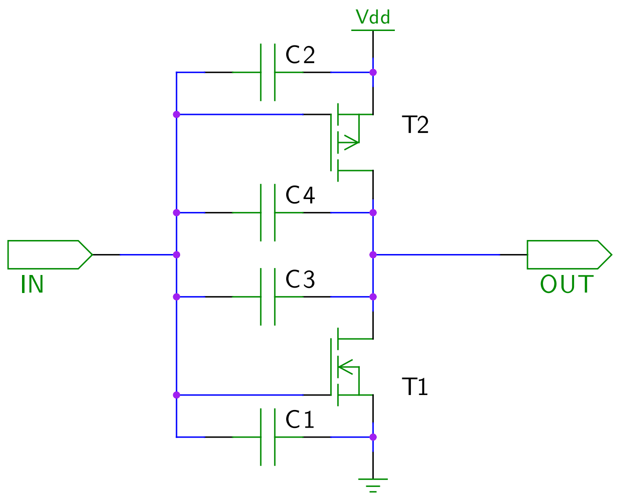


Figure 2.3: CMOS output stage with C_{GS} and C_{GD} modeled

input is a digital LOW or higher than a certain threshold V_{IH} if the input is a digital HIGH .

Modern integrated circuits use different variants of the CMOS technology, but still have the common property that their output stages use a push-pull transistor configuration. The push-pull construction is used to allow fast switching because active drivers cause every change of the output. Consequently, an output stage can both actively source or sink current if either the P-channel or N-channel FET is turned on. Such an output stage driver can be seen in Fig. 2.2. It has the side effect of inverting the input signal because an active N-channel FET (logical HIGH input) switches the output against GND and an active P-channel FET (logical LOW input) switches the output against the positive supply voltage, V_{DD}.

In an idealized device, switching is instantaneous and lossless, i.e., without any power dissipation. Real-world devices, however, exhibit parasitic effects which do cause switching losses. One contributing factor is the parasitic capacitance which forms between the transistor terminals. Fig. 2.3 shows a more accurate model which explicitly models the gate-source capacitances $\mathrm{C_{GS}}$ (C1 and C2) and the gate-drain capacitances $\mathrm{C_{GD}}$ (C3 and C4). While these capacitances do not contribute to energy consumption in a static (non-switching) case, they do cause a current surge when the input signal switches and the capacitances are charged or discharged. Since the energy W_C stored in a capacitor is

$$W_C = \frac{1}{2}CV^2$$

Where C is the capacitance and V is the voltage that the capacitor is charged up to, this means that the associated switching loss W_l is

$$W_l = 2fW_C = fCV^2$$

Another contributing factor to the buffer input capacitance is the *Miller effect*. It was first described by J. M. Miller (1919) and describes the effect that inverting amplifier stages see a perceived increase in input capacitance dependent on the amplifier gain. While the original discovery was revolving about input capacitances of amplifiers built with vacuum tubes, the same principle still applies to modern semiconductors.

In addition to the energy dissipated through the periodic charging and discharging of parasitic FET capacitances, the FETs itself traverse their *linear region* when switching from the off to the on state or vice versa. This means that during the transition both transistors are conductive to a certain extent. Therefore, a current path forms the positive supply voltage over both drain-source paths $\mathrm{R_{DS}}$ to ground.

2.2.2 Power Analysis

As explained in the previous section, the static power consumption of a complementary transistor pair is comparatively small, but switching causes a surge in power consumption. Within integrated circuits, complementary transistor pairs are assembled to form logic gates. Using gates it is possible to build memory cells such as the most basic latch: the RS flip-flop circuit. Such memory cells are used in computers to hold a particular state. One example of this is the accumulator of a processor, which is fundamentally just an array of memory cells.

Breaking such a hardware register down into its constituent components, the implications regarding power consumption become evident: Flipping a bit in a register means that some constituting gates change their output value. Intuitively, the magnitude of the current spike associated with such a data change within latches corresponds to the amount of bits flipped by the operation. This immediately leads to the prerequisite that makes differential power analysis possible: there exists an externally measurable parameter (i.e., the power consumption of the whole device) which correlates with the data that the CPU computes.

Consider, for example, the implementation of a symmetric encryption algorithm such as AES which we view as a black box. The algorithm itself is public knowledge, but the key which parameterizes algorithm is unknown to the attacker. However, the attacker has access to a hardware implementation which performs the computation. With this given hardware, she can encrypt arbitrarily chosen plaintexts while simultaneously measuring the power consumption during this computation. The first thing the attacker would do is record a number of such traces with randomized inputs. She then records the input and output values of the cryptographic computation and stores them along with recorded current consumption over time, called the *power trace*.

In a second step, the algorithm is analyzed. In particular, the attacker chooses a target operation which handles intermediate values of the cryptographic computation. This secret intermediate value needs to have a certain correspondence to parts of the secret key. The attacker then makes a random guess about this intermediate value. Under the assumption that her hypothesis is correct, she estimates for each of the previously computed ciphertexts the amount of leakage that this certain operation (such as an XOR of an S-box output value used within the algorithm) would have created. Then she groups the previously captured traces accordingly, for example into one group where little power fluctuation can be expected and into a second group in which a significant surge would appear if her guess was correct.

With randomly chosen plaintexts and a reasonable cryptographic algorithm, each group roughly contains the same amount of power traces. Both groups are then averaged, and the differential trace between the two averaged groups is calculated.

There are two possible outcomes: Either the guessed intermediate value is right, or it is wrong. If it is wrong, then the calculated estimates for the power consumption are likewise equally incorrect. The grouping, therefore, is wrong in roughly half of the cases. Therefore, both groups have an identical statistical constitution, and thus the differential trace only contains noise but is otherwise flat.

If, however, the attacker's intermediate value hypothesis was correct, then the estimation of flipped bits and consequently of dissipated power also is correct. The grouping is also correct, and one group contains a lot of traces with large power dissipation while the other group contains many traces with comparatively small dissipation. The differential trace, therefore, exhibits a sharp *spike* at the point in time where the attacked operation occurs. This peak reveals to the adversary that her hypothesis was correct, and she, therefore, learned part of the hidden key.

Using this revealed information, the attacker would now proceed to make another guess about a different part of the algorithm — this time integrating the knowledge gained from the first attack into her model. She again groups traces and repeats the process until she has learned the whole key. Similar to lock picking, with this approach, the adversary also leverages the fact that she can break the lock pin-by-pin — or, in our electronic case, break the algorithm byte-by-byte. That is the basic idea behind power analysis, and while more sophisticated statistical methods have since superseded this simple approach, the underlying principle remains the same.

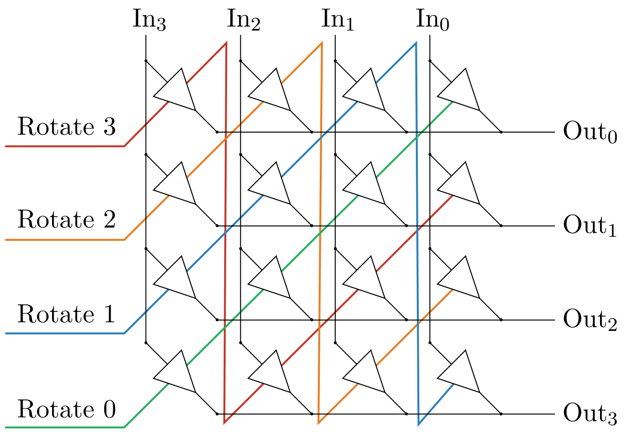


Figure 2.4: Simplified view of a 4 bit barrel shifter construction made from 16 tri-state buffers.

While modeling the hardware behavior might seem to be difficult, it is surprising that even simple emission models are surprisingly effective. One example is the described *Hamming difference model*, e.g., described by Brier, Clavier, and Olivier (2004), which predicts a proportional change in power consumption in dependence on the number of flipped bits. Depending on the sophistication of the implemented algorithm and the measuring equipment, successful attacks can be performed with as little as one single trace (Schuhmacher 2014) for simple targets. For targets which are heavily protected by countermeasures, however, a successful attack might require traces that range well into the hundred thousand, e.g., as shown by D. Oswald and Paar (2011).

2.2.3 Cortex-M Barrel Shifter

A barrel shifter is a programmable digital circuit which can shift or rotate data words in constant time. This means its use for defense against DPA does not create additional security issues such as timing side channels. For a given word width n it consists of an $n \times n$ matrix of tri-state buffers. Such a buffer matrix is shown in Fig. 2.4. It shows how the barrel shifter would be configured to perform a rotation of $0 \dots 3$ bits by enabling exactly one of the four inputs on the left side. In practice, this circuit is slightly more complicated because it can also accommodate for bitwise or arithmetic shifts.

An integral part of the Cortex-M3 and Cortex-M4 architectures is the MCU-internal barrel shifter (ARM Ltd. 2014). Using this barrel shifter is — different from most other assembly dialects — not done via special assembly instructions. Instead, it attaches to several instructions if the encoding is chosen in the so-called 32-bit *wide form*. To make this a bit clearer, consider a 24-bit binary shift left in Intel64 assembly (Intel Inc. 2012a). It is achieved by calling the `shl` opcode, as shown in List. 2.1.

In contrast, on the Cortex-M3 or M4 architecture one could use a `mov` opcode and specify through the mnemonic's encoding that the barrel shifter should simultaneously

```
; rax <<= 24
shl $24, %rax
```

Listing 2.1: Intel64 representation of a 24 bit shift.

```
; r0 = r0 << 24
mov r0, r0, lsl #24
```

Listing 2.2: Thumb-2 representation of a 24 bit shift.

be engaged, as shown in List. 2.2. This connectable barrel shifter is available for many instructions and gives the developer a powerful tool; it is something that we are using to our advantage for masking computational intermediates on the Cortex-M3/M4 variants.

2.3 Code Polymorphism as a DPA Countermeasure

Our DPA protection scheme is based on randomizing the computation immediates and certain instruction order at runtime through dynamic re-compilation of the protected binary. In this section, we go over some examples of what instructions are relevant for this type of polymorphic code transformation. Some explained measures make explicit use of the Cortex-M3/M4 barrel shifter. Note that our enumeration of methods is incomplete; we also use other techniques such as random pre-charging and dummy instructions in our code. Since these have been thoroughly studied and are well-known in literature, for example in the work of Coron and Goubin (2000), we do not repeat them here and instead omit them for brevity.

2.3.1 Displaced Loading

A typical point of vantage which an attacker uses to create an accurate model for a power analysis attack includes assumptions about the implementation under attack. This information, therefore, should be considered public knowledge. An example of this is that an attacker who knows that she is attacking a table-based AES implementation can assume there is an S-box lookup at some point during the computation. With this lookup and the following computations, a data-dependent power emission characteristic can be accurately modeled.

Looking at a typical table-based AES implementation without any PA countermeasures, we experimentally verified our assumption about the origin of loads. For this, we look at the public domain AES implementation of Paulo Barreto. We just focus on AES encryption in the following, but all observations apply to decryption analogously. The example code, which we compiled for the Cortex-M4 with gcc 5.2.0, emitted 60 load instructions. Of those, 38 (63 %) were within the calculation-intensive inner loops of the algorithm, the remaining 22 (37 %) were constant address initializations typically generated at function entry. The inner loop loads are the ones which are of relevance to power emission. Of those, roughly 50 % were lookups of the S-box table while the other 50 % were lookups of the inner state. Of these 38 most interesting load instructions, 34 (89 %) were register indirect loads with displacement.

To mask power emission of loading, any load can be replaced by an arbitrary number of load instructions if the code ultimately performs the correct action. When there is a scratch register available, it is possible to replace one load operation by many load operations which have the same base address, but randomized displacements. These dummy instructions can then be placed alongside and around the target instruction because their actions only affect the scratch register.

```
; r4 = r4[14]
ldrb    r3, [r4, #14]
```

Listing 2.3: Load of a single byte.

```
; r3 = r4[n]
ldrb    r3, [r4, #n]

; r4 = r4[14]
ldrb    r3, [r4, #14]

; r2 = r4[m]
ldrb    r2, [r4, #m]
```

Listing 2.4: Load of three bytes.

Such an example is shown in List. 2.3 and List. 2.4. The left version shows the original variant. On the right, the code that has been post-processed is shown, There, the first `ldrb` instruction loads one byte relative to `r4` with a randomized offset n. The destination is the register `r3`. This is semantically correct because the next `ldrb` operation — the one that performs the actual load — overwrites `r3` with the right result. The third load is again a dummy with randomized displacement m into the spill register `r2`. There are many degrees of freedom when reorganizing load instructions which all lead to different runtime characteristics.

Note that for the code which we show in List. 2.4, the first and last `ldrb` instruction are not actual valid assembly opcodes — they merely demonstrate that the variables n and m are invariant for the correct execution of the code snippet. In reality, there would be hardcoded offsets in these locations replacing n and m. During binary recompilation, we change these hardcoded immediates. We also change the order in which these instructions execute.

2.3.2 Exclusive OR Instruction

Bitwise instructions are common in code which performs cryptographic computations. Therefore, in such code, they are also responsible for a significant amount of side channel leakage and should be given attention when trying to minimize that leakage. Among the bitwise instructions, `eor` the most prominent; it is the Cortex-M mnemonic for the exclusive or (XOR) operation.

Since our aim is to obscure power emanation of bitwise instructions we can, on Cortex-M3 and M4 devices, utilize the barrel shifter to our advantage. We first cause bit flips by rotation of the source register, then perform the bitwise operation with an identically rotated operand and finally rotate the bits back to their original location. Exemplary, code for an XOR operation of two registers is shown in List. 2.5 and its obfuscated counterpart code is presented in List. 2.6. The penalty of this operation is a tripling of both code size (4 bytes in the plain version, 12 bytes in the obfuscated version) and execution speed. Variants of this code could split up the single register rotation (i.e., the first and last instructions) into multiple rotations which add up to the desired amount of rotated bits at the penalty of more and slower code.

Superficially, the whole ordeal does not look like it is doing anything significant except performing the XOR operation on a rotated word. However, the effect becomes apparent when one looks at the modeled power emission of the combined code snippet on a Hamming difference model in dependence of the input operands. In the simple XOR operation, the power emission is proportional to the Hamming weight of the second operand only. In its obfuscated replacement, the sum of leakage depends on both operands.

```
; r0 = r0 ^ r1
eor r0, r0, r1
```

Listing 2.5: Plain XOR

```
; r0 = ror(r0, 16)
mov r0, r0, ror #16

; r0 = r0 ^ ror(r1, 16)
eor r0, r0, r1, ror #16

; r0 = ror(r0, 16)
mov r0, r0, ror #16
```

Listing 2.6: Obfuscated XOR

This effect is shown graphically in Fig. 2.5 through Fig. 2.10. All images show operations of a 9-bit value XOR another 9-bit value, i.e., 512 by 512 pixels. The x axis in both cases stands for values for one operand while the y axis shows all values the second operand may take. The point $(0, 0)$ is at the lower left corner for all figures, and the color indicates the normalized power emission which the modeled XOR operation would cause. For each image, the maximum value in the graphs is, therefore, the maximum bit flip count for

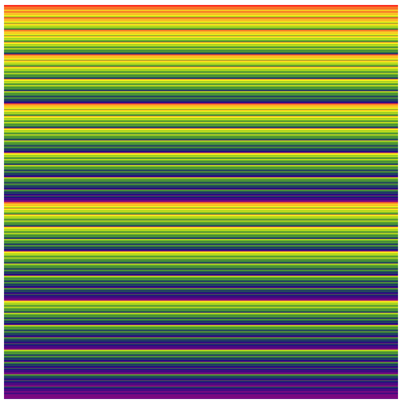


Figure 2.5: Unobfuscated version

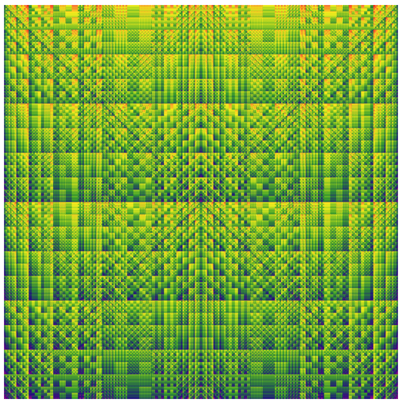


Figure 2.6: 9 bit, obfuscated `ror5`

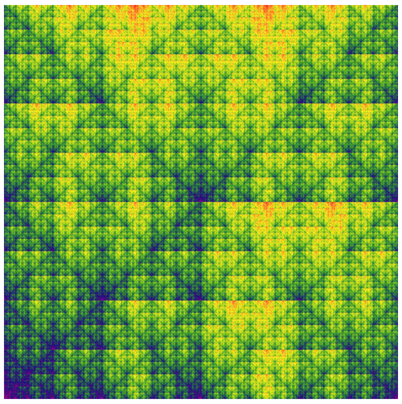


Figure 2.7: 32 bit, obfuscated `ror1`

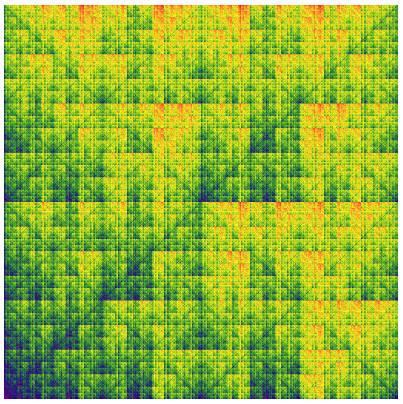


Figure 2.8: 32 bit, obfuscated `ror2`

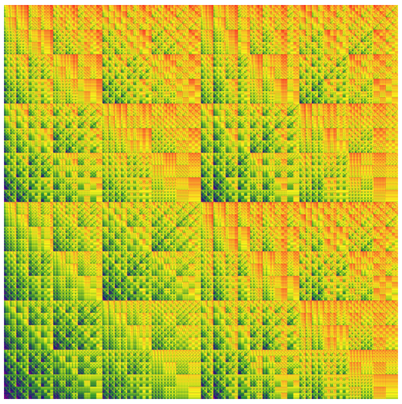


Figure 2.9: 32 bit, obfuscated `ror4`

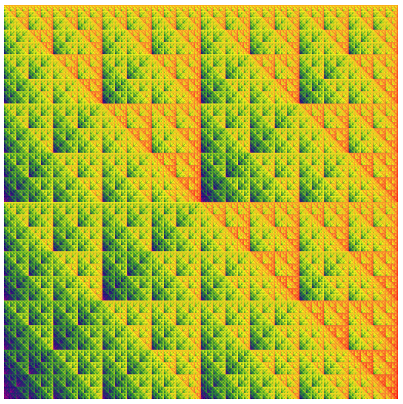
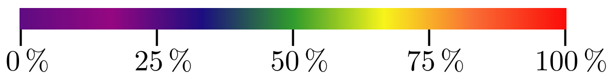


Figure 2.10: 32 bit, obfuscated `ror16`



Heatmap scale: Normalized number of flipped bits.

	Bit flip count		
	Maximum	Average	Std. Dev.
Plain 32-bit (trunc)	9	4.5	1.5
Obfuscated 32-bit (trunc, `ror1`)	27	14.5	2.7
Obfuscated 32-bit (trunc, `ror2`)	27	15.5	2.8
Obfuscated 32-bit (trunc, `ror4`)	27	17.5	3.0
Obfuscated 32-bit (trunc, `ror16`)	36	22.5	4.5
Plain 9-bit	9	4.5	1.5
Obfuscated 9-bit (`ror5`)	25	13.5	2.6

Table 2.1: Bit flip count in various scenarios.

the displayed variant. This number is shown in Tab. 2.1. A normalized representation was shown to highlight the relative differences in bit flips in dependence on the two input registers since it is precisely such a difference which results in data leakage in the form of a side channel.

In contrast to the original variant, the rotated XOR counterparts all have a dependence on both operands. While there are still fractal patterns visible, the relationship is much more intricate in the surrogate instruction combination. Note that even with this approach, the instruction which causes the original leakage still causes the same amount of leakage. It is, however, now surrounded by instructions which cause obfuscating leakage to decrease the signal-to-noise ratio of the taken measurements. Also, note that in practice, multiple of these instructions are executed randomly, and pre-charging is also applied. This means that invariant register values are filled with random data to obscure power omission.

Tab. 2.1 shows statistical data on the performed operations. You can see that in the unobfuscated 32-bit version the maximum number of bit flips is nine. This stems from the fact that we truncated the collected data because the calculation of the whole 32×32 matrix would have taken 2^{64} operations. We therefore only take a look at a small fraction of the 32-bit space, the least significant 9 bits. Note that for all obfuscated variants, the total power emission rises by a factor of about three and the standard deviation of the amount of flipped bits also drastically increases. This means that the overall power consumption of a device which performs these calculations is increased. This worsens the signal-to-noise ratio for an attacker who tries to perform DPA against the implementation.

2.3.3 Register Transfer Instructions

When we wish to obfuscate register transfer operations, we can apply the same trick which we already used in Sect. 2.3.2 and use the barrel shifter to cause bit flips through a rotation. This can be easily done for the register to register transfer, as shown in the original version in List. 2.7. We simply move the rotated variant in a first instruction and rotate the destination operand to its final value in a second instruction. This is shown in List. 2.7.

```
; r0 = r3
mov r0, r3
```

Listing 2.7: Plain move register to register instruction

```
; r0 = ror(r3, 16)
mov r0, r3, ror #16

; r0 = ror(r0, 16)
mov r0, r0, ror #16
```

Listing 2.8: Obfuscated move register to register instruction

For the immediate to register transfer, as shown in List. 2.9 this is a bit more tricky but still doable. In this case, we replace the `mov` by an XOR of a different, randomly chosen register and our desired immediate value as the third operand. The register that is used as a second operand cannot be selected completely at random (e.g., `r13` and `r15` are not usable because they have special meaning on the Cortex-M and therefore cannot be encoded in the `eor` instruction). In a second step, we cancel out our randomly chosen register again by performing another XOR operation, leaving only the desired immediate in place.

The modification relies on the assumption that the randomly chosen register contains some uncorrelated data. For cryptographic algorithms, this usually is not the case. It also has a second drawback: The original variant in List. 2.9 sets the condition code of the CPU according to the value of the second operand. In this case, the condition codes are set in a way that is only dependent on an immediate value and therefore entirely predictable at compile-time.

We can, however, assume with good certainty that the compiler does not issue instructions that rely on conditional code execution of an entirely deterministic comparison. The reason for this is that an optimizing computer would simplify such a construction at compile time. However, the possibility cannot be excluded and therefore we must check that the instructions that follow the move instruction do in fact modify the condition code register again. In other words, we need to be able to prove by static analysis that the condition flags that are set by the move instruction are unused because they are destroyed anyhow in instructions that succeed the `mov`. For all of our examples, this was the case. The final obfuscated variant is shown in List. 2.10.

Note again that we additionally perform random pre-charging for the operands, but omit that display here for clarity.

2.3.4 Bitwise Masking Instructions

In the analysis of instructions that cause the greatest power emanations, bit masking instructions play a major role. Among those is the `and` opcode. Curiously, if one looks

```
; r0 = 12345
mov r0, #12345
```

Listing 2.9: Plain move immediate to register instruction

```
; r0 = r5 ^ 12345
eor r0, r5, #12345

; r0 = r0 ^ r5
eor r0, r0, r5
```

Listing 2.10: Obfuscated move immediate to register instruction

closer at the generated assembly code of a cryptographic computation, such as in the case of AES-128, it becomes apparent that the `and` opcode is used almost exclusively to mask out specific bits. Concretely, the third operand is usually an immediate value. Again, for the AES-128 example which we analyzed, only three unique immediates are ever emitted by the compiler: `0x03`, `0x0f`, and `0xfc`. They are used to mask the least significant 2 bits, the least significant nibble and the most significant 6 bits of the least significant byte, respectively. All other 30, 28 or 28 bits of the register are set to zero after the respective instruction executed.

This means in reverse that the value of the bits before execution of the `and` operation is invariant to the semantic correctness of the code and can, therefore, be arbitrarily modified by us to obfuscate power emission.

```
; r0 = r3 & 0x0f
and r0, r3, #0x0f
```

Listing 2.11: Plain bit masking instruction

```
; r0 = r3
mov r0, r3

; r0 = r0 ^ (r7 << 4)
eor r0, r0, r7, lsl #4

; r0 = r0 & 0x0f
and r0, r0, #0x0f
```

Listing 2.12: Obfuscated bit masking instruction

To obfuscate the code, we again choose the randomized register approach as was shown for the `mov` instruction. We examine the immediate that is used in the `and` instruction and determine its most significant bit. For `0x0f`, this is bit three. We copy the register in the first instruction using a `mov`. Then we combine a second, randomly chosen, register with the copy using an XOR instruction. For the XOR we take care that the bits which we want to have in the ultimate result are not affected. We do this by using the barrel shifter to perform a logical shift left with the random register so that the shift width is one bit greater than the exponent of the most significant bit of the `and` immediate. In the given example, the exponent is three and we, therefore, choose to perform a bit shift of four bits. The register `r0` now contains garbage data in the most significant 28 bits. This garbage is masked out in the final `and` instruction. The combination of instructions produces the same result as its unobfuscated counterpart, but with a difference in power emission.

2.4 Efficient Runtime Recompilation

In the previous section, we showed some concrete examples of transformation which alter the power emanation that a device exhibits during computation. This is, however, only the first step to implementing code polymorphism at runtime. Indeed, the concrete implementation is the much more complicated part and can be considered core to our contribution.

To effectively perform the described masking operations, it is necessary to randomize the selected primitives before each run of the algorithm. This ensures that, even if some part of the data input to a cryptographic routine (like a symmetric key) remains identical, the power emission still differs even for identical input data.

One simple approach to achieve this could be to pre-compile multiple differently obfuscated variants of the same code all into the same binary. When masked function is called during runtime, the system would randomly choose which variant is actually called by some trampoline code. Such an approach has, however, the following problem: Consider an algorithm which has such countermeasures applied in the described way. For example, imagine an AES-128 encryption routine which has been obfuscated in n different ways. Firstly, it would require that the code side would increase by a factor of $n\omega$ where ω is the factor by which the code increases through the applied transformations. This means that n must be comparatively small because flash ROM of embedded systems is usually very limited.

The other downside is, however, that an attacker can easily learn n. First, she can create a single reference trace with arbitrary but fixed plaintext. Then she executes that same operation with the same plaintext over and over again and correlates the recorded traces against the references. If it does not match, it is recorded as a second variant in the reference pool. This operation continues until the number of variants in the reference pool does not increase anymore. In probability theory, this problem is well-known as the coupon collector's problem and has been described, for example, by Flajolet, Gardy, and Thimonier (1992). Croucher (2006) explain how the mean μ and standard deviation σ of the underlying normal distribution can be calculated for a given value of n:

$$\begin{aligned}\mu_1 &= 1\\ \mu_n &= \frac{n \cdot \mu_{n-1}}{n-1} + 1\end{aligned}$$

			LoC	
k	μ	σ	0.95	0.999
8	21.74	3.71	29	34
16	54.09	6.17	66	74
32	129.87	9.89	149	162
64	303.61	15.48	334	354

Table 2.2: Number of required probes to catch all variants with certain levels of confidence (LoC)

$$\sigma_n = \sqrt{\sum_{i=1}^{n-1} \frac{i}{n-i}}$$

Tab. 2.2 shows some typical values. For example, if we calculate 64 differently obfuscated variants to use for our countermeasure, it would take an attacker only 354 tries to see each different variant at least once with a probability of at least 99.9 %. As soon as an attacker knows a distinguishing feature of these traces, she can bin them together and perform power analysis on the binned traces. Therefore, it would only increase the difficulty of the attack by a linear factor of n.

To prevent such binning, it is important to randomize not only which algorithm we take, but also randomize the leaking fragments of code themselves. We decided to explore how well runtime binary recompilation would work for this purpose. As Agosta, Barenghi, and Pelosi (2012) have noted, there is a hefty runtime performance penalty which comes with a recompilation. While it may work on a larger Cortex-M3 or Cortex-M4, runtime disassembly and analysis of Thumb-2 code is definitively not computationally feasible on a Cortex-M0 type device. However, when abstracting away from the actual assembly code and viewing it as a pure bitstream, it becomes apparent that the recompiler does not necessarily have to be aware of this kind of abstraction level while still being able to perform adequate trace randomization. Our trick is to precalculate most operations beforehand during compile time and generate an extremely simple bytecode stream which performs the power emanation obfuscation. A minimalistic virtual machine (VM) then interprets this bytecode stream at runtime, obfuscating the code which performs the cryptographic computation. Afterward, the trampoline code calls the cryptographic routine that was just modified by the VM code. The actual recompilation operates solely on binary bitstreams and is blind to the assembly instruction abstraction. It, therefore, can perform its purpose with such efficiency that it is feasible to call the reobfuscation VM for every call to the underlying cryptographic primitive.

2.4.1 Virtual Machine Internals

The fact that the virtual machine bitstream is highly optimized makes the runtime performance penalty comparatively small compared to other approaches. However, it

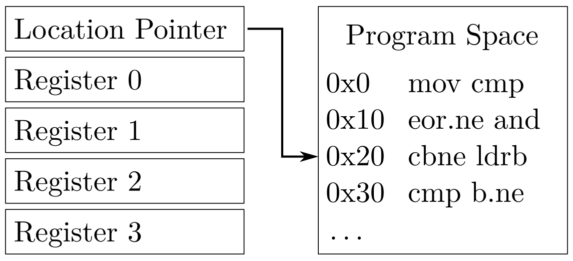


Figure 2.11: Internal construction of the morphing VM.

still is powerful enough to perform a wide range of code transformations. Our virtual machine has five registers, all of which are 16 bits wide. The location pointer (LP) is special since it points to the obfuscation target which resides in RAM. Addressing is done solely with absolute addresses; they are, however, absolute only within the VM itself and are, for the host system, relative to the start of the target in memory. Addresses are also always 32-bit aligned. In fact, it is not possible to encode setting the LP to a non-aligned address. This means that dereferencing of memory behind the LP can never throw a bus error and is always executed with maximum performance (compared to two split loads which some processors may perform in the background for misaligned access).

The VM features four registers `r0` to `r3`, which are also all 16 bits wide. Fig. 2.11 shows the general constitution of the reobfuscating VM. The machine bytecode typically resides in a different memory area (typically inside the flash ROM of the microcontroller). It allows encoding of four different opcodes:

- **`goto #absoffset`**: Sets the location pointer to an offset that is absolute within the recompilation unit. `#absoffset` must be a multiple of four.
- **`genrnd #num, #range`**: Generates `num` $(1 \dots 4)$ disjoint random integer values in a range from 0 up to `range`, inclusive. Stores those integer values in the virtual machine registers $r_0 \dots r_{\text{num}-1}$.
- **`patch reg, #width, #srcoffset, #dstoffset`**: Takes `width` bits of a virtual machine register `reg` starting at bit offset `srcoffset` and patches those bits into the 32-bit word at the current location pointer at bit offset `dstoffset`.
- **`shuffle #blksize, #blkcnt`**: At the current location pointer, shuffle the following `blkcnt` memory blocks of `blksize` each. Block size is a multiple of 2 bytes because any valid Thumb-2 instruction is either 16 or 32 bits wide.

2.4.2 Examples

Some opcodes can be understood intuitively, like the `goto` opcode. The only caveat there is that the encoded address is multiplied by four before being interpreted as an address.

Opcode	Parameters	Range	Encoding
`goto`	`A`: #absoffset	0 . . . 65532	`00AA AAAA AAAA AAAA`
`genrnd`	`N`: #num	1 . . . 4	`01NN RRRR RRRR RRRR`
	`R`: #range	0 . . . 4095	
`patch`	`R`: reg	`r0` . . . `r3`	`10RR WWWW SSSS DDDD`
	`W`: #width	1 . . . 32	
	`S`: #srcoffset	0 . . . 31	
	`D`: #dstoffset	0 . . . 31	
`shuffle`	`B`: #blksize	2 . . . 16	`11BB BCCC`
	`C`: #blkcnt	1 . . . 16	

Table 2.3: Opcode encoding.

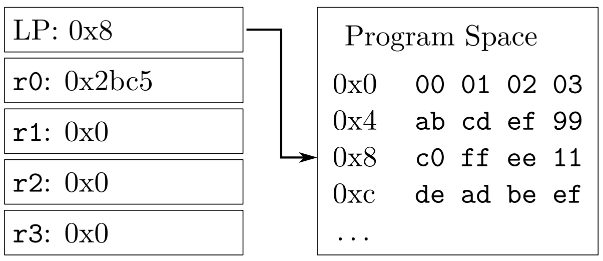


Figure 2.12: Exemplary state of the VM before execution of the `patch` opcode.

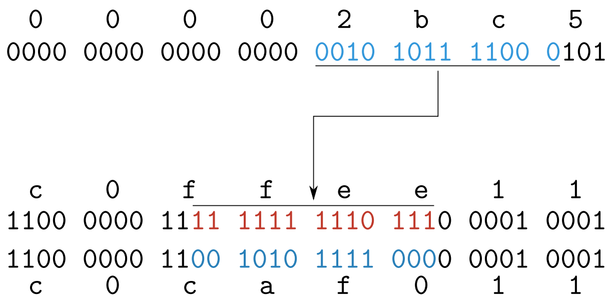


Figure 2.13: Applying `patch r0, #13, #3, #9` with `r0 = 0x2bc5` and `*LP = 0xc0ffee11`.

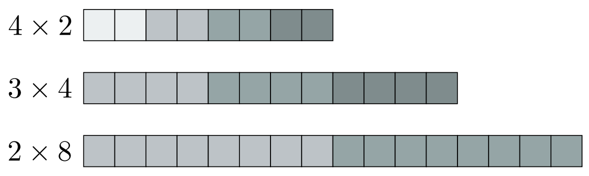


Figure 2.14: Shuffling of different blocks and counts.

Opcodes like `patch` are more complicated, but still straightforward: Fig. 2.12 shows the state of the VM just before execution of a `patch` instruction. In Fig. 2.13, we can see the execution of this VM instruction. The patch width is 13 bits, and these bits are taken with a bit offset of three from the source (register `r0`) and patched into the target with a bit offset of nine bits. In the example, the original memory content consequently changes from `0xc0ffee11` to `0xc0caf011`. List. 2.13 shows an example of Python code which could perform this transformation; the C equivalent looks almost identical and is therefore omitted here for brevity.

```python
def patch(value, regval, width, src, dst):
    mask = (1 << width) - 1
    patchval = (regval >> src) & mask
    value = value & ~(mask << dst)
    value = value | (patchval << dst)
    return value

assert(patch(0xc0ffee11, 0x2bc5, 13, 3, 9) == 0xc0caf011)
```

Listing 2.13: Illustration of `patch` code run by the VM.

With a bit of illustration, the `shuffle` opcode is similarly easy to understand. Fig. 2.14 illustrates the mode of operation. It shows shuffling of four memory blocks with two bytes each, three blocks of four bytes each and two blocks of eight bytes each, respectively. In all cases, blocks with identical color are always moved contiguously. This means that for a block count of k there are $k!$ possible combinations that the resulting code can have, giving a great number of possible randomizations for values up to $k = 32$. Note that for the `shuffle` opcode, the block size can vary from $2 \ldots 16$ in two-byte increments. Shuffling blocks in a misaligned manner (i.e., at offsets which are not divisible by two) does not make sense practically since all opcodes are at least 16 bits wide.

2.4.3 Workflow

To clarify the entire workflow, it is shown graphically in Fig. 2.15. When the user wants to protect a particular cryptographic routine which is available in source code, it first needs to be placed in its own compilation unit. Then it is compiled by the target compiler

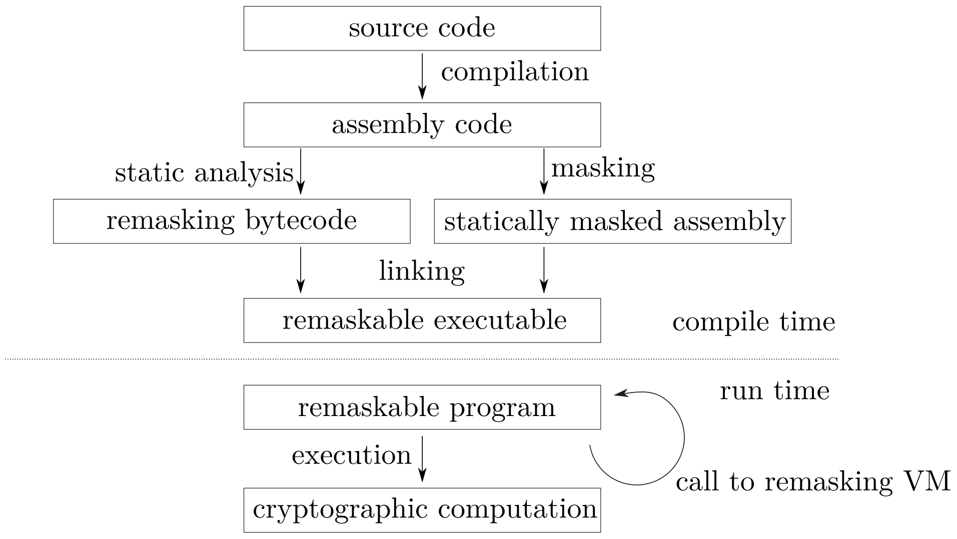


Figure 2.15: Remasking workflow.

to its assembly representation. This assembly code is then processed in two different ways: Control-flow analysis is performed with the code and fragments which exhibit potential side channel leakage is identified. These fragments are then substituted by masking alternatives and yield the statically masked assembly. This assembly code then already includes all the instructions which are used for obfuscating power emanation at runtime, but all intermediates (such as the random values used for pre-charging) are statically hardcoded. As such, when running from ROM, this code would only provide little to no additional protection compared with the unprotected binary.

However, during the same time that the substitutions are generated, the analyzer also records which exact substitutions the obfuscator has applied to which points of the code. For this, during code generation of the statically masked assembly, the obfuscator inserted unique machine-readable symbols at the specific locations where substitutions apply. The statically masked executable is then assembled to its object code representation. With the help of this object code representation, the code analyzer identifies at which location within the compilation unit the applied code transformations lie. With that information and knowledge about the constitution of the transformations, it can then emit the remasking bytecode. This is the VM bytecode which is stored as raw data in a C array.

The compiler combines the VM engine itself, its VM bytecode and the statically obfuscated protected compilation unit and forms the final binary. During runtime, when the cryptographic computation needs to be performed, the following happens: First, the minimal trampoline stub relocates the protected compilation unit into SRAM. Then it starts the VM and gives it the memory address of the relocated code. The location pointer (LP) of the VM, therefore, indexes relative to the beginning of the relocated code. Running the VM causes all statically applied obfuscations to be dynamically randomized and changed according to the previously defined VM recipe. Only then does the stub function call the cryptographic routine. This guarantees that every execution (and associated side channel leakage) varies randomly for each call of the target code.

2.4.4 Hardware True Random Number Generators

All software masking approaches, which want to evaluate the amount of protection that they can offer honestly, need to consider the quality of the underlying entropy source. To securely reobfuscate the target code, a good source of entropy is required. In our experiments, we used Cortex-M0, Cortex-M3 and Cortex-M4 microcontrollers. Of those, only the M4, an STM32F407VG, contains a true random number generator (TRNG; STMicroelectronics N.V. 2011b, 2012). It would be dishonest to simply assume a TRNG exists in all devices and omit the approaches for controllers without such luxury, however. Even though we do not use masking tables, the results of Tunstall, Whitnall, and E. Oswald (2014) apply to our approach: an adversary who faces masking countermeasures which rely on low entropy sources can circumvent these and therefore easily break the system.

To tap a source of entropy on Cortex-M microcontrollers, an approach that we looked into was that of Van Herrewege, van der Leest, et al. (2013). They analyzed the entropy present in the microcontroller SRAM after power up for an STM Cortex-M3 and came to the conclusion that it is well suited to generate strong, true random seeds for a PRNG. This PRNG, in turn, serves as the device entropy source.

A commonly described technique is the extraction of entropy from analog to digital converter (ADC) noise. This would undoubtedly work on the devices we examined since all are equipped with an analog to digital converter (ADC) in conjunction with an internal temperature sensor. In firmware, code can connect them internally so they fit our purpose (STMicroelectronics N.V. 2011b, 2012, 2015a,c).

We, however, tried to explore novel ways of utilizing existing hardware as sources of entropy that would be difficult to manipulate externally. For this, we took a closer look at the device-internal phase locked loop (PLL). The PLL is usually used to multiply an internal reference clock to a much higher speed. However, due to the physical constraints of a PLL, this clock multiplication cannot happen instantaneously, but the PLL needs some time to *lock*, i.e., to settle down and become stable. Whenever the PLL is locked, this is indicated by a register bit of the microcontroller because only then can the PLL output be used as an internal clock source. Our idea is to extract entropy out of the PLL locking time. This proved to be a surprisingly good source of entropy. We made measurements with this entropy source and give an estimate of the amount of extractable entropy in Sect. 2.5.

2.5 Experimental Evaluation

In this section, we highlight the experimental results of the ideas we developed in the previous sections. Sect. 2.5.2 explains how well our masking approach performed while Sect. 2.5.3 shows how efficient our techniques for entropy generation on low-cost devices are.

2.5.1 Static Analysis of Target

Since our approach made assumptions about the type of instructions which cause leakage in cryptographic code — namely, binary manipulation and load/stores — and since we mainly focused on these instructions, we now verify by analysis of actual code that these instructions are really of primary interest for defending against DPA. For this purpose, we evaluate some common cryptographic routines and categorize the instructions that the compiler emitted for our target, the ARM Cortex-M3.

The Cortex-M3 uses the Thumb-2 instruction set (ARM Ltd. 2014; Yiu 2009). We grouped Thumb-2 instructions which are commonly emitted into the following categories:

- *arith*: Arithmetic operations. Examples are `add`, `sub`, `mul`.
- *bitwise*: Bitwise operations. Examples are `and`, `or`, `eor`, `lsl`, `ror`.
- *branch*: Branch operations. Examples are `b`, `bx`, `bl`, `bne`, `cbz`.
- *cmp*: Comparator operations. Examples are `cmp`, `tst`, `it`, `ite`, `cmn`.
- *ldst*: Load and store operations. Examples are `ldr`, `str`, `ldmia`, `stmia`.
- *mask*: Bit masking operations. Examples are `uxtb`, `uxth`, `sxtb`, `sxth`.
- *move*: Register move operations. Examples are `mov`, `movn`.
- *stack*: Direct stack manipulation operations. Examples are `push`, `pop`.

Two things are noteworthy about the measurements which are shown in Tab. 2.4. First, the optimization level does not make a large difference and can be disregarded for all further analysis. Secondly, it shows that our basic assumption is true: The amount of bitwise instructions, load/store functionality and register transfer account for most of the instructions. We furthermore looked into the source of arithmetic instructions in the code. Such instructions were almost always used for purposes in which little or no side channel leakage can be expected, such as incrementing loop counter variables. However, the algorithms we looked at are also focused primarily on binary data transformations; for algorithms which use arithmetic transformation such as XTEA by Needham and Wheeler (1997), our approach could however be easily extended also specifically to protect arithmetic manipulation of data.

	AES-128 (LUT)			AES-256		
	-O2	-O3	-Os	-O2	-O3	-Os
arith	14.4 %	8.4 %	16.2 %	7.0 %	4.7 %	7.0 %
bitwise	16.4 %	19.3 %	16.8 %	18.1 %	25.7 %	18.5 %
branch	9.6 %	6.2 %	11.2 %	11.3 %	10.7 %	15.6 %
cmp	8.5 %	8.1 %	10.9 %	7.5 %	8.4 %	3.6 %
ldst	28.3 %	40.3 %	25.4 %	38.1 %	25.5 %	33.0 %
mask	3.7 %	2.8 %	3.2 %	5.6 %	10.3 %	4.3 %
move	16.1 %	12.8 %	13.3 %	8.7 %	13.7 %	13.4 %
stack	2.8 %	2.1 %	2.9 %	3.8 %	1.1 %	4.6 %

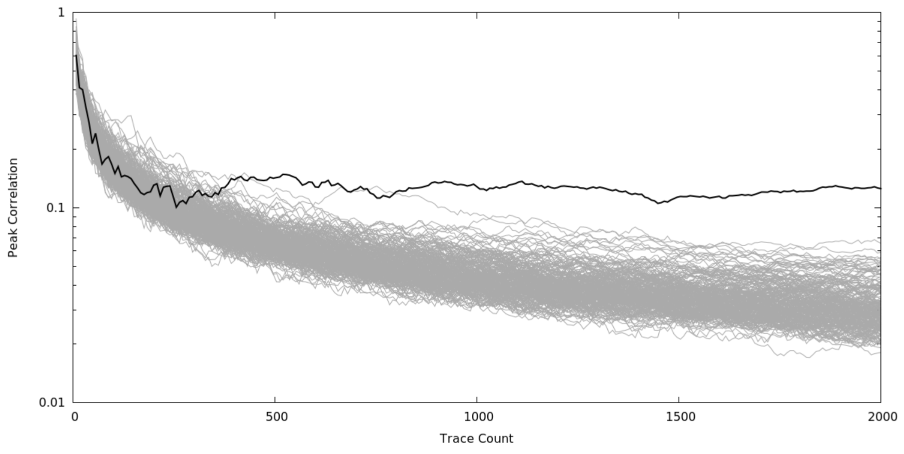
	SHA-256		
	-O2	-O3	-Os
arith	14.1 %	13.3 %	15.6 %
bitwise	12.5 %	13.0 %	14.1 %
branch	7.7 %	6.5 %	8.0 %
cmp	6.1 %	3.8 %	3.8 %
ldst	38.4 %	46.6 %	41.4 %
mask	0.0 %	0.0 %	0.0 %
move	18.5 %	14.6 %	14.8 %
stack	2.7 %	2.2 %	2.3 %

Table 2.4: Emitted instruction groups of different cryptographic algorithms.

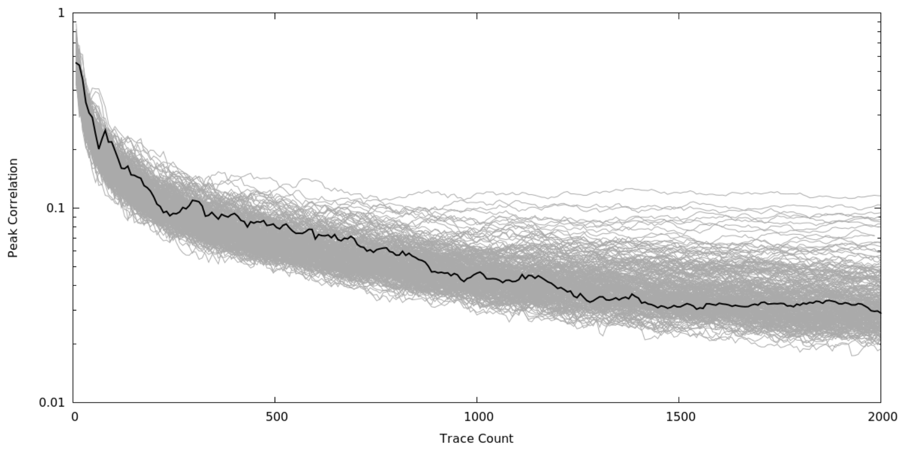
2.5.2 Masking Results

We did measurements on the effectiveness of the DPA masking algorithms on both a Cortex-M0 STM32F030C8 and a Cortex-M4 STM32F407VG. We only discuss the details of the weaker device, the M0, in-depth, but unsurprisingly the results for both platforms are quite similar. The current was measured over a resistive 10 Ohm high side shunt with a custom-made differential probe that used the Analog Devices AD8129AR. Our schematic was heavily influenced and inspired by the ChipWhisperer differential probe of O'Flynn and Chen (2014). We used a mid-range Rigol DS2202 oscilloscope to perform our measurements to take the role of an attacker with reasonable hobbyist equipment. We sampled the current at 1 GS/s for acquisitions of 2.7 MPts in each trace. We transferred this data to the evaluating PC via Ethernet using the LXI protocol. We then used differential power analysis using a standard Hamming difference model to extract one byte of the cryptographic key of an AES-128 encryption in ECB mode. The AES source code we used for evaluation was a public domain reference implementation by Paulo Barreto. Our results compare the unprotected against the protected AES-128 variant.

The results of the DPA attack are shown in Fig. 2.16a and Fig. 2.16b. In both cases, the black trace shows the correct key hypothesis. On the top, the unprotected variant



(a) Unprotected.



(b) Protected.

Figure 2.16: Number of considered power traces against peak correlation coefficient. Black trace shows the correct key hypothesis.

Static AES modifications	234 bytes (1034 bytes total)
PRNG	1124 bytes
VM code	210 bytes
VM bytecode	288 bytes in 145 opcodes

Table 2.5: Flash ROM space demands incurred by protection mechanisms.

Necessary entropy for VM run	231 bit
PRNG speed	3.6 μs (29 CC) per byte
Total time for entropy collection	104 μs (835 CC)

Table 2.6: Time and clock cycle (CC) count necessary for entropy collection.

is shown. The correct key hypothesis is easily discernible from the remaining traces. The image on the bottom shows the runs after randomized masking had been activated. Reobfuscation was applied before each AES operation.

Tab. 2.5 shows the requirements on flash ROM that were demanded by the implemented countermeasures. The AES implementation itself only grows by a small amount (800 bytes for the original, 1034 bytes for the protected variant, i.e., a 29 % increase in code size). Most of the ROM was needed by the reference ISAAC PRNG implementation. The VM code, due to its extreme simplicity, is tiny (210 bytes). In fact, even the VM bytecode instructions (288 bytes) are larger than the VM execution engine itself.

Regarding runtime, Tab. 2.7 gives an overview. The unprotected AES encryption operation took 2.18 ms on our device. This rises to 2.99 ms for execution of the protected AES. Additionally, to the pure execution time, there is management overhead that comes with the protected variant. For once, our PRNG was measured to produce 256 bytes of entropy in about 920 μs. This averages at about 3.6 μs per byte of entropy out of the PRNG, as shown in Tab. 2.6. For the VM runtime requirement of 231 bits, i.e., 29 bytes, this means a duration of 104 μs. Additionally, the code needs to be relocated from ROM to RAM (880 μs), and the VM needs to run to perform the remasking (770 μs). All in all, a fully protected AES run takes around 4.74 ms which is about 2.2 times that of its unprotected counterpart.

Entropy collection (see Tab. 2.6)	104 μs (835 CC)
Relocation duration	880 μs (7040 CC)
Reobfuscation duration	770 μs (6160 CC)
Protected AES runtime	2.99 ms (23920 CC)
Total time for protected AES run	4.74 ms (37955 CC)
Unprotected AES runtime	2.18 ms (17472 CC)

Table 2.7: Protection runtime and clock cycle (CC) count.

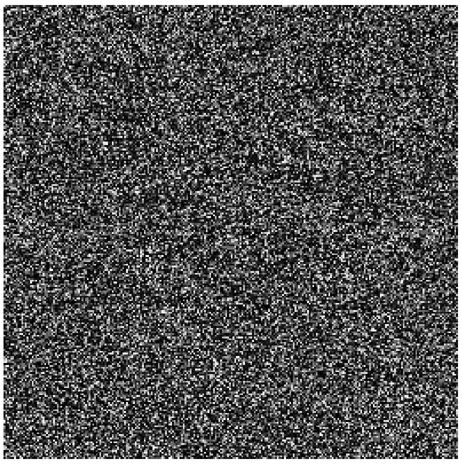


Figure 2.17: 512 kiBit of preconditioned collected entropy from PLL lock time source.

2.5.3 Entropy Collection

In our experiments, we implemented our idea to use the PLL as entropy source (see Sect. 2.4.4). On an STM32F030C8 which was clocked from an 8 MHz high-speed external (HSE) crystal oscillator, we collected entropy in the fashion that was described there. We made around 57 million trials in which we configured the internal PLL to get its reference clock from high speed internal (HSI) RC oscillator clock divided by two (i.e., 4 MHz). This clock was to be multiplied by a factor of 12 to result in the maximum speed of 48 MHz. The typical time for the PLL to settle was 69 μs in 98.2 % of cases, 61 μs in 1.5 % and 78 μs in the remaining 0.3 % of cases. Of these, we interpreted the 98.2 % majority as one outcome of a coin flip operation and the remaining 1.8 % as the other. Then we applied the standard unbiasing procedure as described by von Neumann (1951), where the outcome of two captured events was compared. If the outcomes were identical, both were discarded; were they not equal, the first of both values was taken as the unbiased value. While 98.2 % to 1.8 % seems like a significant disparity, the high acquisition speed of about 14.5 kBit/s compensates for this. In our experiments, we were able to extract successfully about 927 kBit of unbiased entropy from the 57 million operations. The ratio of extracted entropy to the number of trials was 1.6 %. This gives about 268 bits of unbiased, extractable randomness per second from this entropy source.

Also, we repeated the experiments of Van Herrewege, van der Leest, et al. (2013) to check if they also apply to the STM32F030C8. For this, we erased the on-chip flash ROM memory so that the chip would immediately hard fault at reset. Then we extracted the SRAM contents via JTAG. During measurement, we were careful that the JTAG adapter did not parasitically power the chip itself by separating all the serial wire debug (SWD) interface lines as well as the supply voltage using mechanical relays. Before power-off, we used the JTAG interface to fill the SRAM with random data to eliminate hysteresis effects of memory cells. We performed 2700 of these measurements, extracting 8 kiB of uninitialized SRAM at each run (i.e., about 21 MiB of data). In our trials, we deliberately varied the ambient temperature between 15 °C up to 40 °C. We found that the SRAM power-up initialization contained many static elements. Over all 2700 runs, we saw behavior reminiscent of physical unclonable functions (PUFs): 48370 of the 65536 bits (i.e., 73.8 %) always took a fixed value after the chip reset. There still is an abundance of random bits within the SRAM initialization pattern with the remaining 17166 bits (26.2 %). So overall, we could confirm the results of Van Herrewege, van der Leest, et al. (2013): Uninitialized SRAM is a plentiful entropy source that comes “for free” regarding runtime cost for acquisition and only has to be conditioned to be used. However, using the entire SRAM as source creates the problem of storing entropy for later use. Furthermore, the entropy is only generated once per reset of the system. In contrast, our entropy source is infinite but requires additional runtime cost.

For the concrete implementation and running of our VM, we used both entropy sources to feed an ISAAC CSPRNG (Aumasson 2006; Jr. 1996) for entropy mixing and conditioning. If entropy pool mixing performance should be a problem, an alternative is to follow the steps described by Van Herrewege and Verbauwhede (2014) who describe how to implement a lightweight, fast CSPRNG using Keccak for the Cortex-M platform.

2.6 Conclusion

We presented novel techniques with which effective masking transformations can be realized efficiently even on small Cortex-M0 microcontrollers. Obviously, the main drawback of our solution is that it requires a writable code segment for the dynamic recompilation to occur. In exchange for this, however, it executes with excellent runtime performance all while implementing resistance to power analysis. No special hardware such as a TRNG is required for our approach to work. Our work aligns well with the existing literature on software-only countermeasures and provides a different angle to complete the spectrum of tools and possibilities which software engineers have.

Another drawback of a software-only approach is the negative impact on performance. However, this is something that can be fine-tuned with our approach in two dimensions: Firstly, the developer can decide that reobfuscation of the relocated binary code does not need to happen every time. This gives a linear performance gain with the number of run trials. The second dimension happens during compile time. It also is up to the user to decide how heavily surrounded the potentially leaking code should be with code that aims to obfuscate power emanation. Our work, therefore, gives the developer the tools to fine-grain their actual implementation according to their needs regarding efficiency and security.

In our opinion, the amount of attacks on low-cost microcontrollers is, in all likelihood, increasing in the near future due to the sheer simplicity with which attackers can conduct them today. We hope our work gives a good impression on what cost-effective countermeasures can be realized by creatively leveraging the tools that are already present in hardware.

Chapter 3

Timing Channels

Leakage of information through timing side channels is a problem for all sorts of computing machinery, but the impact of such channels is especially dramatic on embedded systems. The reason for this is that these environments allow attackers to exploit small timing differences down to clock cycle accuracy. On the defensive side, it is, therefore, advisable to evaluate cautiously if security-critical code contains data-dependent timing discrepancies. When working with real hardware, testing for such vulnerabilities is a tedious process. To reduce the burden of vetting, we study approaches that allow cycle-accurate behavioral emulation of relevant CPU behavior such as instruction pipeline flushes and bus contention. We show that our approach is feasible and efficient by implementing an emulator of the popular ARM Cortex-M core. Furthermore, we give an overview of the problems of cycle-accurate emulation and demonstrate our approach towards a cycle-accurate ARM Thumb-2 simulator. As a practical application, we show how this simulator can be integrated into the build process of firmware to check for the presence of timing side channels before the system is deployed.

This chapter has been accepted for publication as a full paper at the 11th International Conference on Availability, Reliability and Security (Bauer and Freiling 2016).

3.1 Introduction

Security issues in real world systems do not only arise due to a flawed design but also due to parasitic side effects which any computing machine exhibits. Differences in the power consumption, for example, lead to the presence of so-called *power emission side channels*. When the electromagnetic emission of the hardware changes with the data that is handled in a certain computation, we speak of *EM side channels*. The most intuitive class, however, are *timing side channels*. In these, some leakage is inadvertently generated by the fact that computations on secret data exhibit timing differences that depend on that data.

These timing side channels have been known since their first introduction by Lampson (1973). In modern systems, they usually arise due to different optimizations within hardware or software. The reason why they are so prevalent on desktop computers, for example, is that desktop CPUs are using extraordinarily sophisticated techniques to optimize system performance aggressively. Such techniques include caching, instruction reordering or branch prediction. Unfortunately, these methods give rise to data and code dependent timing behavior.

Traditionally, in embedded environments, these hardware optimization techniques were neither necessary nor were they particularly welcome. Since embedded systems and real-time computing often go hand in hand, predictability is of utmost importance to developers. Sophisticated mechanisms like branch prediction or caching were not prevalent at all. In more recent microcontroller architectures, however, these mechanisms slowly start to appear. In particular, the popular ARM Cortex-M architecture, which has picked up significant momentum in the last few years, shows features which previously only were present on highly sophisticated PC CPUs. One reason for their introduction is that internal MCU peripherals often cannot keep up with the high core clock speeds of modern microcontrollers. Without caching mechanisms, the slowest peripherals, such as flash ROM, would restrict the overall performance of the system significantly. With the introduction of these techniques, however, the same timing side channels which were also seen in PC environments before now also appear in microcontrollers.

One significant difference, however, is the fact that in an embedded environment, timing side channels leak more information than in a PC environment. On a PC, an attacker usually needs to rely on imprecise measurements of time stamp counters and significant noise is present due to effects of the operating system. Such limitations are far less common in the embedded world. An attacker with physical access to a device is usually able to control the primary clock source and has, therefore, the ability to slow down time to his liking. With today's mid-range hobbyist equipment, it is possible to perform cycle-accurate timing measurements on such embedded systems. In many cases, the firmware runs directly on the *bare metal*, i.e., directly on the hardware with no operating system layer in between. Even if an embedded operating system is present, it usually exhibits much more predictable timing characteristics than operating systems for the consumer market.

The motivation of such an attacker could be to gain access to an otherwise unavailable administration interface or extract information about the internal workings of such a device. For example, if an attacker would control a smart meter and obtain the asymmetric private key, she could forge meter values and send arbitrary data to the utility company. Likewise, an attacker could find a master password for a manageable switch using timing attacks; although bad security practice, many vendors still ship their devices with such back doors in the hope that the protecting password remains secret.

3.1.1 Related Work

As with a lot of practical side channel work, Kocher (1996) also pioneered the field of timing side channel analysis. He highlighted that timing differences in asymmetric cryptographic operations could lead to the disclosure of private key data. Two years later, other implementations of his proposals emerged and were published by Dhem et al. (1998, 2000). Around the same time, Kelsey et al. (1998) generalized on these side channels and showed their presence in popular algorithms such as IDEA, RC5, and DES. Of particular interest to our work is the timing side channel they described in IDEA, where they exploited the timing difference present in a multiplication modulo $2^{16} + 1$. In their example, a multiply-by-zero operation took significantly less time than a multiplication by a non-zero value.

Such side channels commonly accompany software-computed algebraic field operations. At that time it was widely believed that an effective defense against this kind of timing attacks would be to use constant-time lookup tables for field operations. Page (2002), however, showed how cache timing might cause these supposedly constant-time lookups to exhibit exploitable timing differences. Indeed, Tsunoo et al. (2003) demonstrated the effects of leakage caused by caching effects to be relevant on real-world systems. Exploiting timing differences, they were able to break DES with a probability of over 90 % with an astonishingly low amount of 2^{24} operations. Such attacks were later shown by Bernstein (2005) also to apply to the more recent AES block cipher. A relatively well-known hack on the MSP430 mask ROM boot loader was presented by Goodspeed (2008): He exploited the timing differences of different control flow paths to find out the correct boot loader password of MSP430 devices.

Osvik, Shamir, and Tromer (2006) proposed a general mitigation method for these problems based on different approaches such as normalization of cache timings, disabling of caches altogether or hiding mechanisms to obfuscate the leakage. Z. Wang and Lee (2007) proposed hardware countermeasures, namely a new type of cache architecture, which claimed to solve side channel emission. However, Kong et al. (2008) highlighted serious issues with this proposal, confirming yet again how difficult it is to eliminate side channels in cached architectures.

Cycle-accurate simulation is a topic that is not only of interest to security researchers but also for optimization purposes. Yourst (2007) presented such a cycle-accurate simulator for the x86-64 architecture. Since the x86-64 is much more complicated than the Cortex-M architecture, their goal was to achieve 5 % accuracy for all the key simulation parameters; since the sophisticated optimizations of the x86-64 are not present in the Cortex-M architecture, the relative timing determinism of the Cortex-M enables much more accurate results in our case.

The Cortex-M uses internal SRAM for volatile storage, which is another contributing factor to the relatively straightforward implementation of an emulator. In contrast to DRAM, SRAM exhibits deterministic timing characteristics. To apply our results to DRAM, one could rely on complex simulations of DRAM timings, such as in the paper presented by Rosenfeld, Cooper-Balis, and Jacob (2011).

If the performance of the emulator is of interest — which for us was a secondary objective only — the work of Ye et al. (2000) is also of interest. They show which optimizations apply to a simulator while preserving its emulation accuracy. In their case, they use their technique to improve the efficiency of a power consumption simulator.

Our approach relies on behavioral simulation of the architectural features. How processors can be modeled from a hardware perspective, for example during simulation of a synthesized FPGA, is covered by the work of Reshadi and Dutt (2005).

3.1.2 Contributions

We present a practical approach to detecting timing side channels in Cortex-M firmware. Our goal is to detect such channels reliably at implementation time. To do this, we built a behavioral Thumb-2 emulator with the particular purpose to correctly model timing behavior of the Cortex-M architecture down to clock cycle accuracy. We show how this emulator can help identify possible timing side channel leakage and how it can be incorporated into automatic vetting checks. To summarize, we make the following contributions:

- We describe our semi-automatic method of extracting hardware-dependent information about runtime behavior from a real microcontroller and how to use this knowledge to create a cycle-accurate Cortex-M core emulator.
- We show how such an emulator can be integrated into the vetting process to prevent timing side channel leakage for embedded systems in an automated fashion.

3.1.3 Outline

This chapter is structured as follows: Sect. 3.2 gives the necessary background to understand aspects of modern CPU design which are relevant to our topic. We then proceed to discuss the factors which influence runtime behavior of our target platform in Sect. 3.3 and highlight important aspects of the design of a cycle accurate emulator. In particular, we point out how real-world measurements on physical hardware can quickly be turned into a model for an emulator which we wrote. Our design is then evaluated in Sect. 3.4 against real-world cryptographic algorithms and we give an example of how our simulator can be integrated into a build workflow to achieve a semi-automatic vetting process. Afterward, we discuss the results and give an outlook in Sect. 3.5.

3.2 Background

In the following section, we give an overview of relevant concepts of modern CPUs in Sect. 3.2.1. Then we proceed to show details about the particular CPU family (ARM Cortex-M) which we worked with in Sect. 3.2.2.

3.2.1 Factors Influencing Execution Time in Modern CPUs

The internal construction of a modern CPU divides instruction execution into three phases: the fetch, decode and execution stages. At the first stage, one or more instructions are loaded from the position to which the program counter points. This is done via a read on the bus which fetches the data from that address. In the next state, the instruction is decoded. This means the CPU evaluates which sub-components of the CPU need to be

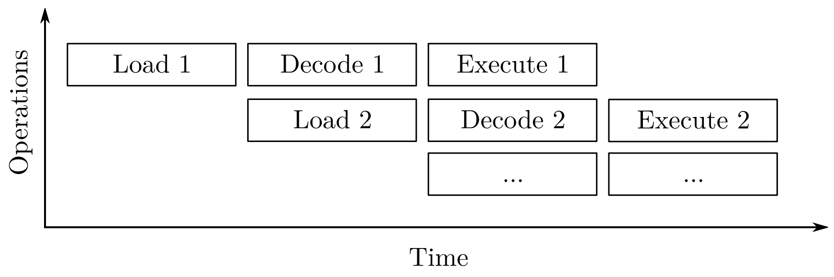


Figure 3.1: Pipelined instruction execution

enabled to perform the action which is requested by the opcode. After it determines this, the instruction is executed. In the process of execution, there might again be access to a memory bus required, depending on the action that the opcode is supposed to perform.

Loading of instructions and decoding or execution can be parallelized. This is called *pipelining* and is shown in Fig. 3.1. While the CPU executes the first instruction, it can already — at the same clock cycle — concurrently fetch the next instruction. If the CPU has prefetched instructions and filled up the pipeline, but notices in the decoding stage that the prefetched instruction is not the next in line to be executed, the pipeline needs to be *flushed*, and it needs to be refilled with the correct instructions. This is the case, for example, when a conditional jump takes control flow away from the instructions which the CPU had already prefetched.

There are also multiple factors which influence how long the execution stage of instructions takes. First and foremost is, of course, the actual computation that has been requested by the opcode itself. Some complex instructions need to be broken down by the CPU into smaller micro-instructions which are computed sequentially. For example, if there is a register indirect access with displacement, the CPU has first to calculate the effective address and then execute the actual memory operation. The instruction is therefore broken into two parts: address calculation is performed by the *arithmetic logic unit* (ALU) after which comes the store operation. Both parts may or may not be pipelined, depending on the concrete architecture. The typical example of a complex instruction which takes a variable amount of clock cycles to execute is the integer division operation.

Another factor which influences runtime is the dependency on a bus. Only one load or store can happen on a bus at any point in time. Access to the bus, therefore, has to be carefully coordinated. If multiple concurrent requests require bus access, *bus contention* occurs, and the CPU must perform *arbitration* between the concurrent requests. The bus peripherals might also have some inherent latency associated with it. For example, typical external memory or internal flash ROM cannot serve data as fast as the internal CPU clock might require it for continuous operation. Therefore, the CPU has to wait some amount of time after the address has been put onto the bus before the data becomes valid. The time during which the CPU waits for a reaction from the bus is indicated by the number of *wait states*.

Lastly, *caching* is something that has a significant influence on the real world runtime of a system. A cache miss is associated with the penalty to perform the actual read from the bus while access to cached data is typically faster by several orders of magnitude.

3.2.2 STM32 Cortex-M4 Specifics

The popular 32-bit Cortex-M architecture uses the ARM Thumb-2 instruction set. This is an instruction set in which opcodes are encoded either in the narrow 16-bit form or in the wide 32-bit form. The CPU core of the M4 uses an instruction pipeline which is 32 bits wide. Therefore, depending on the width of the instructions at the location of the program counter, either two narrow or one wide instruction is prefetched into the instruction pipeline (STMicroelectronics N.V. 2011b). Most instructions of the Cortex-M take either one or two clock cycles to execute, with the notable exception of the division unit which, depending on the processed data, takes anywhere in between 2 and 12 clock cycles for execution (ARM Ltd. 2010). For operations which perform load/store actions, there is an additional penalty associated that is directly proportional to the amount of data that is to be loaded or stored.

While most System-on-Chips (SoCs) that target the embedded market go without any caches because it makes predictions about execution time much more challenging, the STM32 Cortex-M4 does have one instruction cache. This cache is referred to by STM as the adaptive real-time memory accelerator (ART). It caches access to the internal flash ROM memory which is unable to keep up with the core clock when the microcontroller unit (MCU) runs at high speeds. For example, at 3.3V the internal STM32F4 flash ROM can only provide zero wait state operation up to 30 MHz, but can require as many as seven wait states at the low-voltage 1.8V operation when the CPU is clocked faster than 112 MHz (STMicroelectronics N.V. 2011a,b).

Like almost all architectures within the Cortex-M family, the M4 is based on a Harvard memory architecture. Concretely, this means that data and instructions are accessed via different buses. For normal operation, the text segment (i.e., where the executed instructions reside) is located within the flash ROM of the MCU and data is stored in the internal SRAM. Instructions are usually fetched via access to the instruction bus (I-Bus), but may also be fetched on the system bus (S-Bus), albeit less efficiently. Data access is performed on the data bus (D-Bus) or also via the S-Bus. The I-Bus and D-Bus can access only the lower 512 MiB of the 32-bit address space while the S-Bus can access almost all the remaining 3584 MiB (ARM Ltd. 2010; STMicroelectronics N.V. 2011b).

3.3 Cycle-Accurate Timing Simulation

We now focus on a particular microcontroller and briefly describe the effects that make a naïve prediction of execution time difficult. We continue by explaining the constitution of our emulator model is and how it integrates into semi-automatic verification of code.

3.3.1 Execution Time Prediction

The standard attacker model for embedded systems places the system itself under full physical access of the attacker. This means an attacker can control the environment in which the microcontroller executes code. Part of the environment is a reference clock which is usually supplied externally in the form of a quartz crystal. An attacker who has physical access to such a system can, therefore, modify the hardware itself (e.g., by changing this clock crystal) to force the system to slow down. This allows maximally precise, cycle accurate measurement with even mid-range commercial off-the-shelf hobbyist equipment. Any single clock cycle difference in timing can lead to an exploitable security vulnerability of such a system.

Consider the source code which we present in List. 3.1. It shows a `memcmp` function which differs from the standard `memcmp` in the way that no lazy abort is performed as soon as the first inequality is encountered between characters of the two supplied input buffers. While the overall result still is computed in a lazy fashion, the function always walks over the complete buffer in every case in an attempt to achieve constant execution time. This is something that a programmer who is aware of potential timing side channel leakage might do.

```c
int memcmp_cet(const uint8_t *a, const uint8_t *b, int len) {
    int result = 0;
    for (int i = 0; i < len; i++) {
        int char_result = a[i] - b[i];
        if (result == 0) result = char_result;
    }
    return result;
}
```

Listing 3.1: High-level `memcmp` routine which tries to achieve constant execution time

If you take a look at List. 3.2 you can see how the GNU C compiler gcc 5.2.0 translated this code into ARM Thumb-2 assembly. You can see that the loop indeed covers all `len` bytes. However, you might also notice that the compare-branch-if-not-zero instruction at `0x9b4` conditionally skips the following `subs` instruction if `result != 0`. This is the translated equivalent of the `if` condition. The code has, therefore, less work to do once `result != 0` and you might assume that it executes a tiny bit faster whenever the `subs` is skipped.

To show the real-world effect of this code, we ran it on an STM32F407 microcontroller with variable input data. We then used the *embedded trace macrocell* (ETM) which we configured to monitor the executed cycle count using the CPU-internal *data watchpoint trigger* (DWT). The code to do this is shown in List. 3.3. Our results were checked for plausibility by connecting a Rigol DS2202 oscilloscope to the microcontroller. The code

```
memcmp_cet:
  9a4:  2300  movs    r3, #0              ; r3 = 0 (i)
  9a6:  b570  push    {r4, r5, r6, lr}
  9a8:  4604  mov     r4, r0              ; r4 = r0 = a
  9aa:  4618  mov     r0, r3              ; r0 = r3 = 0 (retval)

loop:
  9ac:  4293  cmp     r3, r2              ; if (i < len)
  9ae:  da05  bge.n   9bc                 ;     then goto exit
  9b0:  5ce6  ldrb    r6, [r4, r3]        ; r6 = r4[r3] (a[i])
  9b2:  5ccd  ldrb    r5, [r1, r3]        ; r5 = r1[r3] (b[i])
  9b4:  b900  cbnz    r0, 9b8             ; if (r0 != 0) goto skip
  9b6:  1b70  subs    r0, r6, r5          ; r0 = r6 - r5 (retval)

skip:
  9b8:  3301  adds    r3, #1              ; r3 += 1 (i++)
  9ba:  e7f7  b.n     9ac                 ; goto loop

exit:
  9bc:  bd70  pop     {r4, r5, r6, pc}
```

Listing 3.2: Compiled `memcmp` routine in Thumb-2 assembly

surrounding the profiling target generated a rising edge on a GPIO pin upon entry of the function and a falling edge on the return. We then triggered on the rising edge but watched the *falling* edge of the signal with the infinite persistence function enabled. The different runtimes of the function can then clearly be seen on the oscilloscope.

When running the code from internal flash ROM, with a core clock of 8 MHz, the runtime of a comparison in which the first characters differed was 196 cycles. For each byte at the head of the buffers which were equal, the routine became *faster* one clock cycle. Similar effects could be observed with code running from internal RAM: A comparison which differed at the first character took 279 clock cycles, but the routine also became faster for each correct heading character by *four* clock cycles.

Two aspects of this might seem odd and surprising: One is that the routine, no matter from where it is executed, becomes *faster* with an increasing count of equal heading characters. From the high-level perspective, more work has to be done for each equal heading byte, so you might assume the routine to become *slower* for each match. Intriguingly, the opposite is the case. It also seems counterintuitive that the routine would run much faster from internal flash memory than when it runs from internal SRAM since the latter is typically far quicker regarding access times than flash ROM.

When taking a closer look at the architecture, however, both effects can be explained:

```
// Instruction Trace Macrocell, unlock register access
ITM->LAR = 0xC5ACCE55;

// Debug Exception and Monitor Control, enable trace cell
COREDBG->DEMCR |= TRCENA;

// Reset counter and set trace cell mode to cycle counting
DWT->CYCCNT = 0;
DWT->CTRL |= CYCCNTENA;
```

Listing 3.3: Activation of cycle counting on the STM32F4

That the routine becomes faster with each matching character stems from the fact that the compare-branch-if-not-zero instruction `cbnz` needs to skip the following `subs` by performing the conditional branch. The performance penalty which incurs with this taken branch is caused by the required instruction pipeline flush.

That flash ROM, in this particular case, is actually faster than SRAM is also explainable: When instructions are loaded from internal flash ROM, the I-Bus is connected to the flash peripheral and the data access to RAM is performed via the S-Bus. As soon as both instructions and data come from RAM, however, the S-Bus has to be used for both: it is the only remaining bus that can access SRAM. This explains why performance decreases once instructions are served from SRAM. Bus contention and arbitration are leading to this degradation in performance.

Another effect that we would like to illustrate is the effect of wait states when retrieving data from flash ROM. Consider the code given in List. 3.4.

```
#define FLASH_ROM  ((volatile const uint8_t*)0x08000000)
#define RAM        ((volatile const uint8_t*)0x20000000)

int waitstate_test(int len) {
    uint8_t result = 0;
    for (int i = 0; i < 10000; i++) result ^= RAM[i];
    for (int i = 0; i < len; i++) result ^= FLASH_ROM[i];
    return result;
}
```

Listing 3.4: Code to demonstrate wait state influence

This code first XORs the first 10000 bytes of SRAM to get a steady baseline and then XORs n bytes of flash ROM on top of it. We executed that function and for each run determined the clock cycle differential in runtime between the invocation with a byte

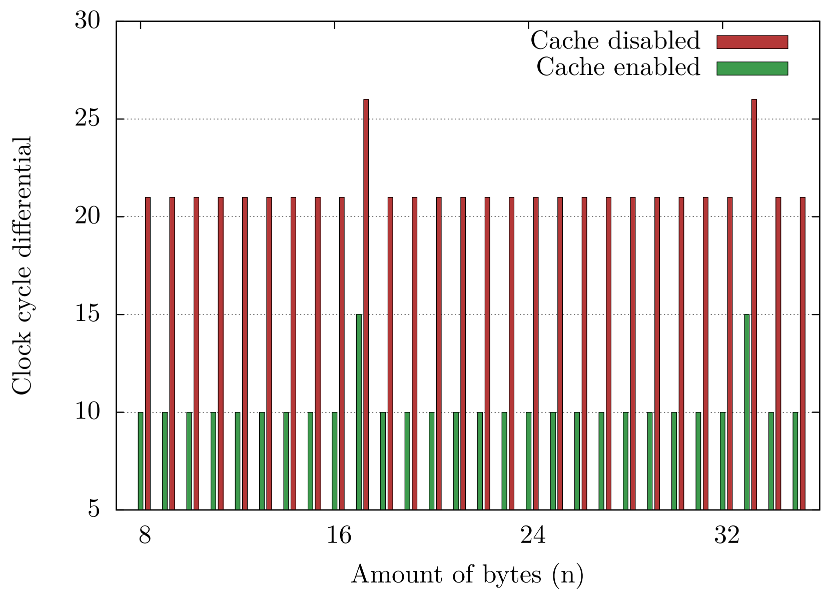


Figure 3.2: Different execution times in dependence of copied bytes n

count n and the subsequent invocation with byte count $n + 1$. Intuitively speaking, this gives us for every n the amount of clock cycles that the run additionally takes compared to its previous run. To not confuse issues, we ran our tests both with instruction and data caches enabled and later on again with all caches disabled. On actual hardware, the timings we measured are shown in the plot in Fig. 3.2.

What can be seen quickly is that the cache does have an effect on the total number of clock cycles, but it does not have an effect on the clock cycle differential. With all caches enabled, each new byte takes 10 more clock cycles except for the crossing of a 16-bytes boundary where this additional byte takes 15 clock cycles. For the case in which caches have been disabled, the same effect can be observed with the exception that the differential is usually 21 clock cycles and jumps to 26 clock cycles when crossing a 16-bytes boundary. The difference (5 clock cycles) is a direct consequence of the system's flash ROM wait states.

3.3.2 Architectural Modeling

To try to predict the timing accurately, we developed a behavioral ARM core emulator. In the beginning, we briefly looked into the option of modifying already existing emulation code (such as QEMU), but it soon became apparent to us that most already existing code was written predominantly with performance in mind. Such code would likely have been difficult to turn into something that was usable for cycle-accurate simulation and we, therefore, developed our emulator from the ground up in the C programming language.

To systematically determine execution time of code on hardware, we wrote platform evaluation code that allowed us to download code dynamically into the MCU's SRAM and have it execute with enabled ETM/DWT instrumentation. The result was a semi-automatic process in which the instructions which were of the greatest interest to us were evaluated regarding their runtime performance. We omitted modeling of instructions which are not relevant for cryptographic purposes in general and for our use case in particular. Those were, among others, all floating-point unit (FPU) instructions the MCU offers. The runtime information was collected by a host PC which was attached to our evaluation platform via RS232.

In our semi-automatic modeling process, the act of combining already evaluated instructions to form more complex code fragments was taken care of by a Python program. For all instructions which were not modeled, initially, an execution time of zero was assumed — something that is deliberately wrong. We then randomly generated valid code snippets using a Python program which emitted increasingly complex sequences of instructions. These randomly generated code fragments were executed within the emulator and compared against the results returned by the actual hardware over the RS232 connection. In the first training stage, it only emitted single instructions with no memory access, and that could not fail (for example, division instructions were not used in this stage since they can produce arithmetic errors when the dividend is zero). At a later time, we added memory-transfer operations; afterward, we added even more complex snippets of code in which conditional branch instructions occur. For the last group, we gave our code generator a coarse framework in which the branching instructions were to be embedded to avoid infinite looping or other undesired, undefined behavior.

Every produced code snipped was automatically generated, compiled and run locally by our emulator. As described, it was simultaneously downloaded on an STM32F4 and its execution was profiled on the real hardware. Whenever a discrepancy between the simulation and hardware arose, the code generator stripped the examples down to a minimal code fragment that still exhibited the issue. This allowed the developer in charge of modeling the device behavior to be able to pinpoint erroneous instruction emulation precisely and update the model accordingly. By this process, we were able to achieve convergence towards an accurate model which simulated standard cryptographic code (i.e., code that makes heavy use of bitwise Boolean arithmetic such as that in the AES or the SHA families) within just a few days.

The whole emulator is around 12000 lines of code (LOC). The greatest part of this, however, is taken up by instruction decoding (around 6000 LOC), while simulation core is around 1800 LOC long. Of the 233 non-FPU opcode variants, we emulate 114, i.e., around 49 %. The instruction decoding code is generated using a self-written Python code generator from an XML architecture description. This description, in turn, was transcribed by us from the ARMv7-M Architecture Reference Manual (ARM Ltd. 2014).

A simplified model of the emulator architecture we developed is shown in Fig. 3.3. The code under test is compiled into an ELF binary first, and the relevant data is extracted afterward using standard tools into a binary that could be written into the flash ROM

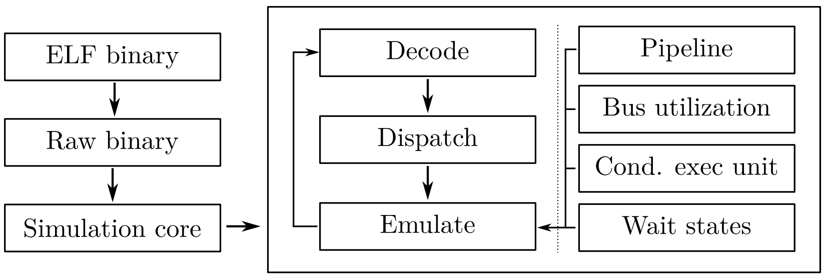


Figure 3.3: Model of our Thumb-2 emulator

of a microcontroller. This binary file is then fed to the simulation core together with some metadata information. Such metadata could be a particular entry point or register configuration to suspend and later resume emulated execution. When a single instruction simulation step occurs, decoding and dispatching are performed by the simulation core to the particular unit responsible for that opcode variant. Since the decoding stub knows about specific data types and their respective encoding (e.g., immediate Thumb expansion or sign extension as explained in the ARMv7-M Reference Manual (ARM Ltd. 2014)) the handler functions can work on the already decoded data and do not need to care about specifics of their argument encoding.

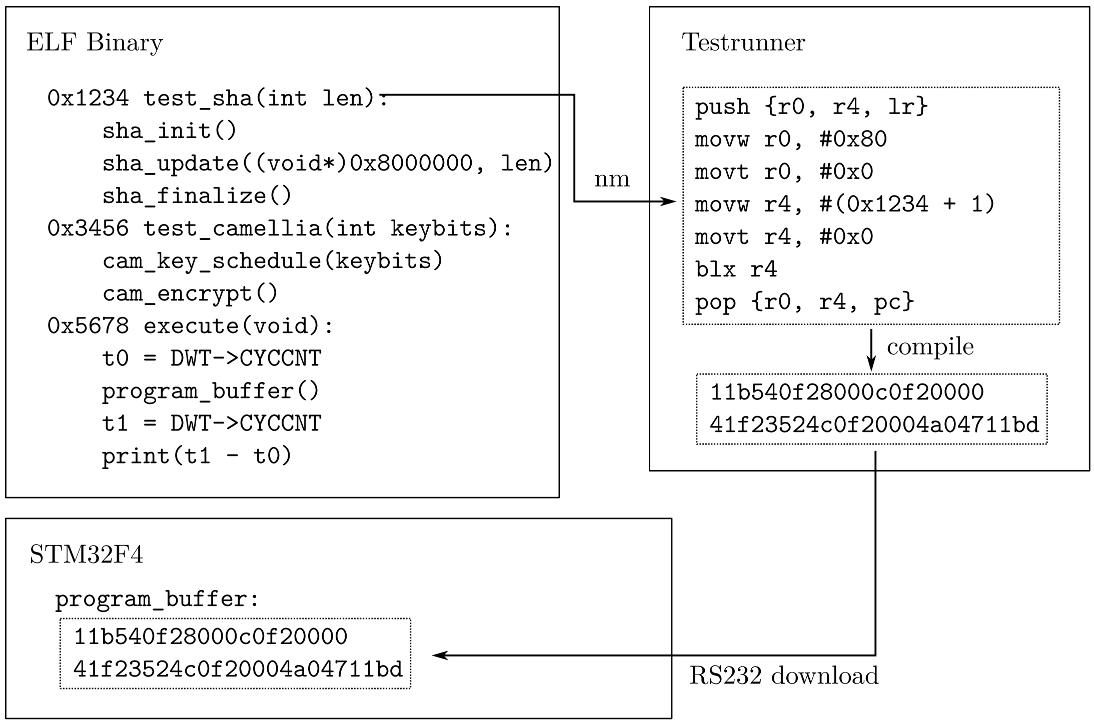
That way, a particular decoded instruction is emulated while relying on a behavioral model of the architecture in the background. A global state is held which takes into account the aspects of the system previously described in Sect. 3.3, such as memory wait states, bus utilization, pipeline fill and the state of the conditional execution unit of the CPU. All this impacts the actual runtime of the operation within the execution and — depending on the performed operation — possibly updates the internal state again to reflect the changed state.

3.4 Evaluation

To build up confidence in the accuracy of our emulator, we checked regularly during development that the model mimicked its physical counterpart carefully. The way in which we used training code was explained previously in Sect. 3.3.2. In this section, we highlight the tests we conducted against sophisticated, real-world software to find out how closely our emulator can predict runtime in these non-artificial pieces of software.

3.4.1 Experimental Setup

For our tests, we first created a monolithic ELF binary which contained all algorithms we wanted to test; these included public-domain variants of the AES and Camellia block ciphers as well as a public domain SHA256 implementation. We also evaluated the official Keccak-compact reference code in its version 3.2. Above those, some minor examples



```
0x1234 test_sha(int len):
    sha_init()
    sha_update((void*)0x8000000, len)
    sha_finalize()
0x3456 test_camellia(int keybits):
    cam_key_schedule(keybits)
    cam_encrypt()
0x5678 execute(void):
    t0 = DWT->CYCCNT
    program_buffer()
    t1 = DWT->CYCCNT
    print(t1 - t0)
```

```
push {r0, r4, lr}
movw r0, #0x80
movt r0, #0x0
movw r4, #(0x1234 + 1)
movt r4, #0x0
blx r4
pop {r0, r4, pc}
```

```
11b540f28000c0f20000
41f23524c0f20004a04711bd
```

```
program_buffer:
11b540f28000c0f20000
41f23524c0f20004a04711bd
```

Figure 3.4: Work flow in the experimental setup

(`memcpy` and `sprintf`) were also added which used the embedded C library (newlib in our case). For all test code, small test stubs were written which invoked the respective functions. For example, the SHA256 testing function does the initialization of a SHA256 context, updates the context with argument-defined data length (we used the internal ROM as a data source in this case and just varied the length) and finalizes the context afterward.

The main program then waits for commands on the RS232 interface, which was hooked up to a host PC. One offered command was to store received data into an application buffer, which was located in SRAM. Another command then invoked code execution of this program buffer and timed its execution time using the hardware-provided facilities described in Sect. 3.3, notably List. 3.3. This allowed us maximal flexibility because we were able to execute arbitrary code and therefore have custom setup and tear down trampolines without having to re-flash the microcontroller every time. Instead, we compiled some code which called the functions we wanted to test on the host machine. For this process, we wrote another Python program which scanned to ELF binary for relevant entry symbols using `nm`, emitted Thumb-2 assembly code, compiled this code and sent it to the microcontroller's program buffer via RS232.

The whole process is illustrated in Fig. 3.4: At first, the monolithic ELF binary is compiled and flashed onto the microcontroller. From the ELF, we extract the address of a particular testing function which we would like to call using `nm`; in the shown case, this is

Group name	Description	Examples
`addsub`	Arithmetic addition and subtraction variants	`add`, `adc`, `sub`, `sbc`, `rsb`
`bcc`	Branch on condition code	any conditional branches as well as compare-then-branch instructions like the "compare branch if nonzero" instruction `cbnz`
`bitwise`	Bitwise operations	`and`, `bic`, `eor`, `orr`, `neg`
`ldr`	Family of load operations	`ldr`, `ldrb`, `ldrd`, `ldrsh`, `pop`, `ldmia`
`mov`	Family of move operations	`mov`, `movt`, `movs`, `movw`
`shift`	Bitwise shifting instructions	`lsl`, `lsr`
`str`	Family of load operations	`str`, `strb`, `strd`, `push`, `stmia`

Table 3.1: Used instruction grouping

the `test_sha` function at address `0x1234`. The trampoline stub is then generated which initializes the first parameter to `0x80` (i.e., hash 128 bytes using SHA256). Afterward, the register `r4` is set to `0x1235` (the least significant bit is always set, indicating to the processor that it is to use Thumb mode). Finally, a register indirect call is performed which jumps into flash ROM. This code is compiled on the host and its binary stream sent to STM32's SRAM program buffer. Execution of this program buffer now times the function (which is located in flash ROM) with the previously determined arguments.

By using empty testing functions stubs which merely returned, we could determine the amount of time spent in setup and tear down of our trampoline function to later subtract that amount of clock cycles from the measurements. This gave us the number of clock cycles which depended solely on the test function. All code snippets by default ran 512 times on the actual hardware. Our testing program ensured that all timing results were in agreement to avoid accidental mismeasurements.

3.4.2 Experimental Results

To verify the correct operation of our emulator, we performed tests using a variety of functions, many of which were of cryptographic nature. Among those were encrypting single blocks using the block ciphers AES128 and Camellia and hashing using the SHA256 and Keccak hash functions. Some other tests did non-cryptographic work; one example performs a call to `memcmp` and yet another issues a `sprintf` call which prints a format string of 24 bytes containing two integer substitutions (%d and %u).

The result of these tests is shown in Tab. 3.2. Dynamic instruction count (i.e., the real number of instructions executed at runtime) is shown as well as the amount of clock cycles on the real hardware and the predicted amount of clock cycles of the emulator. The host, an Intel Core i7-5930K, took a worst case of about 300 clock cycles to emulate one target clock cycle. There are, however, significant differences in speed depending on

		Clock cycles				
Operation	Insns.	Native	Predicted	Diff.	Emulator	Ratio
`memcmp`-32	295	413	413	0	75 k	182
`memcmp`-100	907	1254	1254	0	227 k	181
AES128-16	8479	11634	11634	0	3.00 M	258
Camellia-128-16	1647	4469	4469	0	699 k	156
Camellia-192-16	2239	5962	5962	0	944 k	158
Camellia-256-16	2197	5892	5892	0	933 k	158
SHA-256-16	4091	5198	5198	0	1.53 M	294
SHA-256-32	4155	5299	5299	0	1.56 M	294
SHA-256-50	5227	5417	5417	0	1.59 M	294
Keccak-512-32	25866	40330	40326	4	8.19 M	203
Keccak-512-64	25938	40438	40430	8	8.22 M	203
Keccak-512-256	26370	41107	41075	32	8.37 M	204
`sprintf`	1088	2027	2010	17	367 k	181

Table 3.2: Measurements showing the dynamic instruction count (Insns.) and taken clock cycles on the target and in the emulator. Block ciphers were used for encryption, suffix denotes the payload in bytes.

Operation/Group	AES-128 (16)		SHA-256 (32)		sprintf	
addsub	1290	15.2 %	999	24.1 %	160	14.8 %
`b`	144	1.7 %	87	2.1 %	28	2.6 %
bcc	515	6.1 %	227	5.5 %	158	14.6 %
bitwise	2116	25.0 %	944	22.7 %	15	1.4 %
`clz`					1	0.1 %
`cmp`	512	6.0 %	218	5.2 %	154	14.2 %
`it`	298	3.5 %			49	4.5 %
ldr	1671	19.7 %	579	13.9 %	196	18.1 %
mov	498	5.9 %	833	20.1 %	140	12.9 %
shift	160	1.9 %	43	1.0 %	18	1.7 %
str	535	6.3 %	223	5.4 %	152	14.0 %
`tbh`					2	0.2 %
`tst`	298	3.5 %			1	0.1 %
`umull`					8	0.7 %
`uxtb`	442	5.2 %				

Table 3.3: Detailed dynamic instruction breakdown of tests with assembly instruction grouping

the type of code that is simulated. This can be explained mainly by code making use of the barrel shifter, something that's common in cryptographic computations and costly to emulate in software.

For most cryptographic algorithms, our emulator correctly estimated the amount of taken native clock cycles. This is unsurprising since accurate modeling of cryptographic code was our primary objective, and hence we put the most emphasis on that problem. We did not model instructions which are seldom used in cryptographic with complete accuracy. In particular, the division instruction which is needed for the Keccak-call (which uses the `sdiv` opcode) causes some slight discrepancies, and similar issues arise at the `sprintf` example.

An estimate of how different the constitution of cryptographic and non-cryptographic code can be is given in Tab. 3.3. In the AES128, SHA256 and `sprintf` examples we mentioned above we counted 108 unique opcodes that were executed. This includes different conditional variants of the same opcode after the occurrence of the Thumb-2 `it` "if-then" opcode as well as opcodes in their wide and narrow form. To do meaningful analysis with them, we grouped some of them into different categories as shown in Tab. 3.1. It is immediately obvious that the `sprintf` example differs significantly and uses instructions and code which is not required for cryptographic computations.

3.4.3 Semi-automatic vetting

To embed the emulator into a vetting process, we wrote a fuzzer using Python. A necessary prerequisite for functions which shall be tested is that they are runnable without any previous initialization right after a call to `main()`. If this is not possible, the developer can additionally define initialization functions which are called before executing the actual tests.

As can be seen in List. 3.5 the implementation of a fuzzing directive is done by implementing the method of a class. The test case generates two random byte arrays of 16 bytes each. A call to the `assert_crt` then actually triggers the verification. In this instance, `memcmp` is asserted to have constant runtime independent of the content of the random byte arrays.

```python
def tc_memcmp(self):
    array1 = self.randbytes(16)
    array2 = self.randbytes(16)
    self.assert_crt("memcmp", array1.addr, array2.addr, 16)
```

Listing 3.5: Python fuzzer test case

To come to a conclusion whether the assertion is true or false, the framework first takes the compiled ELF binary and simulates the code until `main()` is called. A snapshot of the memory and the CPU state is then generated as an optimization to be able to switch back to this state later quickly on for subsequent trials. The random byte arrays are generated at this point and mapped to a memory region which is otherwise unused by the hardware as not to interfere with the normal runtime behavior. Then, `memcmp` is called with the appropriate memory addresses. The cycle count of the runtime is recorded, and the previous memory snapshot is restored. When the procedure is run again, it is verified that the second cycle count is equal to the first one. An arbitrary number of runs can be specified to get a reasonable amount of confidence that the probed function actually exhibits constant runtime behavior. For the above example, a set of 1200 runs ensures with a probability of at least 99 % that the first byte in both random arrays was identical at least once. It would be equally possible to hard-code this, however, in Python, but we chose to keep the fuzzing example as simple as possible to illustrate our main point.

Integration of this Python fuzzer into the build process is trivial; in our case, we simply added a `.PHONY` target called `check` into the `Makefile` which initiated the vetting procedure. If this target is defined as a dependency of, for example, the programming target, it can easily be assured that programming the microcontroller only then proceeds once the internal checks have passed (since the build process would abort otherwise).

If a discrepancy in the tested code arises, the user can re-run the emulator with tracing enabled. This gives a detailed breakdown at every executed instruction. It shows what the elapsed cycle count is and what all register values are and therefore allows to locate easily the sections which caused incorrect cycle count predictions.

3.5 Conclusion and Outlook

We have described theoretically and shown in practice that modern microcontrollers, such as those of the ARM Cortex-M family, exhibit timing phenomena which closely resemble behavior previously only seen on sophisticated desktop CPUs. In our explanations, it becomes clear how difficult it is to predict these effects to the naked eye. While the reasons for which these mechanisms have been incorporated into modern MCUs are beneficial for performance, they can lead to the presence of timing side channels. We explained how easy it is to exploit these tiny timing discrepancies by using mid-range commercial-off-the-shelf equipment. The described attacks are realistic for attackers with minimal hardware knowledge and physical access to the device.

To strengthen the defensive side, we have developed an emulator with the primary objective of emulating code with clock cycle accuracy. We have demonstrated the effectiveness of our approach and also describe the outline of a streamlined process that improves the simulation model. With the help of this emulator, it is possible for a software developer to regularly and semi-automatically probe code after each compilation to achieve continuous quality monitoring during the development process. It only has to be defined what results are expected from the function under test for the fuzzer to be able to do its work properly. If for example, by an upgrade of the compiler, the timing behavior changes in a critical manner, this can be detected at an early stage within the development life cycle; timing analysis can then selectively performed on real hardware to confirm and pinpoint such irregularities.

Modern microcontrollers today are much more advanced than MCUs of previous generations. While this is a blessing, on one hand, these intricate performance boosters are a curse at the same time, since they are a major contributing factor to the presence of timing side channels. It is our hope that by accurately describing the effects present in these systems as well as releasing the complete source code of our project, our work contributes to both raising awareness and strengthening of mitigation strategies of timing side channels on embedded systems.

Chapter 4

Covert Channels

Involuntary transmission of data is commonly referred to as leaking via a *side channel.* In contrast, if malicious software or hardware deliberately uses certain hardware constraints to transmit information, this is referred to as a *covert channel.*

In this chapter, we present a new class of intra-packet covert channels which can be created on common hardware, but that cannot be detected by such. Our idea is to abuse anti-EMI features of a microcontroller to create a covert channel on the physical layer. Thus, the sender uses the invariants in how digital signals are encoded over analog channels to covertly transport information. This leaked data is present on the wire-bound connections of the compromised device, but is also by definition present in the vicinity of the device and can be picked up by radio equipment. As the covert channel is present only on the physical layer, the data on all layers above, as well as the timing behavior on those layers is indistinguishable from uncompromised devices. We present two example implementations of such channels using RS-232 as the carrier and use a common oscilloscope to decode the resulting covert channel. Using this setup, we observed symbol rates of around 5 baud. We derive the theoretical upper bound of the covert channel's bandwidth and discuss the factors by which it is influenced.

This chapter has been previously published in the proceedings of the International Symposium on Hardware Oriented Security and Trust as a poster and short paper (Bauer et al. 2016a); an extended version of the published paper was made available as a technical report (Bauer et al. 2016b).

4.1 Introduction

On the lowest layer of the OSI model, data is transmitted over a physical medium like a wire. To do this, the source data is encoded into physical parameters of the medium such as voltages or currents and thus comprises a signal $f(t)$. Any physical medium is subject to *noise* which can be modeled as an additive component to f. This means that the receiver does not receive the pure signal $f(t)$ but rather $f(t) + g(t)$ where g models the noise. Tolerance to noise on the physical layer is achieved by certain tolerance levels implemented by the interpretation function at the receiver. So, if $\mathcal{D}$ denotes the function which translates the analog signal on the physical layer to code words on the data link layer, this function satisfies the following equation for error-free transmission:

$$\mathcal{D}(f(t) + g(t)) = \mathcal{D}(f(t)) \tag{4.1}$$

As an example, consider the transmission of a single byte using RS-232. Note that we selected RS-232 because the simplicity of the protocol allows for a precise presentation of our core ideas. Protocols like USB, SPI or I^2C would work equally well. In Fig. 4.1, ten RS-232 symbols are transmitted on the physical wire: one start bit, 8 data bits, and one stop bit. Since the bit order is least significant bit (LSB) first, in the example the value `0xb1` is transmitted. Instead of letting involuntary noise $g(t)$ act upon the signal we could similarly specifically craft a function that would slightly change the signal in a way that would not alter the outcome of the interpretation function $\mathcal{D}$. Examples for this could be marginally lower or higher voltages, slightly faster or slower transitions between the Low and High states or a small phase shift of the signal. Two of these examples are shown in Fig. 4.2. The signal is sampled at the indicated points in time. Variations of the transition speed or signal phase do not matter as long as the signal

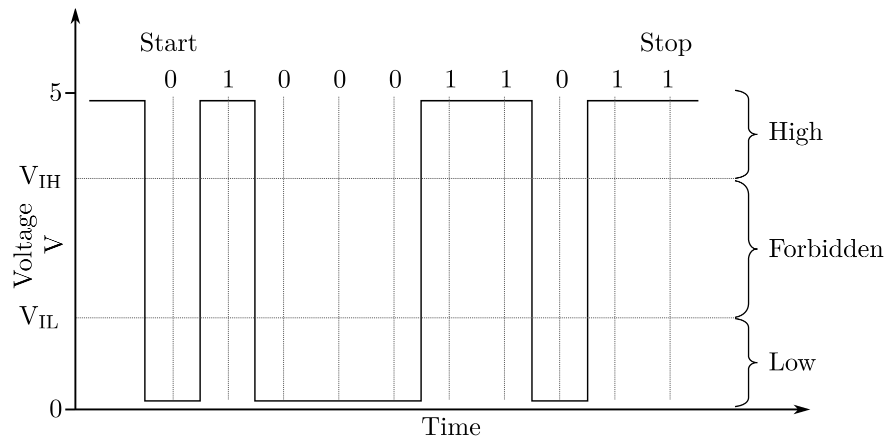


Figure 4.1: Exemplary RS-232 transmission of a single octet

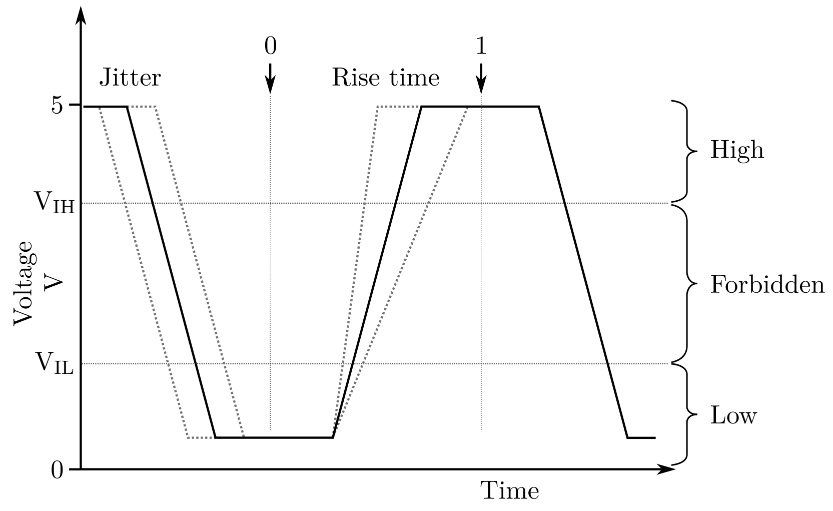


Figure 4.2: Signal changes of jitter/rise time that preserve the physical property

has the correct value at its intended sampling point. This means that for a LOW value, the voltage must remain below the threshold V_{IL} and for a HIGH value, it must stay above V_{IH}. The interpretation $\mathcal{D}$ is guaranteed to remain identical and thus the physical property is preserved.

We make the following observation in this chapter: If $g(t)$ is misused to encode secret information by slight variations in voltage or timing while ensuring that $\mathcal{D}(f(t)+g(t)) = \mathcal{D}(f(t))$, then there is no easy way for the standard receiver to decode or even detect this information. However, if $g(t)$ *can* be measured by a *specialized receiver* with an interpretation function $\mathcal{D}'$ (such as an oscilloscope with custom recovery algorithms), it is possible to extract the information while from a data link point of view, there is no observable difference between the modified and unmodified signals. An attacker who knows the exact signal deviations caused by the implanted covert channel could easily build a specialized receiver with dedicated hardware. Such hardware could be an FPGA development board for which the total hardware cost of the receiver would be somewhere in the range of around \$100.

4.1.1 Attacker Scenario: Covert Communication

Consider an attacker who wishes to access secrets covertly that are inserted into sensitive security hardware (like a tamper-resistant key store) *after* it has been deployed. To achieve his goal, the attacker acquires access to the supply chain between a silicon manufacturer and the original equipment manufacturer (OEM), as illustrated in Fig. 4.3. Within the supply chain, the attacker can intercept and modify the hardware and later can get *close* to the hardware in the field to access the secrets without actually having physical access to it.

Clearly, an attacker with physical access to a device has many possibilities to create backdoors. We, however, assume that (1) the attacker can only physically access and modify the device once, and (2) the modified devices are subject to intensive security checks by the OEM before deployment. Therefore, covertness cannot be achieved by classical hardware backdoors such as the approach by King et al. (2008) shows, and the extraction of information from hardware by physical access Gruhn and Müller (2013) and Halderman et al. (2009) is not an option. Note that such attack scenarios are not uncommon in practice (Appelbaum, Horchert, and Stöcker 2013), and in common end-of-line test bedding environment scenarios run by OEMs (such as a bed of nails test fixture), the focus is only on the *digital* semantic correctness of the devices under test. A covert channel in the way we describe in this chapter would pass such a digital test effortlessly, even when probed for with more sophisticated methods like fuzzing.

We demonstrate that the logic that is necessary to perform such an attack is minimal — in fact, many modern devices already have an abundance of possible circuitry on board which would allow deployment of such a channel. To demonstrate that little hardware modification is needed we show that the already present circuitry in off-the-shelf hardware

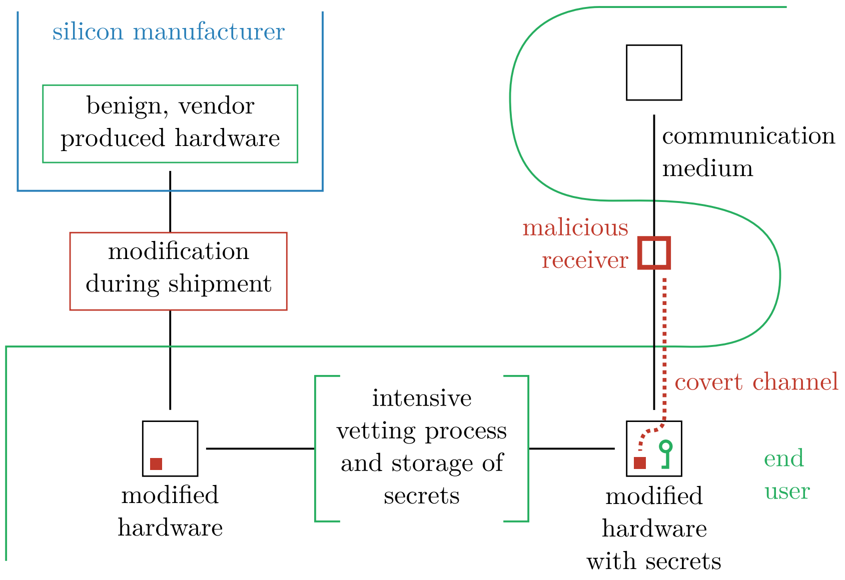

silicon manufacturer
benign, vendor produced hardware
modification during shipment
modified hardware
intensive vetting process and storage of secrets
modified hardware with secrets
communication medium
malicious receiver
covert channel
end user

Figure 4.3: Example of supply chain poisoning

is completely sufficient to construct a covert channel with it by doing only modifications of software (i.e., the firmware).

4.1.2 Abuse of Anti-EMI Features

In this chapter, we show that anti-EMI functionality can be misused to implement a covert channel and want to raise awareness of such threats. *Electromagnetic interference* (EMI) is an unwanted and inconvenient side effect which every electronic device exhibits. Governmental regulations limit the maximum amount of emitted EMI, and so the signal processing logic of many electronic devices contains suppressing techniques such as *Spread Spectrum Clocking* (SSC) or *Rise Time Control* to reduce the EMI emission. Our idea is to use such facilities as a covert communication medium. Both options are available on standard microprocessors today and can be used by software to reduce radiated emission. This allows us to create a covert channel with the following properties:

1. because it can be realized in software, *sending* information on the channel is easy and can be performed with a wide range of commercial off-the-shelf hardware, and
2. *receiving* and decoding the information requires specialized measurement equipment such as oscilloscopes or custom hardware. This renders the channel invisible to an observer on the data link layer.

4.1.3 Related Work

The notion of covert channels goes back to Lampson (1973) when he distinguished *timing channels* and *resource channels*. Timing channels encode information in the inter-packet timing delay, while resource channels use packet ordering or the state of a packet to transport information. Later, Kemmerer (1983) generalized the notion to scenarios with any form of shared resources. While the concept evolved in military contexts, today, the threats of covert communication have reached the mainstream and various implementations based on many communication methods, such as IP, exist as research prototypes such as that of Giffin et al. (2003), Murdoch and Lewis (2005), and Rowland (1997) and "in the wild" as demonstrated by Appelbaum, Horchert, and Stöcker (2013) and Dietrich et al. (2011). While the specific encoding and methods for creating covert channels were refined over the years, the actual implementation almost always focuses on protocol layers at or above the data link layer. This is the case with the works of Ji et al. (2009), Y. Liu et al. (2009), Murdoch (2007), and Wendzel and Keller (2012). Consequently, defensive methods, i.e., attempts to detect a covert channel, as shown by Cabuk, Brodley, and Shields (2004), Gianvecchio and H. Wang (2007), Moskowitz and Kang (1994), and Zander, Armitage, and Branch (2007) usually assume a packet abstraction such as the one provided by the Internet Protocol. In contrast, our work is completely independent of such an abstraction.

Works that focuses on hardware Trojans, such as those of Farag, Lerner, and Patterson (2012), Iakymchuk, Nikodem, and Kepa (2011), King et al. (2008), Shah and Blaze (2009), and Tehranipoor and Koushanfar (2010), make use of physical properties of hardware or communication on the data link layer to implement covert communication. They embed malicious circuitry in FPGA targets and therefore augment the present hardware to create backdoors. In contrast, our approach can be implemented using functionality that is already present in MCUs and can often be achieved by only modifying the firmware. Moreover, existing work is detectable by observing the digital behavior of the circuit.

Iakymchuk, Nikodem, and Kepa (2011) use heat dissipation to implement a covert channel; their work can be regarded close to ours, but their proof of concept implementation also requires hardware modification by altering the FPGA netlist. Shah and Blaze (2009) use the physical properties of the transport medium to implement a covert channel and encode covert information by selectively disrupting the physical carrier. Their attack, however, requires special radio frequency equipment, i.e., specialized sending and receiving hardware since a deliberate carrier disturbance is not possible with benign hardware.

Interesting observations are presented by Genkin, Pachmanov, Pipman, and Tromer (2015); they analyzed low-cost methods of key recovery in systems which exhibit electromagnetic side channel emission by using a Software Defined Radio (SDR). Their recovery approach could be applied to recovery of our EM covert channel as well.

4.1.4 Contributions

In this chapter, which is the extended version of the presentation at HOST 2016 (Bauer et al. 2016a), we make the following contributions:

- We introduce a new class of covert channels that uses *sub-digital* means to transport information, i.e., it abuses the degrees of freedom in the representation of digital signals on physical channels. In a sense, instead of looking at the *inter*-packet delay, our covert channels modify timing properties *within* packets, i.e., we use *intra*-packet timing channels. To the best of our knowledge, we are not aware of any other work that formulates and demonstrates this idea.
- We argue that these channels pose a relevant threat by showing that they can be *implemented* easily in *software*. We demonstrate this by using anti-EMI features that are supported by many commercial off-the-shelf processors. Such channels have an *asymmetry* property in that it is easy to send information over the channel but it requires special hardware to decode the covertly transmitted information. This, in turn, means that deliberate, targeted effort is required that specifically looks at aspects of the signal to detect the presence of such a covert channel. Our proof-of-concept example implementation demonstrates two different ways to encode information on RS-232 as the carrier protocol.

Since we exploit sub-digital features, the attacker necessarily needs either physical access to the compromised medium over which the information is sent or close physical proximity to the device. This is because our covert channel by definition is present only in the analog/digital encoding ambiguity and as a side effect in the form of parasitic electromagnetic emission. A standard receiver or relay (for example a network switch in the case of Ethernet) destroys the covert signal because it digitally interprets and reconstructs passed on signals. This means an attacker has to deploy a decoding unit somewhere within the network (for example by intercepting the wire) or use radio frequency equipment in the vicinity of the compromised device. We argue that this is a necessary downside of this new type of covert channel.

Note that for our example implementation we chose RS-232 for convenience only. Our approach could likewise be applied to many other carrier protocols, including but not limited to I^2C, SPI, I^2S or USB (Motorola, Inc. 2003; NXP Semiconductors N.V. 2014b; Philips Semiconductors 1997). Even though RS-232 has largely disappeared from the desktop computing environment, it is still widely used in embedded environments as a means of chip-to-chip communication. The fact that the output driver configuration is independent of the selected function of the MCUs port pin means that all peripheral functions on that port pin are affected by our channel. Our second approach affects the main system clock of the microcontroller (and because all peripheral clocks are derived from that clock source) directly leads to the consequence that *all* output peripherals are affected in the same manner as the RS-232 transport. This includes all peripherals that are supported by the used MCU. An attacker only needs to be able to monitor at least one affected channel to recover the covertly transmitted data. That these properties are independent of the used communication protocol makes our approach versatile.

4.1.5 Outline

This chapter is structured as follows: In Sect. 4.2, we discuss the signal theoretical background information necessary to understand the used carrier signal and also describe the anti-EMI mechanisms that modern microcontrollers employ. We continue by demonstrating two concrete covert channels in Sects. 4.3.1 and 4.3.2 which we implement with a Cortex-M4 microcontroller (STMicroelectronics N.V. 2011b). Sect. 4.4 gives a brief overview of how the transmitted symbols are encoded in our case and elaborates on the theoretic maximum channel capacity. Then we continue to explain how our channel can be applied to a real-world scenario in Sect. 4.5. Finally, Sect. 4.6 gives a summary and an outlook on what future work could be based on these methods.

4.2 Background

The way our physical layer covert channel works is closely entangled with the underlying electrical foundation and signal theoretical background. We, therefore, give a brief primer on electromagnetic interference and signal composition in Sect. 4.2.1 first. Afterward, we highlight common countermeasures to limit this — usually unwanted — EMI in Sect. 4.2.2.

4.2.1 Electromagnetic Interference

Digital square wave signals are composed of superimposed sine waves of different frequencies and amplitudes. Any square wave signal can be decomposed into its components using the Fourier transformation, as shown by Bracewell (1999). For a square wave with n harmonics, i.e., n superimposed integer multiples of the fundamental frequency, the signal amplitude at a point in time φ is given by

$$f(\varphi) = \sum_{i=0}^{n} \frac{1}{2i+1} \sin((2i+1)\varphi). \tag{4.2}$$

Note that the second derivative of this function $d\varphi$ is

$$f''(\varphi) = -\sum_{i=0}^{n} (2i+1) \sin((2i+1)\varphi). \tag{4.3}$$

By solving $f'' = 0$, one can see that the function has an inflection point at $\varphi = 0$, which yields $f'(0) = n + 1$. This means, the maximum slope of the composed digital signal, $n + 1$, is directly related to the number of contained harmonics. Thus, an ideal square

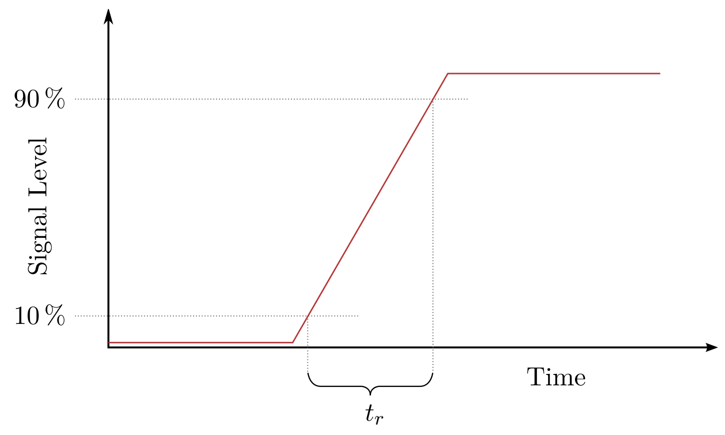


Figure 4.4: Illustrated rise time of a signal

wave signal has a positive edge slope of infinity and consequently contains an infinite number of harmonics.

Real-world digital signals cannot change instantaneously, and the transit from the Low to the High state or vice versa takes a certain amount of time. We refer to this time as the *rise time* when a signal does a Low to High transition and *fall time* in the inverse case. Since the following argumentation applies to both rise and fall times analogously, we only discuss rise times in detail. By convention, the measured rise time begins when the signal has reached 10 % of its peak value and ends when it is at the 90 % mark. This is illustrated in Fig. 4.4.

The edge that constitutes the Low to High transition also has a certain *slope*, usually measured in voltage per unit of time. A steeper slope intuitively corresponds to a shorter rise time and vice versa. We refer to both interchangeably. The steeper the slope of a digital signal is, the more harmonics are contained within the signal. Additionally, part of the dissipated power of signals is emitted in the form of *electromagnetic interference* (EMI). Intuitively spoken, this means that the device involuntarily acts as a radio transmitter. This effect becomes especially significant at high frequencies. Any signal with steeper slope only increases EMI though the presence of additional high-frequency harmonics but does not benefit the actual data transmission.

4.2.2 EMI Countermeasures

As explained in Sect. 4.2.1, signal slopes ideally are only as steep as they need to be by the application requirements. For example, if the data rate that an output driver uses is relatively slow (for example Full Speed USB with 12 MBit/s), a steeper edge slope of the signal does not make transmission of data any faster because the signal steepness is not at all the limiting factor. Higher rise times do, however, add additional radiated emission to the device. It is, therefore, advantageous to limit the slope of the signal to the amount that is necessary by the constraints of the connected peripheral.

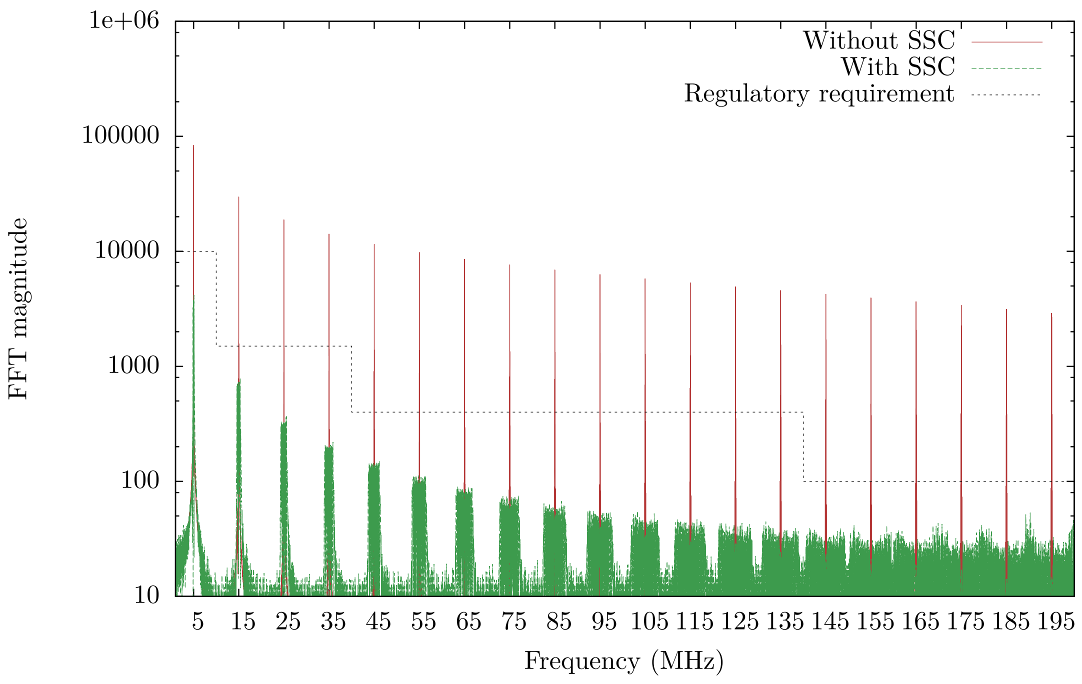


Figure 4.5: FFT magnitude of a 5 MHz clock signal with and without Spread Spectrum Clocking and exemplary regulatory permissible maximum regarding EMI

Apart from the actual rise time of the signal, two other factors influence the emission spectrum of any device: One is the clock frequency which also directly affects the frequencies and amplitudes of all contained harmonics. The other is the antenna efficiency of the parasitic antenna that is constituted by the integrated circuit itself. Antenna efficiency is highly nonlinear over the frequency spectrum. To reduce unwanted electromagnetic emission, one could theoretically change any of these factors. While changing the characteristic of the parasitic antenna is not possible in software, changing the clock frequency easily is. The main idea of *spread spectrum clocking* is, therefore, to vary the clock frequency periodically to *smear* the emitted spectrum. Although this means that the emitted energy (i.e., the integral of the power over the frequency spectrum) stays nearly the same, spectral peaks which could cause trouble in an EMI examination can easily be avoided, as Fig. 4.5 shows.

For spread spectrum clocking, two parameters influence how the clock frequency f changes over time: The period at which a frequency is modulated is called P and the associated modulation frequency is called f_m. The amplitude of the modulation is referred to as the *modulation depth* and is abbreviated with d. The example in Fig. 4.6 uses triangular clock modulation.

Microcontroller manufacturers are aware of countermeasures to EMI issues and have therefore equipped many of the newer, faster devices with the possibility to limit the rise and fall times of digital signals by use of special configuration registers (STMicroelectronics N.V. 2011b). According to the needs of the application, the analog properties of the signal can be modified within certain limits.

For implementing our covert channels, we chose a general-purpose microcontroller of the ARM Cortex-M family, the STM32F407VG (ibid.). Chips within that family are priced starting from \$2 up to around \$20, depending on the equipped amount of peripherals they contain. The STM32F407 is approximately the middle of that price range. They are built into a variety of embedded devices such as automotive ECUs (Electronic Control Units), RFID readers and consumer appliances like wireless routers, printers or air conditioners (Sadasivan 2006).

On the STM32F407, the rise time of GPIOs can be controlled in software by accessing the registers in the GPIO I/O map. Even if memory protection would be enabled using the MPU, in many scenarios at least some of these registers are accessible to unprivileged

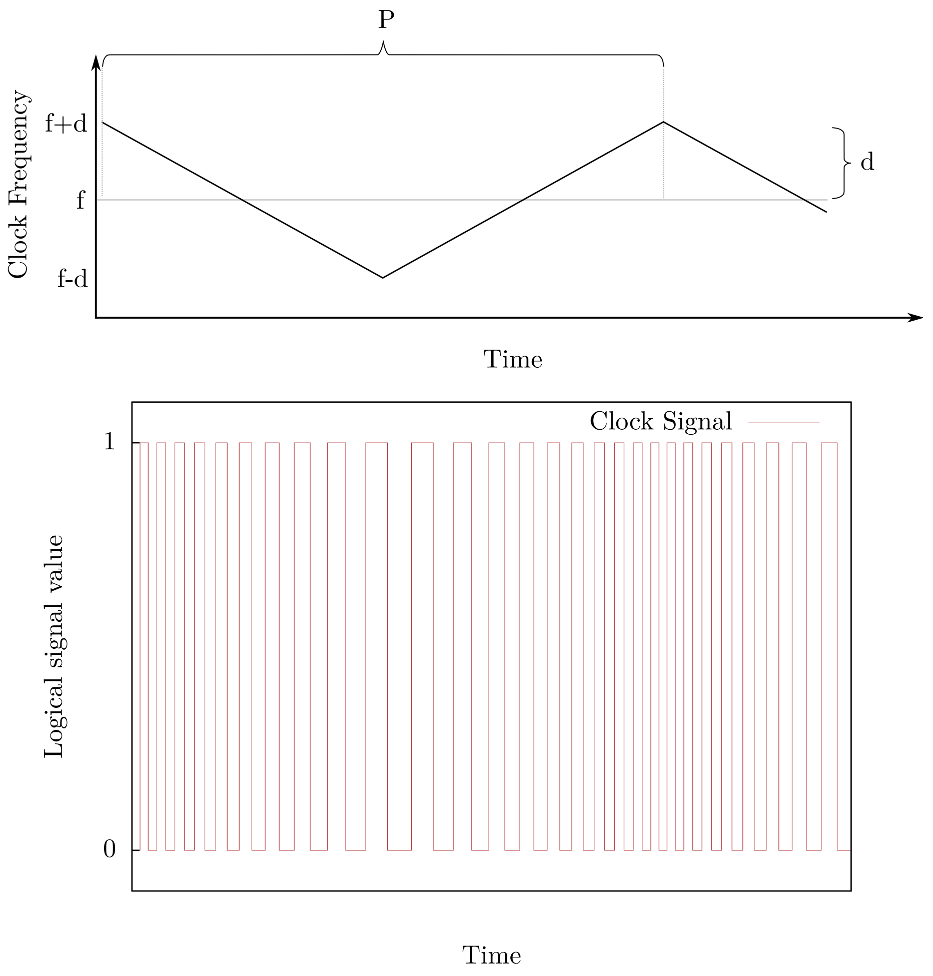


Figure 4.6: Frequency of a spread-spectrum clock signal

application software. Internally, the hardware implementation which allows control of rise time in four different levels — as it is possible with the STM32F4 — only needs to consist of two switchable parallel capacitors at the output driver. If both capacitors are switched off (disconnected), the highest rise time is achieved. When both capacitors are switched on, they act as low-pass filters on the signal and therefore lower the slew rate.

SSC is also implemented on the STM32F407VG in exactly the fashion we show in Fig. 4.6; it is using triangular clock modulation. The MCU provides two main registers which control how the internal phase-locked loop is affected by spread spectrum clocking. These registers are the 15 bits wide `INCSTEP` and the 13 bits wide `MODPER` register. The former affects the modulation depth d while the latter affects the modulation period P. According to the absolute maximum ratings of the device, the modulation depth d must remain in between $0.25\,\% \leq d \leq 2\,\%$ while the modulation period P must always remain greater than 100μs. There are additional constraints, all of which are described in Sect. 5.3.11 of (STMicroelectronics N.V. 2012). In our case, for a PLL input frequency f_{PLL} of 8 MHz and a PLL numerator value PLLN = 336, there are 163 values (1 to 163) to vary the `INCSTEP` register in ways that are within the absolute maximum rating of the device; d ranges from $0.25\,\%$ to $1.49\,\%$ in this case.

4.3 Implementation of the Covert Channel

The anti-EMI facilities that microcontrollers provide are usually configurable in software. An application programmer must be able to decide how and when these mechanisms are enabled since some may have a detrimental effect on the overall performance of the system. For example, while a certain anti-EMI measure may improve electromagnetic emission, it could also simultaneously have a negative effect on the sampling precision of an analog-digital converter (ADC). Therefore, it makes sense to give the application programmer the ability to turn the anti-EMI features on and off at will. This means the chip itself is equipped with the functionality to influence directly how much EMI is emitted at any point in time — a fact that we exploit in the following sections to construct a covert channel.

4.3.1 Implementation using Spread Spectrum Clocking

We now show how to implement a covert channel using spread spectrum clocking (SSC). While SSC can arguably be implemented in lots of ways, the devices we looked at (the ST32F4xx family) provide means to modulate the clock with a triangle signal of up to 10 kHz and a modulation depth d of $0.25\,\%$ to a maximum of $2\,\%$ (ibid.). Either of both variables can be used to encode data. To achieve covertness, it is beneficial to choose the parameters so that the resulting signals look like they have been affected by naturally occurring clock jitter. For comparison, the STM32F4xx datasheet (ibid.) lists a typical peak-to-peak period jitter of $\pm$200ps which is always present while at 168 MHz a worst-case 2 % SSC modulation depth corresponds to a period anomaly of only around 119ps.

In our experiments, we used both variants: modulating the SSC modulation depth on one hand and modulating the modulation frequency on the other. We never changed both variables at once, but always kept one of the two constant. Therefore, if modulation depth was modulated, the modulation frequency was chosen to be fixed at 10 kHz. For data transmission using a variable modulation frequency, the modulation depth was constant at 0.25 %.

One drawback of choosing the SSC unit on the ST32F4xx family is that each change between states requires the internal phase-locked loop (PLL) to be shut off before the CPU allows modification of the SSC registers (STMicroelectronics N.V. 2011b). This is a comparatively lengthy procedure and takes approximately 170μs in our case. Shutting down the PLL requires the system clock to be switched to a different clock source; typically this is the much slower internal RC oscillator. This is an operation which would appear suspicious to someone monitoring a continuous stream of output data with an oscilloscope or similar measurement equipment.

The advantage of using SSC is that the covert information is encoded in what resembles ordinary jitter. Here, d directly corresponds to the amplitude of artificially generated jitter while f_m corresponds to the rate at which the jitter amplitude changes.

Recovery was performed with an Agilent DSO-X 3014A oscilloscope. We triggered on a rising edge of the carrier and dislocated the trigger point one bit length in time, effectively showing the jitter which was artificially generated by the SSC. This data was transmitted to a PC using the USBTMC (USB Test and Measurement Class) protocol. The achieved covert symbol rate was about 2 baud using constant 10 kHz modulation frequency and three distinct modulation depths (i.e., three distinct symbols).

The limiting factor for this type of covert channel is the long fixed time which is necessary by hardware constraints to switch from one state to another. For each such change, the PLL has to be stopped and restarted, which takes a fixed amount of time. Therefore, to covertly use the SSC unit for information transmission, the transmission speed (i.e., the number of state changes per unit of time) would have to be exceptionally low so that the time for the state change itself becomes negligible.

4.3.2 Implementation using Rise Time Control

We now show how to implement a covert channel using a completely different method than SSC, namely the rise time control of the microcontroller unit (MCU).

The STM32F407VG which we used provides a facility to modify the output speed of the general purpose input/output (GPIO) pins in four different speed categories: 2 MHz, 25 MHz, 50 MHz and 100 MHz. Of those four, the measured average rise times were 19ns, 4.3ns, 2.4ns and 2ns at Vcc = 3.3V. While three of these (19ns, 4.3ns, 2ns) are trivially

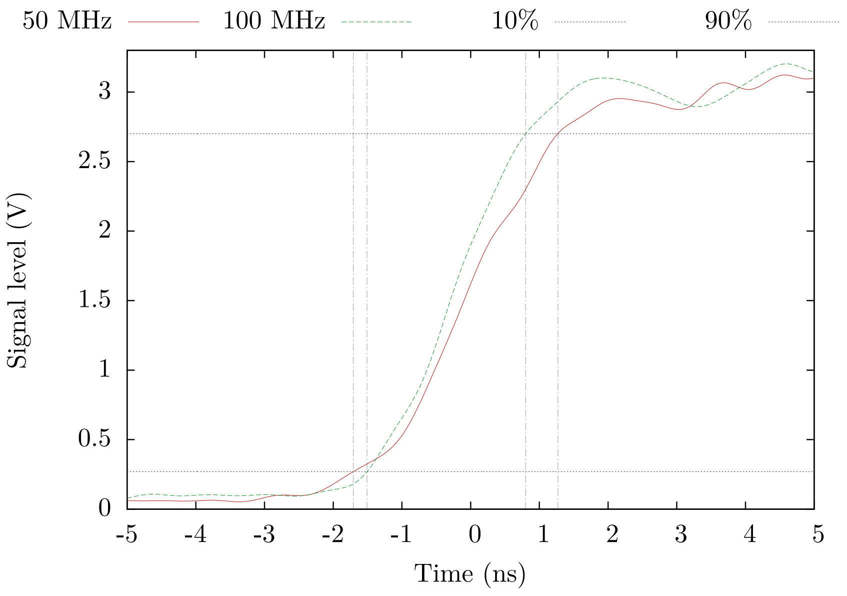


Figure 4.7: Signal differences of 50 MHz and 100 MHz drivers

distinguishable from each other, we wanted to stress covertness of our channel. This is why we chose to select only the 2.4ns and 2ns alternatives even though discriminating those options is technically most challenging.

Note that the selection of output driver speed does *not* affect the transmitted overt bit rate in any way; it only affects the slew rate of the signal and therefore the theoretically maximally achievable bandwidth using that output driver. For example, Full Speed USB uses a 12 MBit/s data channel. For this type of communication either one of the 25, 50 or 100 MHz drivers could be used with no observable difference in the overt communication. A plot that highlights the subtle differences in rise time is shown in Fig. 4.7. It was captured using a Tektronix MSO4034.

This channel has the advantage that state changes are exceptionally fast in software and that access to the required GPIO registers is usually allowed even to unprivileged software. That is, even when the chip's MPU is used, access to the relevant memory regions is usually permitted even to user space applications. Since the difference between the two fastest output driver states is marginal, the channel also exhibits an outstanding covertness property. Furthermore, it is versatile in the sense that it can apply to any output which does not require more than 50 MHz of signal bandwidth; such output peripherals could, for example, be USB, I^{2}C, SPI, or many others.

4.4 Data Encoding

Depending on the number of discrete states that a receiver can discriminate, an appropriate encoding can be chosen for transmission of data. Concretely, we used a ternary encoding

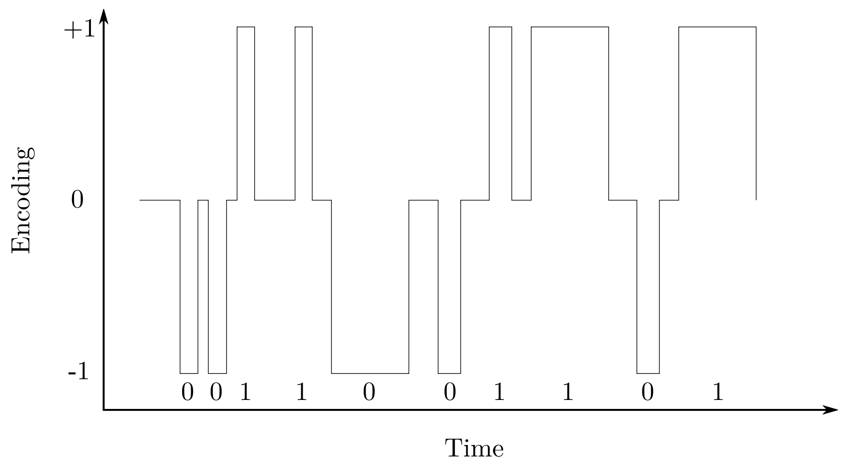


Figure 4.8: Ternary encoding of data bits with varying data clock rate

for the SSC variant described in Sect. 4.3.1 and a binary encoding for the rise time approach of Sect. 4.3.2. Note that the choice of encoding is orthogonal to the type of channel (i.e., SSC channel or rise time channel) itself.

The ternary variant allows for trivial clock recovery: With symbols $-1, 0, +1$, data bits are encoded by the transitions between the *idle* state 0 to and from either the -1 or $+1$ state. We furthermore require that only the transitions $-1 \leftrightarrow 0$ and $+1 \leftrightarrow 0$ are valid and all other symbol transitions constitute an invalid encoding.

Fig. 4.8 shows how this encoding looks in practice. Clock recovery is trivial: the only restriction is that the frequency of transmitted bits may not exceed the Nyquist frequency of the recovery unit's sampling rate—in other words, all three symbols must at least appear long enough to be reliably detected, but they can appear arbitrarily longer.

When using binary encoding, the clock recovery of the demodulated covert channel is more complex than with a ternary approach. This is because the actual data that is transmitted is intermixed with the data clock, i.e., the information at which point in time the data is valid. We relax the difficulty by assuming that the malicious code within the system is called at constant intervals, providing at least clock stability (yet at an unknown frequency) and that transmitted data is random. This is not an unreasonable assumption because leaked data usually is cryptographic material. Even with completely random data, however, there is often significant disparity within the signal which complicates clock recovery. To avoid this, a bit stuffing technique like 8B10B encoding introduced by Widmer and Franaszek (1983) could be used to keep signal disparity to a minimum. Alternatively, we could use whitening of the signal using an LFSR-based synchronous additive scrambler. The associated cost would in both cases be a significantly increased malicious code size, which is why we did not explore this further and assume that our transmission signal is already without relevant bit-bias. Randomly generated cryptographic keys should, in practice, exhibit no such bit-bias. An example of the actual measurement data that we did this type of clock recovery on can be seen in

	SSC	Rise time
Symbol Count (n)	3	2
Switch Time (t)	170 μs	120 ns
Overt symbol rate (b_o)	10 kBd	115.2 kBd
Covert symbol rate (b_c)	10 kBd	75.9 kBd

Table 4.1: Examples of channel parameters for our case

Fig. 4.9.

In summary, with these encodings, we achieved recovery speeds of about 1 bit/sec for the SSC variant (symbol rate of 2 baud, ternary encoding) and about 2.5 bit/sec for the GPIO variant (symbol rate of 5 symbols/sec, binary encoding).

4.4.1 Channel Capacity

After measuring the actual transmission speeds of our implementation, we now investigate the theoretically achievable maximum channel capacity. To calculate this, three main variables have to be taken into account:

1. Time it takes to switch between one output symbol to another (t)
2. Number of distinct, unique covert output symbols (n)
3. Effective baud rate b_c in dependence on the overt channel baud rate b_o

The two channels that we demonstrate experimentally in Sects. 4.3.1 and 4.3.2 use Spread Spectrum Clocking (SSC) on one hand and rise time encoding on the other hand as the covert transport. In the SSC example, we modulated the modulation depth d to create the channel. For the channels that we create, an overview of the values of these constants is shown in Tab. 4.1. The symbol switch time t is significantly greater in the SSC variant compared to the rise time variant due to the necessity of stopping the phase locked loop (PLL) to change the SSC registers of the ST32F407, as already explained in Sect. 4.3.1. It has also been explained there that the covert symbol rate depends not only on the overt symbol rate but also on the transmitted data. For our examples, we transmitted alphanumeric protocol data which gave us the shown values for b_c.

The reason why there is data dependence of b_c on b_o can be intuitively explained by the fact that the number of edges within the signal — and therefore the number of possibilities to inject covert data — depends on the data. For a worst-case word of `0x00` the number of symbols is minimal (two edges per byte of data) while for a constant stream of `0x55`, it is maximal (10 edges per byte of data). The plain text we transmitted had on average 6.6 edges per transmitted byte of data.

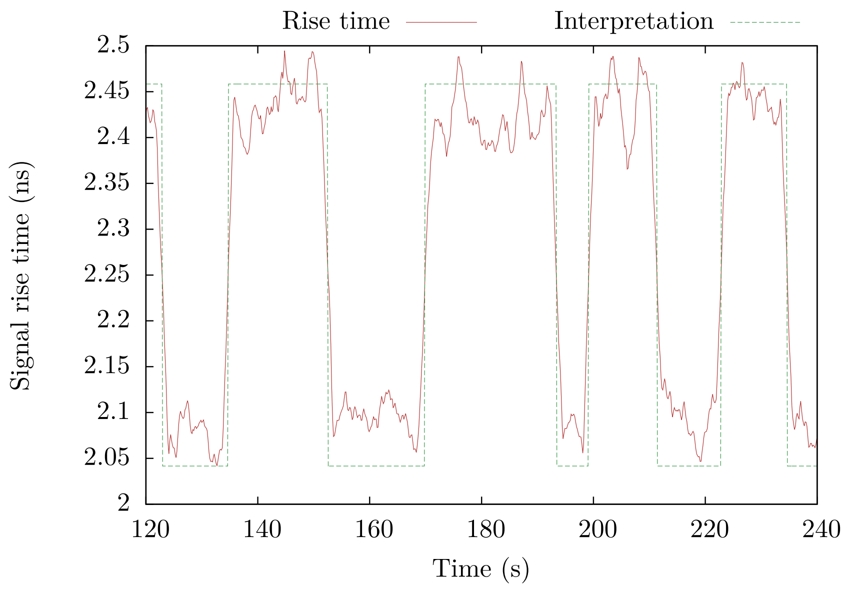


Figure 4.9: Rise time over time and the conditioned signal

For our calculations and to get a theoretical upper bound, we assume the best case of having at least one edge change per carrier symbol transmission. We then can derive the maximum channel capacity symbol rate as

$$B = \frac{\log_2 n}{t + b_c^{-1}}$$

We provide an exemplary calculation for the parameters chosen in our actual experiments, and we give an estimate for the theoretical maximum capacity in Tab. 4.2. In the experiments we send alphanumeric data over the RS-232 overt channel with an average of approximately 6.6 edges per transmitted byte, accounting for the lower b_c. For easy discriminability, we also limited the number of used symbols n significantly from the theoretical maximum.

	SSC		Rise Time	
Variable	Experiment	Theory	Experiment	Theory
n	3	193	2	4
t	170 μs	170 μs	120 ns	120 ns
b_c	10 kBd	10 kBd	75.9 kBd	115.2 kBd
B	5.9 kBd	28.1 kBd	75.2 kBd	227.3 kBd

Table 4.2: Theoretically achievable B under ideal conditions compared to conducted experimental evaluation

For our channel, the maximum b_c would be equal to b_o, i.e., 115.2 kBd, and we have four discrete symbols available. Therefore, we could achieve $B = 227.3$kBd. At first glance, it is counter-intuitive that the covert channel capacity could ever exceed the overt channel capacity. This has several reasons:

1. The symbol count of the covert channel can exceed those of the overt channel.
2. When the covert symbol rate depends on the transmitted data, we assume the best-case values. In our example, this means that a constant overt data stream of `0x55` would need to be sent—something that does not make sense in the real world.
3. Any computational power that is needed to control the covert channel is neglected.

In conclusion, while the constructed covert channel might theoretically have a large bandwidth, there are many practical aspects which decrease the practically achievable bandwidth by about five to six degrees of magnitude. Data that an attacker would want to leak through such a channel is in all likelihood cryptographic material that is exceptionally short. Therefore, the drawback of limited practical channel capacity is outweighed by the advantage that the channel itself is difficult to detect.

For our experiments, the factor that by far dominated the achieved covert bandwidth was not the transmitter, but the receiver. Since we relied on general-purpose equipment, the recovery bottleneck was the USBTMC transmission of the results to the decoding PC.

4.5 Practical Example

We now elaborate in depth on how an attack as motivated in Sect. 4.1.1 could look like and how it could be applied in a real-world example. As we stated before, our type of covert channel can be applied to every scenario where the output can be modulated in ways that are invariant for the digital interpretation function. The channels we talked about in-depth previously were wire-bound channels which misused the ability to influence involuntary electromagnetic emission.

Our idea can be explored further, however. In the realm of analog circuitry, it is quite common that fine-tuning of antenna circuits is done in software. For vendors of radio frequency equipment, this is rather convenient, since it enables them to fine-tune antennas to their circuits using a completely automatic process where the alternative would otherwise be the manual tuning of hardware components such as variable capacitors. Coincidentally, these framework conditions create precisely such an opportunity for the construction of a sub-digital covert channel as we have shown before. The only difference, in this case, is that the modulation is not performed via slight variations in rise time or signal phase but in the modulation of the radio frequency amplitude which is emitted via an antenna.

Consider a Radio Frequency Identification (RFID) reader that controls a door lock. Users get special RFID tokens which they can hold in front of the RFID reader. The reader performs a cryptographically secured handshake with the token and, upon successful verification that the token is legitimate, opens the door. Such RFID readers are manufactured by many companies, and they all usually provide the functionality that a user can set and store their cryptographic keys on the device. This is to ensure that the owner of the locking system is the only one who knows the keys and can issue new, genuine, tokens. We show that a vendor might have equipped such a product with a backdoor that allows gaining access to the system in an almost undetectable fashion by utilizing covert channels as described.

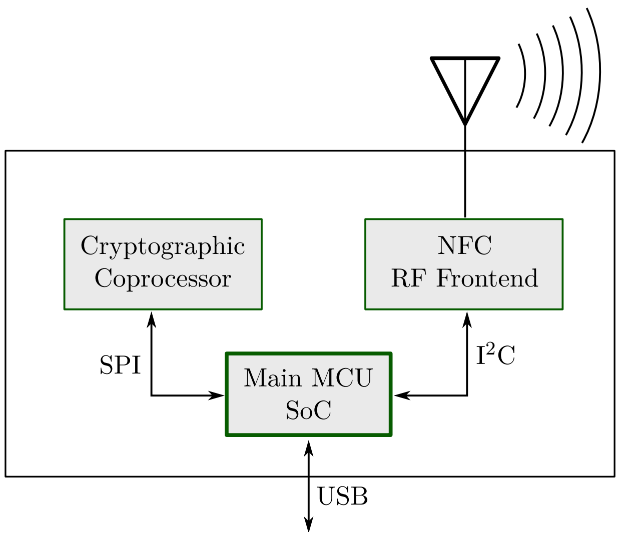


Figure 4.10: Notional RFID reader with its components

Consider such a notional RFID reader and its basic design which is shown in Fig. 4.10. The board consists of the main processor (SoC) that is connected to various peripherals using different buses. One is a cryptographic coprocessor which contains master secrets that are written into the chip as part of the commissioning of the reader board. Communication between the cryptographic coprocessor and SoC is encrypted and authenticated over a Serial Peripheral Interface (SPI). To communicate with near-field communication (NFC) tokens, a standard RF front-end IC is used that connects to the antenna. Successful authentication requests on the NFC interface are propagated over USB to an external lock control unit.

In such a system, multiple sub-digital covert channels could be existent: The most obvious one is a covert channel on the SPI. If the chip manufacturer embedded a sub-digital covert channel in the cryptographic coprocessor, then the coprocessor itself could be sending out its master secret periodically on the SPI, for example by using rise time modulation as we described before. These secrets are sufficiently short (256 bits) such that transmission can be extremely slow (in the range of 1 bit per minute to prevent detection) and still would allow an attacker to perform relatively fast recovery of the secrets (in around 4.5 hours). From an attacker's point of view, this channel is hard to exploit because it requires either physical access or high-end radio equipment.

If the firmware of the SoC itself were to be compromised, then multiple covert channels could be constructed. Again, the most obvious one is an onboard channel over any of the three connected interfaces. The manufacturer of the RFID reader would be able to steal keys of units in the field that have been parameterized by customers, but again physical access to the hardware itself or sophisticated equipment would be necessary.

The most interesting channel, however, is the one we hinted at in the introduction of this section: Usually, RF front-end ICs allow software tuning of antennas to compensate for small physical differences that occur during antenna manufacturing (NXP Semiconductors N.V. 2014a). During the production of the system, the antenna is connected to the RF front-end IC and software calibration is performed to achieve impedance matching between both components and therefore maximize field strength. This means that it is possible to attenuate the RF signal deliberately in software by introducing slight impedance mismatches and effectively modulate the field in a user-customizable fashion. Such a channel could leak keys that are required for successful authentication over the RF interface. An attacker who would have implanted this channel into an RFID reader would need little effort to perform channel data recovery.

An attacker who knows that the RFID reader is compromised could analyze the modulated RF stream externally using special receiving equipment (an oscilloscope would be sufficient for regular 13.56 MHz HF-RFID communication) and learn the keys that are required to authenticate against the reader, effectively creating a backdoor.

All these channels have in common that they are sub-digital in the sense that during test and auditing the equipment which is used cannot detect the presence of the covert channel because it is not part of the digital representation. Concretely, when using a logic analyzer to look at the transmitted bits, there is no visible difference between a system that is backdoored and one that is not, precisely because the channel itself is only present in the gray area of the encoding ambiguity of the analog interpretation of digital signals. Similarly, the covert channel in the RF case does not survive demodulation because the physical differences that constitute the covert signal are so small that the demodulator is indifferent towards them.

4.6 Conclusion

We have demonstrated the existence and feasibility of intra-packet physical layer covert channels. We implemented such channels on commodity hardware by abusing already present anti-EMI facilities. Concretely, we transmitted data via the Spread Spectrum Clocking unit and also showed that the approach works by modulating the rise time of an arbitrary output pin. Thus, we have demonstrated the practical feasibility of such attacks and by evaluating the channel capacity on real hardware, we have shown that transmission bandwidth is still high enough to be a relevant threat because it could be used to leak cryptographic material.

These channels by definition exist on any device which allows control over electromagnetic emission countermeasure facilities. Since the speed of microcontrollers has increased significantly in the last years, the necessity for the presence of such EMI countermeasure facilities has equally risen. Such countermeasure facilities are therefore already present in a wide variety of device by default as of today.

Something we wish to highlight is that data exfiltration is not merely limited to wire-bound tapping like we showed in our examples. It would be practical for a real-world attacker with more sophisticated equipment (such as a high-end spectrum analyzer) to pick up the covertly transmitted data remotely. Our rationale why this is possible is as plausible as it is catchy: The only reason these EMI-countermeasure facilities are present in modern microcontrollers in the first place is because they cause an externally observable difference in radio frequency electromagnetic emission. It is the prerequisite for them to be effective. This means in turn that any EMI-manipulation regardless of its physical carrier simultaneously causes subtle differences in the power levels within the EM spectrum. So even when wire-bound protocols are affected primarily, it is our estimate that remote, wireless data exfiltration is well within the arsenal available to a sophisticated attacker.

In future work, we wish to explore the possibility of purely wireless data exfiltration further. Another goal is to narrow the gap between the channel capacity we were able to achieve and the maximally achievable channel capacity by using more sophisticated equipment. This can be done by using special-purpose equipment such as an FPGA board in contrast to general-purpose equipment like an oscilloscope.

Furthermore, while we focused on wire-bound physical layer covert channels, wireless equivalents should warrant further research.

Finally, we have shown that for security auditing, it is insufficient to only examine the inter-packet timing characteristic and transmitted data on the data link layer. To detect physical layer covert channels, one has to dig deeper and take a look into the analog realm and *intra-packet* timing. The fact that such channels can be constructed easily and cheaply with off-the-shelf hardware means that they are even simpler to incorporate in custom hardware.

Because of their asymmetry property, they are invisible to standard receivers and therefore easy to miss. Since they pose a threat to confidentiality and system integrity, we hope that our work encourages further research in that area to reveal where such channels might already exist in today's real-world systems.

Chapter 5

Hardware Trust Anchors

Communication of resource-constrained nodes with backend servers is common in today's infrastructure. In marketing terms, these minimalist embedded devices are often described as *smart* systems. Such embedded nodes can be temperature sensors or actors such as heating regulators which communicate wirelessly. In the field of the smart grid, similar nodes are present at energy producers (e.g., wind turbines or solar collectors) or consumers (e.g., storage power stations). When all of these nodes communicate over the Internet, a well thought-out security foundation is imperative.

Traditionally in embedded system design custom protocols and ciphers have been used because cryptography was dictated by the resource limitations of the nodes themselves. As the nodes become more powerful regarding their computational capabilities, it is now possible to employ strong cryptography even in such constrained systems. With the use of well-designed cryptographic protocols such as Transport Layer Security (TLS) — even in small device nodes — attacks on the actual cryptography become difficult to the point of computational infeasibility. The focus of attackers, therefore, shifts to physical attacks to gain access to secret information.

In Chap. 2 we described how power analysis could be made more difficult by masking power emission using software techniques. While this is a valid approach and makes power analysis in practice significantly more difficult for an attacker, the drawback is that it defends only against non-invasive, passive attacks. There is nothing that can be done in software against invasive attacks such as physically opening the chip — a process called *decapping* — and directly connecting internal control lines with techniques such as *microprobing*. Depending on the required security level of the application it might, therefore, be advisable to incorporate a hardware trust anchor into the design of an embedded system to protect it against counterfeiting and the extraction of cryptographic material.

The level of protection with hardware security modules (HSMs) is higher than that of a general purpose microcontroller because the vendors have designed their hardware specifically to resist these types of attack. Generally, the special security ICs detect tampering and destroy any content upon detection of a physical attack. Additionally, during the chip design, obfuscation on hardware level is used to make it more difficult for an attacker to understand the working principle of the HSM, even when she opens up the IC package. Many high-security ICs also protect against passive attacks like DPA using appropriate hardware countermeasures.

While it sounds like HSMs would solve most security-related issues, this is unfortunately not true: In many scenarios, a protocol is given and needs to be implemented according to its specification. Therefore, smooth integration would require the interfaces of a specific HSM and the protocol's cryptographic algorithms to align. This is seldom the case; nearly all of third-party high-security ICs have proprietary interfaces. Therefore, the idea of incorporating HSMs into pre-existing infrastructure is often discarded early on because of the associated incompatibilities. It also is often not a viable solution to strongly couple cryptographic protocol design to a proprietary interface of a specific HSM. Since usually no second source for these custom chips is available, this would mean complete vendor lock-in and is something that is generally best avoided in the design of an embedded system.

In this chapter, we demonstrate one approach to integrating low-cost hardware security modules into pre-existing infrastructure. To this end, we chose the protocol constraints given by the Open Mobile Alliance Lightweight M2M protocol (OMA LWM2M) and use the interfaces of different chips of the Atmel HSM family.

An excerpt of the work of this chapter has been previously published as an extended abstract at DACH 2015 (Bauer and Freiling 2015).

5.1 Introduction

The Internet of Things (IoT) has picked up a tremendous pace in the last few years. With these added communication nodes and their pervasiveness, they become attractive targets to an attacker. Therefore, from a defensive side, the security requirements of these deeply embedded systems rise as well. Unique identification and authentication are considered state-of-the-art. This allows access revocation of nodes which are known to have been compromised, for example, in the way Fan, Haines, and Kulkarni (2014) demonstrate. Many of the popular protocols for machine-to-machine communication are in the dilemma that they need to perform well on low-cost resource-constraint embedded systems but nevertheless provide an adequate security level for the device. They still, therefore, rely heavily on symmetric cryptographic primitives because symmetric algorithms perform reasonably well on low-cost systems-on-chip (SoC). Brachmann et al. (2012) underline the importance of proper end-to-end encryption and take a look at the specific problems in respect to integration of resource-constrained clients in such networks.

For the security of secret key cryptosystems, the deciding factor for its security is whether or not the symmetric key can be kept secret from an attacker. Inherently to the system, this key needs to be present on both the client and server which want to perform communication. Since it needs to be shared before the first communication occurs, it is also referred to a *pre-shared key* or PSK for short. While it is relatively easy to store this PSK securely on the server backend, on the client side it often simply resides on the microcontroller's flash ROM. Thus, the main risk is that an attacker could extract this key either by passive methods such as power analysis or maybe even by invasive attacks such as physically opening the chip package (decapping of the IC). Both kinds are approaches within the capabilities of a skilled attacker (Skorobogatov 2004). This makes it all the more surprising that current work on secret storage of such pre-shared secrets, such as that of Bagci et al. (2012), explicitly excludes physical attacks in the attacker model.

Defending against physical attacks is nothing new, however. On the contrary, there exist numerous approaches which are used by silicon manufacturers to improve the resilience of specific hardware security modules (HSM; Mangard 2004). Among the features typically found in devices which have been hardened are tamper detection mechanisms such as light sensors or trip wire meshes. They are supposed to detect if the chip package has been physically opened and destroy all secret content upon detection of tampering. High-security integrated circuits are additionally protected against passive attacks like power analysis by using specific countermeasures which aim to hamper or prevent attackers (Bucci et al. 2004; Ravi, Raghunathan, and Chakradhar 2004).

While the use of such HSMs is encouraged for hardening against physical attacks, for example by Babar et al. (2011), often concrete and cost-efficient practical implementation hints are missing. Recently, however, low-cost commercial-off-the-shelf HSMs have become available. These HSMs are suitable for integration with resource-constrained embedded systems. One example of such a device is the Atmel ATSHA204A with a cost of around $0.50 per unit. The caveat, however, is that these HSMs usually provide proprietary interfaces which complicate integration into pre-existing, standardized protocols like TLS.

This chapter describes an efficient and generic approach how such an HSM can be integrated into the popular, state-of-the-art TLS protocol. We show the integration of the symmetric HSM Atmel ATSHA204A (Atmel Inc. 2015b) into the handshake of TLS-PSK and also describe how the asymmetric HSM Atmel ATECC508A (Atmel Inc. 2015a) can be used for integration of TLS with key exchange on elliptic curve basis.

With our contribution, the security model of Bagci et al. (2012) can therefore easily be augmented also explicitly to include physical attacks into the attacker model.

5.1.1 Related Work

The field of machine to machine communication is currently in rapid motion. Fan, Haines, and Kulkarni (2014) give a good overview of current approaches from a perspective that takes industrial requirements into account.

There is also some disagreement if standard cryptography is the best choice for resource limited systems such as the embedded nodes that M2M communication focuses on. Ukil, Bandyopadhyay, Bhattacharyya, and Pal (2013) and Ukil, Bandyopadhyay, Bhattacharyya, Pal, and Bose (2014) propose a different approach to authentication using a custom cryptographic protocol. They have used their custom security protocol for vehicle tracking, and they propose using it for a communication network of distributed nodes as well. Note that their cryptographic construction has not yet seen public scrutiny and not been cryptanalyzed (Ukil, Bandyopadhyay, Bhattacharyya, Pal, and Bose 2014). While Ukil, Bandyopadhyay, Bhattacharyya, and Pal (2013) and Ukil, Bandyopadhyay, Bhattacharyya, Pal, and Bose (2014) exclude hardware tampering attacks, this is one of the topics that we do address by the inclusion of hardware trust anchors.

Like others in the field of machine-to-machine communication, Raza, Shafagh, et al. (2013) also use the Constrained Application Protocol (CoAP) for the transport of data. As a security layer, they use DTLS. To fulfill the common requirement of having little overhead by the security layer, they demonstrate how header compression can make CoAP over DTLS more efficient (Raza, Trabalza, and Voigt 2012).

Bagci et al. (2012) take a look at how data can be stored securely by distributed decentralized nodes. They evaluate confidential data storage for wireless sensor networks. Their data storage system also relies on the fact that the system is not analyzed invasively by hardware attacks.

While it is uncommon in the field of sensor networks to contemplate physical attacks on the device nodes to extract cryptographic material, in the field of smart card research this topic got much attention. Messerges, Dabbish, and Sloan (2002) studied how well smart cards, of which should be expected that they provide some level of resilience against physical attacks, behaved when threatened by power analysis.

Constructively, Moore et al. (2002) describe an option to improve the security of smart cards by using self-timed circuits. Their design allows some means of timekeeping even though a typical smart card does not contain a power supply and can, therefore, not have a clock running when independent of its host.

Herbst, E. Oswald, and Mangard (2006) present a specific smart card implementation which offers resistance against power analysis attacks. Their approach studies 8-bit smart cards and they use masking and randomization of operations to achieve resistance against higher-order DPA attacks.

However, with increasing defensive mechanisms, the offensive side also equally gains momentum: E. Oswald et al. (2006) describe higher-order DPA attacks against masked block cipher constructions. They demonstrate their approach to be feasible by attacking an AES smart card.

5.1.2 Contributions

In this chapter, we describe the concrete integration of a real-world hardware trust anchors into the datagram TLS protocol. The distinguishing features of our approach are as follows:

1. We incorporate the cryptographic computation of a hardware trust anchor into the handshake of TLS. In particular, we do this without the need to modify the TLS protocol itself in an incompatible manner. This means that we consolidate two interfaces: the software API of TLS on one side and the hardware API which was given on the trust anchor side — neither of which are easily changeable.

2. We generalize on types of hardware security modules and show how similar integration can succeed even with modules which offer different internal cryptographic primitives.

We furthermore analyze the cryptographic constructions which we used in-depth and calculate the security margin that the devices offer.

5.1.3 Outline

The remainder of this chapter is structured as follows: First, we give an overview of the technical background in Sect. 5.2. For this purpose, we first explain the details of the transport layer security (TLS) protocol in Sect. 5.2.1. Afterward, we highlight the peculiarities of TLS when used with pre-shared keys in Sect. 5.2.2. To understand the usage of asymmetric primitives in conjunction with TLS, a primer of elliptic curve cryptography is given in Sect. 5.2.3. Afterward, we give a concrete example that showcases our idea; namely, the Lightweight M2M protocol and its security layer are introduced in Sect. 5.2.4. After this background, we focus on our main idea and show how to implement a secure key exchange using hardware security modules which only can use symmetric cryptography in Sect. 5.3. How more sophisticated hardware security modules, which rely on asymmetric cryptography, can be incorporated into the TLS handshake is shown in Sect. 5.4. An in-depth evaluation of the used security primitives in afterward given in Sect. 5.5 and we conclude with our final remarks in Sect. 5.6.

5.2 Background

Since we show incorporation of hardware security layers into the TLS handshake, the TLS protocol is now explained in detail. The necessary background of asymmetric elliptic curve cryptography is also provided along with the basic security constraints that are present in the Open Mobile Alliance Lightweight Machine-to-Machine protocol (OMA LWM2M).

5.2.1 Transport Layer Security

Transport Layer Security (TLS) is the de-facto standard protocol for transport encryption on the Internet. It has seen extensive peer-review and is regarded as state-of-the-art regarding its security. One limitation of TLS is that it requires a reliable connection such as TCP/IP (Dierks and Rescorla 2008). To use TLS over a connection in which packets may arrive in wrong order or get lost entirely, an adaption called datagram TLS (DTLS) was created (Rescorla and Modadugu 2006). For systems with minimal networking capabilities, this is convenient because it follows that no heavy-weight TCP/IP stack needs to be present on these systems, and instead the much simpler UDP/IP protocol can be used instead.

Since the computational power in modern microcontrollers has been steadily increasing over the last years, more applications switch from minimalist proprietary cryptographic constructions to a standard like DTLS. Security-wise this is beneficial because TLS stacks are readily available and, when correctly implemented, can be regarded as state-of-the-art without having to do security reviews on a per-case basis. TLS has varying degrees of freedom in the way in which it can be used, ranging from variants which only use little resources to complex integration with systems such as Kerberos. It offers plenty of cryptographic algorithms for encryption and authentication of data as well as for key exchange and key authentication.

As a standard notation, all relevant algorithms for a particular TLS session are concatenated together to form a text string which is called the *cipher suite*. A cipher suite typically includes the key agreement algorithm, the bulk encryption algorithm and pseudo-random function which is used. Not all cipher suites provide the same level of security; some cipher suites do provide forward secrecy while others do not, for example. Some bulk encryption algorithms are weak (for example DES or RC4) while others are strong (for example AES256). It is up to the developer to decide what level of security is acceptable and which cipher suites have a reasonably good performance for that required level of security.

During the TLS handshake, the client sends a list of all cipher suites it is willing to negotiate a connection with to the server. The server chooses one out of this list and determines which cipher suite is going to be used for the connection. If no agreeable cipher suites are found between client and server, the connection fails and is terminated. If the handshake is complete, all communication in that session is protected by encryption and authenticated using symmetric cryptography. This means an attacker can neither eavesdrop on exchanged data nor can she modify any traffic of a TLS session.

The standard mode in which TLS is used in the infrastructure of the Internet today is by using asymmetric cryptography based on X.509 certificates. Usually, only the server provides a certificate to the client. Then the client and server perform a key agreement algorithm such as Elliptic Curve Diffie-Hellman (ECDH) to derive the symmetric session keys. The drawback of such a key exchange is that it is relatively heavy-weight both regarding necessary communication and also regarding computational resources required to perform the key agreement. While the algorithms that are used once a session is established are comparatively lightweight, an initial handshake using asymmetric cryptography is therefore often not efficiently possible on small embedded systems. Fortunately, TLS offers the option to use exclusively symmetric cryptographic primitives in its pre-shared key modes.

5.2.2 TLS with Pre-shared Keys

The pre-shared key (PSK) cipher suites have been added to the Transport Layer Security (TLS) protocol in 2005 by RFC 4279 (Eronen and Tschofenig 2005). As the name suggests these cipher suites require that both the server and client know the same, pre-shared, symmetric key. This key agreement method is particularly popular for resource-constrained devices. Asymmetric cryptography like RSA and ECC internally use finite field arithmetic on arbitrary precision integers, which is particularly slow on small microcontrollers to the point of impracticality. While PSK does not offer the same security guarantees as key agreement algorithms like Diffie-Hellman (most importantly PSK sessions do not have forward secrecy), it has the advantage of using only functions which have comparatively good performance and are therefore suitable for deployment in embedded systems.

If a server is operating in PSK mode and has multiple clients which connect to it, it would be an unwise decision to simply share the same PSK among all clients. Effectively this would mean that one client would be able to decode all sniffed traffic of all other clients. Therefore, it is prudent that each client gets their own, unique, and random PSK. This, however, creates a new problem: During the establishment of the connection, the server needs to identify the client in some way to choose the correct PSK for that particular client.

For this purpose, the specification is designed such that in PSK operation, both the server and client reveal identifiers to their respective peers so each can perform a PSK lookup. Two identifiers are defined in the specification: one identifies the server and is called the *PSK identity hint* while the other identifies the client and is called the *PSK identity*. These identifiers typically are serial numbers or UUIDs. Both strings are completely opaque to the TLS handshake and conforming implementations only need to fulfill minimal requirements. Namely, all Unicode-printable strings up to at least 128 bytes shall be supported by conforming implementations. How the PSKs are determined after each party has knowledge of its peer's identifier is completely implementation-defined. The specification "does not specify how the server stores the keys and identities, or how exactly it finds the key corresponding to the identity it receives." (Eronen and Tschofenig 2005)

In a typical scenario, the server uses the PSK identity of the client to perform a database lookup for retrieval of the PSK. Embedded clients do not usually have a lot of different servers which they need to communicate with, so it is common that the client either completely ignores the server's PSK identity hint and always uses a hardcoded key or that it only selects one out of a handful of options programmatically. If on the client side the PSK identity hint is unused, the specification suggests to leave it empty altogether: "In the absence of an application profile specification specifying otherwise, servers SHOULD NOT provide an identity hint and clients MUST ignore the identity hint field." (ibid.)

The reason that the process to get a PSK from the peer's hint is left open by the TLS-PSK specification is that the authors do not want to limit the possibilities of data sources for PSK determination. In particular, this degree of freedom enables us to implement a TLS-PSK conforming interface while using a proprietary hardware security module for PSK lookup.

When the handshake of a PSK cipher suite is performed, the PSK identity hint is the first identification string that is exchanged. It is sent from the server to the client.

$$\underbrace{\texttt{00 05}}_{\text{Len}}\ \underbrace{\texttt{00 00 00 00 00}}_{\text{Other Secret}}\ \underbrace{\texttt{00 05}}_{\text{Len}}\ \underbrace{\texttt{aa bb cc dd ee}}_{\text{PSK}}$$

Figure 5.1: Construction of the TLS-PSK Premaster Secret

The client then responds with its PSK identity after having just learned with which server it is communicating. After the two identity messages have been exchanged, the server and client should be able to agree on the same pre-shared key. This pre-shared key is then padded according to RFC 4279 (Eronen and Tschofenig 2005) to form the so-called *premaster secret*. TLS allows for mixed operation of symmetric and asymmetric operation; this means that it provides the possibility to use, for example, a Diffie-Hellman-Exchange (Diffie and Hellman 1976) and use the PSK to authenticate the exchanged session keys. In the following, however, we consider only the plain symmetric case where no asymmetric cryptography is involved. For this, the premaster secret padding is performed according to Fig. 5.1. The length of the premaster secret can be determined from the PSK length. Note that the key shown in that example figure is short and is meant only to demonstrate the used format of the premaster generation. In general, its length is $l_{\text{PMS}} = 4 + 2l_{\text{PSK}}$ bytes. The so-called *other secret* is the part that would include data that results from asymmetric operations such as the Diffie-Hellman key exchange. For the plain PSK case, it is set to a constant zero pattern of the same length as the PSK.

The premaster secret is then further processed according to RFC 5246 (Dierks and Rescorla 2008) to yield the *master secret*. It is the calculation of the appropriate pseudo-random function with the following arguments:

- The `secret` is equal to the previously calculated premaster secret.
- The `label` is the constant string `"master secret"`.
- The `seed` is the concatenation of the ClientHello nonce and ServerHello nonce.

This master secret must always be 48 bytes (384 bits) in length. Firstly, an intermediate padding — called A — is specified. The definition of A is:

$$\begin{aligned} A_0 &= \texttt{seed} \\ A_i &= \text{HMAC}(\texttt{secret},\ A_{i-1}) \end{aligned}$$

This intermediate product A is then used to derive the function P which outputs an infinite bitstream:

$$\begin{aligned} P(\texttt{secret}, \texttt{seed}) = {} & \mathrm{HMAC}(\texttt{secret},\ A_1 \,||\, \texttt{seed}) + \\ & \mathrm{HMAC}(\texttt{secret},\ A_2 \,||\, \texttt{seed}) + \\ & \mathrm{HMAC}(\texttt{secret},\ A_3 \,||\, \texttt{seed}) +\ \ldots \end{aligned}$$

Ultimately leading to the full PRF used in TLS:

$$\mathrm{PRF}(\texttt{secret},\ \texttt{label},\ \texttt{seed}) = P(\texttt{secret},\ \texttt{label} \,||\, \texttt{seed})$$

Concretely, for cipher suites that use HMAC-SHA-256 (Krawczyk, Bellare, and Canetti 1997) as the TLS pseudo-random function, the whole derivation is shown in Fig. 5.2. For the keys which we use, the PSK is 32 bytes long and therefore the premaster secret is 68 bytes in total. This premaster secret then serves as the key to the HMAC engines which perform further key derivation. Concretely, four HMAC calculations are performed, all parameterized with the premaster secret as their respective key.

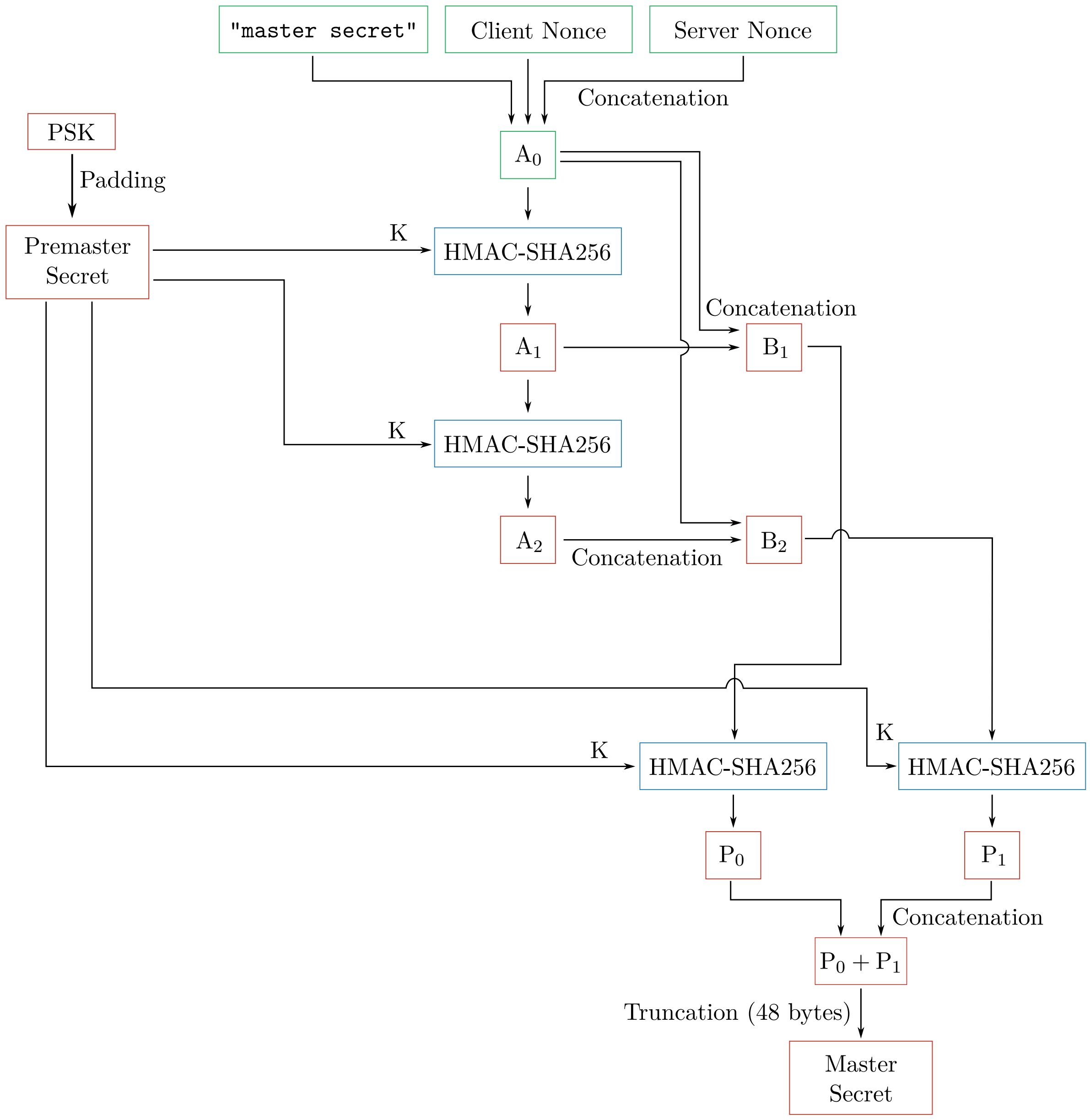


Figure 5.2: TLS-PSK master secret generation

5.2.3 Elliptic Curve Cryptography

Using elliptic curves for cryptographic purposes goes back to the proposals of Koblitz (1987) and V. Miller (1986). Independent of each other they both suggested how an elliptic curve over a finite field $\mathbb{F}$ could have cryptographic applications. In particular, such a curve consist of a number of points P

$$P = \begin{pmatrix} x \\ y \end{pmatrix} \qquad x, y \in \mathbb{F}$$

which need to fulfill the characteristic equation of the curve E given in *short Weierstrass form*:

$$E : y^2 = x^3 + ax + b \qquad a, b \in \mathbb{F}$$

Additionally, there is one distinct point $\mathcal{O}$ which is commonly referred to as the *point at infinity*:

$$\mathcal{O} = \begin{pmatrix} \infty \\ \infty \end{pmatrix}$$

For the underlying field $\mathbb{F}$, in the following we only consider finite prime fields, i.e., $\mathbb{F}_q$ with prime q. The discriminant Δ of this curve in $\mathbb{F}_q$ is defined to be: (Silverman 1986; Tate 1974)

$$\Delta = -16(4a^3 + 27b^2) \mod q$$

Such a curve is smooth and non-singular (i.e., *elliptic*) if the discriminant Δ of the curve is nonzero. As can easily be seen this is the case if and only if:

$$4a^3 + 27b^2 \neq 0 \mod q$$

On a smooth curve, a line connecting two points P and Q on the curve is guaranteed to intersect the curve in a third point R, where R might also be equal to $\mathcal{O}$. Based on this fact, V. Miller (1986) defined the result of the commutative and associative *point addition* $P + Q$ to be

$$P + Q = \begin{pmatrix} R_x \\ -R_y \end{pmatrix} \qquad P \neq Q$$

For this point addition, the point at infinity $\mathcal{O}$ serves as the identity element.

In the case of $P = Q$, the operation is called *point doubling*, and the geometric interpretation is that the tangent of the curve at the point P is intersected with the curve and gives a point S. The resulting point then is

$$P + P = \begin{pmatrix} S_x \\ -S_y \end{pmatrix}$$

Note that for a point P with $P_y = 0$, the tangent in P does not intersect the curve in an explicit point, but is rather said to intersect the curve at infinity:

$$Q + Q = \mathcal{O} \qquad \text{if } Q_y = 0$$

In summary, the following holds true for any points P, Q on the curve:

$$\begin{aligned} P + Q &= Q + P \\ P + \mathcal{O} &= P \end{aligned}$$

With point addition and point duplication we can now proceed to define *point scalar multiplication*. This means multiplying a scalar value k with a point P. The rules to perform this operation can be expressed by the following recurrence:

$$\begin{aligned} 0 \cdot P &= \mathcal{O} \\ k \cdot P &= P + (k-1) \cdot P \end{aligned}$$

Commonly, this is implemented using the double-and-add algorithm:

$$\begin{aligned} 0 \cdot P &= \mathcal{O} \\ 1 \cdot P &= P \\ k \cdot P &= \begin{cases} \frac{k}{2}(P+P) & \text{if } k \text{ is even} \\ P + (k-1) \cdot P & \text{if } k \text{ is odd} \end{cases} \end{aligned}$$

To be able to exchange meaningful information about points on curves, the curve has to be precisely defined. This definition is referred to as the *domain parameters* of the curve. For curves in finite fields over a prime q, these domain parameters are:

- q the prime that defines the finite prime field $\mathbb{F}_q$, i.e., all field arithmetic is performed modulo q.
- $a, b \in \mathbb{F}_q$ which are coefficients in the curve equation E.
- G, the *generator point*, is a distinct point on the curve that has been agreed on. Scalar multiplication of G forms a subgroup of E.
- n, the order of the subgroup formed by the generator point G. It follows that $n \cdot G = \mathcal{O}$.

- h, the cofactor of the curve. This is equal to the number of all points in $\mathbb{F}_q$ which fulfill the curve equation E, also referred to as $\#E(F_q)$, divided by n. Since n is the order of a subgroup of E, it follows from Lagrange's theorem that the cofactor is an integer.

Note that while n and h can be calculated from the other given domain parameters, it is usually provided along with the other parameters. This is because counting the points of the curve $\#E(F_q)$ is a computationally comparatively expensive operation. For a reasonably efficient, polynomial-time algorithm for point counting, Schoof (1985) presented a solution. This was later refined by Schoof (1995) to form the Schoof-Elkies-Atkin algorithm which has even better performance and is currently the state-of-the-art algorithm for counting points on elliptic curves.

With the previous definitions in mind and an agreement on specific domain parameters, one can define meaningful high-level cryptographic operations that are required to use ECC for signing, signature verification, and key agreement purposes. To use elliptic curve operations for signing and verification of data, the Elliptic Curve Digital Signature Algorithm or ECDSA (Johnson, Menezes, and Vanstone 2001) is used. In the context of ECDSA, a private key d is simply a randomly chosen scalar value modulo n. The corresponding public key Q, which is a point on the curve, can be calculated from the private key simply by performing scalar multiplication with the generator point of the curve:

$$Q = d \cdot G$$

To sign a message m using ECDSA with the private key d, the following algorithm is performed:

1. Calculate a hash function over m and convert the result to an integer e.
2. Randomly select a scalar k with $1 \leq k \leq n-1$
3. Compute $H = k \cdot G$
4. Compute $r = H_x \mod n$. If $r = 0$ then restart at 2.
5. Compute $s = \frac{e+dr}{k} \mod n$. If $s = 0$ then restart at 2.
6. The signature is the tuple (r, s)

To verify the signature (r, s) of a given message m against a public key Q using ECDSA:

1. Assert that $1 \leq r \leq n-1$ and $1 \leq s \leq n-1$. If not, reject the signature.
2. Calculate a hash function over m and convert the result to an integer e.

3. Perform the following calculations:
   $u_1 = \frac{e}{s} \mod n$
   $u_2 = \frac{r}{s} \mod n$
   $X = u_1 \cdot G + u_2 \cdot Q$
4. If $X = \mathcal{O}$, reject the signature.
5. Compute $v = X_x \mod n$
6. If $v \neq r$, reject the signature.
7. Otherwise, accept the signature as valid.

The last primitive that is needed to use ECC in the context of TLS is a key agreement protocol. TLS uses the Elliptic Curve Diffie-Hellman (ECDH). It is a transfer of the Diffie-Hellman key exchange (Diffie and Hellman 1976) to the ECC cryptosystem and was already suggested by Koblitz (1987) and V. Miller (1986) in their original ECC papers. A formal specification is also available (American National Standards Institute 2001; McGrew, Igoe, and Salter 2011). Sophisticated public key plausibility checks are of utmost importance for real-world implementations; such checks are explained in detail by Antipa et al. (2002) and Law et al. (2003).

ECDH uses the same semantic for public and private keys as ECDSA. Consider two parties which both have generated key pairs. One party has the private key d_1 with its corresponding public key $Q_1 = d_1 \cdot G$ and the other has the private key d_2 and a public key Q_2.

The ECDH algorithm is now performed viewed from the side of party 1 (i.e., in the beginning, only d_1 and Q_1 is known).

1. Receive the peer's public key Q_2. Verify its integrity by checking:
   - Assert that $Q_2 \neq \mathcal{O}$
   - Assert that $Q_{2_x} \in \mathbb{F}_q$ and $Q_{2_y} \in \mathbb{F}_q$
   - Assert that Q satisfies the curve equation (i.e., it is a point on the curve).
   - Assert that $n \cdot Q = \mathcal{O}$
2. Calculate $K = d_1 \cdot Q_2$. This is the shared point that can be used for derivation of a secret.

ECDH leads to the same point K for both communication parties because of the definition of Q_1, Q_2:

$$K = d_1 \cdot Q_2 = d_1 \cdot (d_2 \cdot G) = d_2 \cdot (d_1 \cdot G) = d_2 \cdot Q_1$$

This point K can then be used as input to other functions such as a bulk encryption algorithm or an authentication function like HMAC.

5.2.4 OMA Lightweight M2M

The OMA Lightweight M2M Specification (Open Mobile Alliance 2016) does not specify a custom security layer. Instead, it refers to and relies on DTLS, the datagram based version of TLS (Modadugu and Rescorla 2004; Rescorla and Modadugu 2006).

Concretely, the LWM2M protocol defines cipher suites that a conforming implementation must implement. On the client-side, only one needs to be implemented while the server side — usually backend systems with enough resources — are expected to handle all different options. The cipher suites of LWM2M which utilize pre-shared keys are:

- `TLS_PSK_WITH_AES_128_CCM_8` (McGrew and Bailey 2012): Key agreement via PSK (no forward secrecy), encryption with AES-128-CCM, authentication using CCM-AEAD with an 8 bytes authentication tag.
- `TLS_PSK_WITH_AES_128_CBC_SHA256` (Badra 2009): Key agreement via PSK (no forward secrecy), encryption via AES-128-CBC, authentication using HMAC-SHA256.

For asymmetric handshakes, the following cipher suites are conforming to the specification:

- `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` (Rescorla 2008): Key agreement via ephemeral ECDH (forward secrecy), exchanged keys signed with ECDSA, encryption with AES-128-CCM, authentication using CCM-AEAD with an 8 bytes authentication tag.
- `TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256` (ibid.): Key agreement via ephemeral ECDH (forward secrecy), exchanged keys signed with ECDSA, encryption via AES-128-CBC, authentication using HMAC-SHA256.

Therefore, the block cipher AES-128 is going to have to be present in any case, as is the SHA-256 hash function, which is implicitly used as the pseudorandom function for these TLS v1.2 cipher suites (and for some of them also performs authentication). If a symmetric HSM would, therefore, be able to perform HMAC-SHA-256, it could directly store the TLS premaster secret and be used for key derivation. Similarly, if an asymmetric HSM would perform ECDH and ECDSA, it could also be used as a transparent drop-in replacement. Unfortunately, the hardware we chose does not have an interface that allows for such trivial integration.

5.3 Implementation with Symmetric Cryptography HSMs

In Sect. 5.2.2 we laid out the cryptography which is used during the TLS-PSK handshake in detail. When taking a look at the master secret computation, as shown in Fig. 5.2, it is noticeable that the basic building block to derive the master secret from the premaster secret is the HMAC-SHA-256 computation. The premaster secret, in turn, is derived from the PSK by fairly trivial padding and length field concatenation. Therefore, if the module allowed for storage of an HMAC-SHA-256 key, one could simply safely store away the premaster secret and integration of the module would be completely transparent. For storage of a 32 bytes key, this would mean that a 14 bytes (112 bits) long PSK could effectively be used.

The caveat is that in practice, the available HSMs are usually not that flexible. Most commercial-off-the-shelf HSMs have a proprietary interface and therefore are not be directly compliant to the process. They usually do not directly support HMAC-SHA-256 in an unaltered (i.e., non-proprietary) manner.

One other solution would be to define a custom cipher suite which internally has a modified master secret derivation which fits the hardware interface. This cipher suite would need to be implemented on both the client and server side. However, such an approach would introduce a great disadvantage: firstly, such a cipher suite would be a completely non-standard, custom extension of the TLS protocol. Secondly, the TLS library itself would need to be modified to implement such a cipher suite. This means that only libraries can be modified of which the source code is available, and modification is permitted as by the license. Lastly, any such changes would have to be ported when the library is updated by the upstream vendor. Because the internal API does not need to be stable, this can be a time-consuming process.

Our basic idea is, therefore, different. We do not use the HSM for the derivation of the session keys or for the derivation of the premaster secret — we instead use it to derive the pre-shared key (PSK). The key which is stored within the HSM itself is, therefore, a pre-PSK (pPSK). This allows us to use all standard mechanisms of the TLS-PSK key derivation while still incorporating a proprietary key derivation mechanism. For this derivation to work, nonces need to be exchanged between the client and server. This is where the identity messages which we explained in Sect. 5.2.2 come into play: we use those identity messages to exchange nonces on a per-connection basis. While the TLS handshake itself does additionally use client and server nonces, these usually are not available from the application which links against a TLS library. In alignment with our previous reasoning of not wanting to modify the TLS library internals, we, therefore, opt simply to use additional nonces on top of the already existing ones.

Because the TLS library needs to perform a PSK lookup using an application-specific mechanism, TLS libraries, in general, provide a callback function for this purpose. A client side TLS-PSK callback function is called with the PSK identity hint of the server and is expected to return the PSK and the client PSK identity. This is a single, clean, interface which is present across all reasonably implemented TLS libraries in one form or another. Therefore, it is the ideal place to implement our custom pre-PSK derivation using an HSM there.

Fig. 5.3 shows an abstract representation of an HSM that can securely store symmetric keys. This means that uploading a key is possible, but this key cannot later be downloaded from the module. However, the key can be *used* via an external interface. Functions which

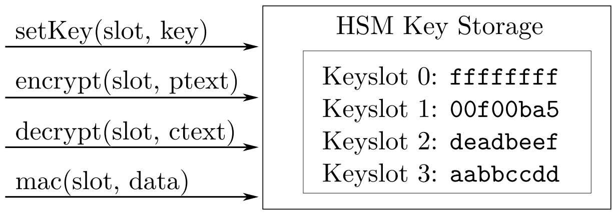


Figure 5.3: Basic layout of a symmetric HSM

rely on shared secrets to performing computation are, for example, en- or decryption operations or hashed message authentication codes (HMAC; Krawczyk, Bellare, and Canetti 1997). These keys can be referenced by their specific key slot identification number to be used by the module to perform certain tasks:

- `setKey`: Set a certain key slot to a value.
- `encrypt`: Encrypt a given plain text to a cipher text using previously stored key.
- `decrypt`: Decrypt a given cipher text to a plain text using previously stored key.
- `mac`: Perform a particular authentication function over a given message and return the MAC, parameterized with a previously stored key.

For our concrete implementation of TLS-PSK with a symmetric HSM, we chose the Atmel ATSHA204A module (Atmel Inc. 2015b). Among other features, it allows computation of a message authentication code (MAC) using an Atmel-proprietary but published input format. In principle, this function could be used to derive the session keys directly; this would, however, mean that the key exchange within TLS would become proprietary as well, and a new, custom, cipher suite would need to be defined for this. Since such an approach would cause all sorts of issues regarding compatibility, we chose a different approach:

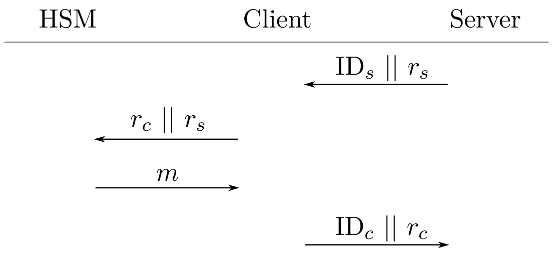


Figure 5.4: Key exchange using a symmetric HSM

In the context of TLS-PSK, we use the following construction: The server includes a random 16 bytes nonce r_s along with its server identity information ID_s in the "PSK identity hint" message. The client, which is in possession of the HSM, generates an own 16 bytes random nonce r_c — possibly, but not necessarily, with the help of the

HSM. Then both nonces are given to the HSM for it to perform a MAC computation using the pre-PSK. The resulting MAC value is used as a PSK. To enable the server also to calculate the effective PSK, the client relays r_c together with the client identity information ID_c within the “PSK identity” message to the server. The server can then, with the help of a lookup of the pre-PSK, reproduce the proprietary MAC computation used by the client and can therefore also derive the session key.

It is important to note that a module does not need to implement all of these functions. In most real-world devices there is also the capability to configure which keys are allowed to be used with which API functions. There are usually also mechanisms in place to prevent arbitrary function execution; for example, the hardware might always incorporate device-specific bits in the authentication function such as a serial number or the device configuration itself. It might also require authentication of some sort before allowing the API to perform any action at all.

A concrete instance of a symmetric hardware security module is the Atmel CryptoAuthentication ATSHA204A (Atmel Inc. 2015b). It is a small, relatively cheap ($\approx$ \$0.40 per unit in volume) device that provides safe key storage, hardware intrusion countermeasures and has a SHA-256-based authentication engine at its core. Even though the device is advertised as being capable of calculating HMAC-SHA-256 this is — probably due to hardware and cost constraints — only possible in a very restrictive manner: When the `HMAC` command is used, the most limiting factor is that the HMAC input data cannot be arbitrarily chosen, but has a fixed format that is not compatible with the way expected by TLS-PSK as explained in Sect. 5.2.2.

A newer version of the device (ATSHA204A) has an added `SHA` command which the older generation (ATSHA204) does not offer. However, it is also not possible to use this command to emulate HMAC-specific padding by hand. The user has to supply all data for operation in this mode, and it is not possible to load secret keys from the internal key storage. Since this is the only point of having the device in the first place, this options also drops out.

One thing the device offers, however, is a `MAC` command that computes the SHA-256 hash in a proprietary format over 88 bytes of data. Our idea that can be used to implement TLS-PSK with a device that allows such a proprietary computation is to use the command to *derive* a pre-shared key from a master key that is stored both on the device itself and on the server side. For the device to be effective, however, it is necessary to challenge the device every time a new session should be established. Since the challenge data in practice is only exchanged in the TLS protocol at the time the PSK callback occurs, another mechanism has to be used. In our implementation, we use the PSK Identity Hint and the PSK Identity for this purpose. Each side supplies 16 bytes (128 bits) of random data to its peer. The PSK that both sides then agree on is the result of the `MAC` command over the concatenation of both nonces. To generate random data with good entropy, the TRNG of the ATSHA204A is used in the process using the `Random` command.

The server and client both select a random nonce during the TLS handshake procedure.

```
0x00  4b 4b 4b 4b 4b 4b 4b 4b  4b 4b 4b 4b 4b 4b 4b 4b  |KKKKKKKKKKKKKKKK|
0x10  4b 4b 4b 4b 4b 4b 4b 4b  4b 4b 4b 4b 4b 4b 4b 4b  |KKKKKKKKKKKKKKKK|
0x20  53 53 53 53 53 53 53 53  53 53 53 53 53 53 53 53  |SSSSSSSSSSSSSSSS|
0x30  43 43 43 43 43 43 43 43  43 43 43 43 43 43 43 43  |CCCCCCCCCCCCCCCC|
0x40  4d 4d 4d 4d 4f 4f 4f 4f  4f 4f 4f 4f 4f 4f 4f 4e  |MMMMOOOOOOOOOOON|
0x50  4e 4e 4e 4e 4e 4e 4e 4e                           |NNNNNNNN|
0x58
```

Figure 5.5: Input data for the SHA-256 function of the ATSHA204A MAC command

Both have an identifier — the PSK identity hint for the server and PSK identity for the client — that they also need to transmit to their respective peer. This is no problem, however: The original PSK identity or PSK identity hint is simply concatenated with the binary 16 bytes nonce. The result is then Base64 encoded (Josefsson 2003). After Base64-Decoding each party simply truncates the last 16 bytes away. We chose this format for efficiency, but any other format that does not violate the TLS-PSK specification could be chosen to transport the nonces r_s and r_c to its corresponding peer.

Fig. 5.5 shows the input to the SHA-256(National Institute of Standards and Technology 2002) algorithm as it is used by the ATSHA204A (Atmel Inc. 2015b) when it executes the `MAC` command. In the image, every letter resembles exactly one byte of input data, and it is fed to the ATSHA204A in exactly the depicted order. The components which influence the outcome of the computation are:

- `K`: 32 bytes secret authentication key. This is one part of the pPSK, which, at production time is stored within the module and is ensured to be kept safe by the security features of the module itself. This pPSK needs to be available on the server side as well to be able to reproduce the key derivation.
- `S` and `C`: A total of 32 bytes public, random nonces which together form r. 16 bytes of this nonce are created by the server while the other 16 bytes are created by the client. The nonces are exchanged during the TLS handshake inside the PSK identity hint and PSK identity messages, as explained in Sect. 5.3. Note that in our construction we first feed the server nonce into the SHA-256 function and feed the client nonce into the module afterward.
- `M`: 4 constant bytes which encode which command was used, which mode the module is used in and what key slot is used. The value of these bytes is public. It for examples encodes the MAC mode itself, the key slot which is used and whether or not the following field, `O`, is fed from internal memory or is regarded as zero.
- `O`: 11 bytes of one-time programmable (OTP) memory of the ATSHA204A. These bytes can also be randomly chosen at production time and can then be considered an addition of the pPSK or the module can be configured to set these bytes to constant zeros. If they are used to augment the pPSK, they need to be stored on the server side to be able to reproduce the key agreement. Because they can be used to augment the key, we regard this value to be private.

- `N`: 9 bytes unique serial number of the ATSHA204A. This unique serial number is given out by the vendor of the module and identifies the module uniquely. This serial number is regarded as public for our security considerations. During production, this number needs to be read out of the specific module and stored on the server side as well.
- Workload factor ω. This is a public value which determines how often the hash function is called to derive the final PSK. It can be used to slow down the rate artificially at which a TLS client in possession of a valid HSM can generate PSKs.

The key derivation, in detail, works like this: For the first iteration, the module challenge ρ is equal to the concatenated server and client nonces, $S \,||\, C$. For each subsequent iteration, the challenge is the output of the previous round:

$$\rho_0 = S \,||\, C = r$$
$$\rho_n = \text{SHA-256}(K \,||\, \rho_{n-1} \,||\, M \,||\, O \,||\, N)$$
$$\text{PSK} = \rho_\omega$$

```python
def derive_psk(self, server_nonce, client_nonce):
    assert(isinstance(server_nonce, bytes) and (len(server_nonce) == 16))
    assert(isinstance(client_nonce, bytes) and (len(client_nonce) == 16))
    assert(isinstance(self.omega, int) and (self.omega >= 1))
    challenge = server_nonce + client_nonce
    for i in range(self.omega):
        challenge = self.mac(ppsk = self.ppsk, mode = self.mac_mode,
            otp = self.otp, serialno = self.serialno,
            challenge = challenge)
    return challenge
```

Listing 5.1: Python code to derive a PSK

The code in List. 5.1 shows the programmatic interpretation of this. For the server to be able to calculate the same PSK as the HSM itself, it needs to know the tuple (K, M, O, N, ω). That means, for example, that it needs to be predefined which specific key slot is used by the hardware, because this influences the outcome of the `MAC` command via the M parameter. Together with this stored information and the session nonces (S, C) it is possible to derive the 32 bytes PSK.

5.4 Implementation with Asymmetric Cryptography HSM

A hardware security module which can conduct asymmetric cryptography can be even more useful than a module which only can use symmetric cryptography. The necessary cryptographic implementations that the relevant TLS cipher suites which are used in the context of LWM2M are all based on the elliptic curve Diffie-Hellman key agreement protocol. To authenticate the keys which were agreed on, they are signed with an elliptic curve key pair. These are two distinct operations. Regarding the TLS cipher suite, this is referred to as the ECDHE-ECDSA key agreement and the authentication is specified in RFC4492 (Blake-Wilson et al. 2006). Concretely, this key agreement works as follows:

- It is assumed that each party is in possession of a static ECDSA key pair (d_1, Q_1) and (d_2, Q_2).
- Upon initialization of communication, each party generates a second, ephemeral, key pair (t_1, R_1) and (t_2, R_2).
- Each party signs their ephemeral public key using their static ECDSA private key. That is, R_1 is signed by d_1 and R_2 is signed by d_2.
- The signed ephemeral public keys are exchanged.
- Each party verifies the signature of the received ephemeral public key. That is, R_1 is verified by party 2 using Q_1 and R_2 is verified by party 1 using Q_2.
- After successful signature verification, each party performs the ECDH computation using their ephemeral private key and the ephemeral public key of the respective peer. The shared point $K = t_1 R_2 = t_2 R_1$.
- Of this shared point K, the X coordinate in affine representation is taken as the premaster secret for all following key derivations.

Concretely, the ECC operations ECDH and ECDSA are performed over the curve NIST P-256 (National Institute of Standards and Technology 1999). Now consider a hardware security module such as the one shown in Fig. 5.6. The concrete instance we show there is a module which uses the RSA cryptosystem, not ECC. This is because we want to illustrate that our approach works even with asymmetric HSMs which have nothing in common with the used cryptography within the TLS handshake.

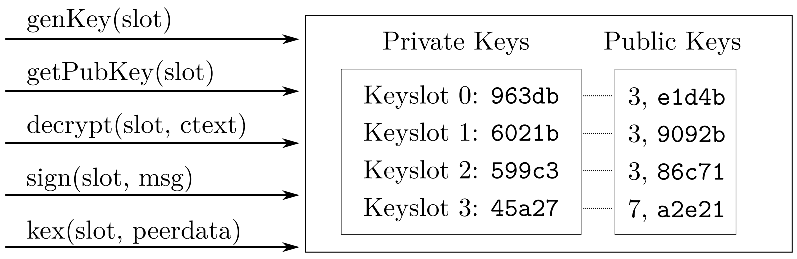


Figure 5.6: Basic layout of an asymmetric HSM based on RSA

The operations this module supports are as follows:

- `genKey`: Generate a new public/private key pair within the HSM on the given key slot. Note that the corresponding private key *never* leaves the HSM. In contrast to the symmetric modules, the key is therefore not known even to the person who legitimately uses the HSM.
- `getPubKey`: Extract the public key for a given key slot.
- `decrypt`: Decrypt some cipher text using the internal private key in a given key slot.
- `sign`: Sign a message using the internal private key in a given key slot.
- `kex`: Perform a key exchange. For this, use the data supplied by the peer together with the private key in the given key slot.

With that formal model of an asymmetric HSM, we now differentiate three different types of concrete HSM:

1. The module supports ECDH and ECDSA on NIST P-256. Such a module is the Atmel ATECC508A (Atmel Inc. 2015a).
2. The module supports only ECDSA on NIST P-256. Such a module is the Atmel ATECC108A (Atmel Inc. 2016).
3. The module supports neither ECDH nor ECDSA on NIST-P256 but offers a proprietary form of asymmetric key agreement and a proprietary form of asymmetric signature. This could be an HSM which operates on different elliptic curves or an HSM which uses an entirely different cryptosystem, like our exemplary RSA-HSM in Fig. 5.6.

Based on this, we can use three different implementations with varying degrees of security.

Case 1 is the easiest of the three. A module which supports the cryptographic primitives just as the TLS handshakes can be integrated effortlessly and in a completely transparent manner. The only thing that needs to be done is implementing the glue code between the HSM and the TLS library as a hardware engine driver. This means that the software calls the ECDH and ECDSA functions of the driver, which in turn relays the commands to the HSM and pass the responses along. At production time, the only thing that needs to be done is to have the module create a static key pair, read the corresponding public key out and store it in the backend. During session establishment, the module creates a new ephemeral key on-the-fly in a different key slot and performs ECDH with it. It also reads out the ephemeral public key and signs it using the ECDSA signing function with the static private key.

Case 2 is only slightly more complicated. Here, the generation of the ephemeral key needs to be done by a software implementation, but the signing of the ephemeral public key is relayed to the HSM. Even though ECDH is performed in software, the important part is the static long-lived key inside the HSM. At the time of production, the same procedure as in case 1 has to be performed.

It becomes a bit more complicated when looking at case 3. However, when looking how we solved the proprietary interface problem for the symmetric case in Sect. 5.3, this might seem familiar. We employ an asymmetric HSM, but we use a symmetric PSK handshake. Then, in the PSK identity hint message of the server to the client, we embed the server's public key. This is an opaque binary blob which might be in a proprietary storage format recognized by the HSM. The HSM then performs the `kex()` function with this provided peer data and an ephemeral key which was generated. The result of this key agreement is used as a PSK. Then it signs this ephemeral public key with a static public key and transmits that signed ephemeral key back to the server inside the PSK identity message. This completely emulates the way the asymmetric TLS handshake is usually done and uses the PSK cipher suite only as a technical means to set the key without having to modify the TLS library. Of the three option, this is the one requiring the most work. It, however, offers — in contrast to a PSK cipher suite used with a symmetric HSM — two advantages: First, this type of key agreement gives the session forward secrecy. Secondly, the backend does not need to store private keys, but only public values of the end nodes. This is an advantage because it makes hardening of backend systems much easier when there is less data that can be potentially lost during an attack.

5.5 Security Evaluation

We now differentiate between the two general types of modules — ones with symmetric and others with asymmetric cryptographic capabilities — and separately evaluate their security.

5.5.1 Security with Asymmetric Cryptography HSMs

To determine the security implications of our asymmetric approaches, we first need to clarify which aspects of our proposed scheme are important for assessing the security of the overall construction. When using an asymmetric HSM, as explained in Sect. 5.4, no protocol messages change if the implementation can manage to offload either the ECDSA or both the ECDSA and ECDH computation on the HSM. In fact, for these two cases the TLS protocol stays the same no matter whether an HSM is used on the client side — the only thing that changes is whether or not the corresponding client private key is safely stored or not. Therefore, these cases do not need to be evaluated separately. If it can be assumed that the ECDH-ECDSA part of TLS is secure algorithmically, and the HSM does not introduce a source of disturbance (such as side channel leakage or a weak random number generator), the protocol is not weakened by the introduction of our hardware security module. It can, in fact, be viewed as if it solely were a cryptographic accelerator chip.

For our constructed imaginary HSM with proprietary handshakes, our reasoning is similar. We make the following assumptions:

- The proprietary key agreement function of the HSM is producing shared secrets with at least the level of entropy that is desirable for the security of the connection.
- The key agreement function does not exhibit key leakage, undefined behavior or side channels, even when confronted with intentionally malicious peer input.
- The signing function provides a level of security that is appropriate for the level of security the connection requires and that signing function is nonsusceptible to low-entropy attacks such as nonce reuse issues in the case of ECDSA.
- The underlying PSK cipher suite is secure to the amount that is required by the connection itself.

Then it can be reasoned that the procedure we suggest is simply an analogous emulation of the TLS standard as it exists today and as it has been extensively peer-reviewed. Therefore, the construction we propose should be secure to the same amount.

5.5.2 Security with Symmetric Cryptography HSMs

For the symmetric HSM, however, our analysis is more complicated. As we explained in Sect. 5.3, a TLS library usually calls a PSK lookup callback function. In this function, we perform out our PSK derivation using the pre-PSK (pPSK) which has been stored in the HSM. It is important to stress that for the security of the TLS-PSK protocol, such a derivation of a key is in no way required. In fact, the contrary is the case: any implementation which returns a constant, but secret, value fully benefits from the security guarantees of TLS-PSK. Therefore, within the security review, we merely need to ensure that our PSK derivation is, from a cryptographic point of view, better than the identity function to achieve at least the same level of security that TLS-PSK would provide without an HSM.

To explain it formally, instead of using a PSK directly in the classical sense, we securely store a pre-PSK in the HSM. From that pPSK we derive the PSK using the module-proprietary derivation function κ and a nonce r:

$$\text{PSK} = \kappa(\text{pPSK},\ r)$$

Formally, for the case without an HSM, we can also define a pPSK which however is identical to the PSK. The function κ refers to the identity function. To stress it again, our proposal does not replace the internal TLS-PSK master secret derivation at all. It is *additionally* used at the beginning of the key agreement of a connection.

5.5.3 Attack Scenarios and Assumptions

We differentiate two different attack scenarios:

1. An attacker who can eavesdrop and modify the connection between client and server gains knowledge of the pPSK, i.e., the secret parts of the key tuple (K, O). In this attack scenario, the master key is compromised.
2. An attacker who actively modifies a connection can force a specific PSK derivation by manipulation of the transmitted server nonce S. In contrast to the previous case, this scenario compromises a session key.

The implications of these attacks is as follows:

1. Attack 1: An attacker who learns (K, O) can decrypt all connections which were recorded and have taken place in the past. This is because the specific PSK cipher suite we use does not offer perfect forward secrecy. Additionally, all connections which take place in the future can be eavesdropped on by the attacker, and the adversary could additionally selectively modify this connection when using a man-in-the-middle (MitM) attack.
2. Attack 2: An attacker who can manipulate the TLS handshake by manipulation of the value S and who is, therefore, able to force a specific connection to derive a PSK known to the attacker needs first to actively take part in the handshake for this manipulation to occur. Then the attacker can decrypt and modify the traffic of the compromised session traffic at will.

For our analysis, we make the following assumptions:

1. An attacker who can passively eavesdrop connections is not able to learn the PSK from the exchanged messages. This implies that we inherently trust the cryptographic primitives which are given by TLS-PSK.
2. The random number generator (RNG) of the client is assumed to be ideal; it produces high-quality random numbers with high entropy and these numbers follow a uniform statistical distribution. For our implementation, this implies that we trust the assertion of the HSM vendor in the quality of their RNG.
3. The hash function SHA-256 is considered to be an ideal cryptographic hash function. It exhibits an avalanche property, and the distribution of hash values is statistically uniform. We, therefore, trust the integrity of SHA-256 and assume there are no algorithmic attacks on the compression function itself.

These assumptions are sensible for the following reasons:

Assumption 1 is the prerequisite to calling TLS-PSK secure in the first place: As described in Sect. 5.5, if no hardware security is used, the system is identical to one where the function κ to derive PSK from pre-PSK is the identity function. If it were possible in the construction of TLS-PSK to gain knowledge of the PSK by observation of the control flow, an attacker could simply do this and break all security goals which TLS tries so offer.

Assumption 2 is plausible because the client side ATSHA204A offers a true random number generator (TRNG). If this generator is used and the vendor's claim is true, there would be a high-quality entropy source present in the client that can and should be used to generate the random nonces.

Last but not least, assumption 3 treats SHA-256 as an ideal compression function with 256 bits hash length. Gilbert and Handschuh (2004) take a detailed, in-depth look at the SHA-2 family of hash functions and conclude that all in the literature described, attacks against hash functions are not practically applicable to SHA-256. Both Aoki et al. (2009) as well as Khovratovich, Rechberger, and Savelieva (2012) describe attacks against SHA-256; in both cases, however, the authors deal with round-reduced versions of the hash function. Up to date, there is no known cryptanalysis against the SHA-256 with the full number of rounds which has a complexity that is smaller than that of a brute force attack.

5.5.4 Theoretic Analysis

In the following we analyze our PSK derivation with the weakest possible load factor, $\omega = 1$. Then the type of message composition the hardware uses can be viewed as an *envelope construction* as described by Tsudik (1992). In it, a MAC m over a given message M is calculated by concatenating M with keys on both the left and right side before the concatenated message is run through a cryptographic hash function (Menezes, Oorschot, and Vanstone 1996):

$$m = \mathrm{H}(K_1 \,||\, M \,||\, K_2)$$

Applying that to the construction which is used by the ATSHA204A we, therefore, arrive at:

$$K_1 = K, \quad M = S \,||\, C, \quad K_2 = M \,||\, O \,||\, N$$

Envelope constructions have been thoroughly analyzed in literature because of their popularity in protocols like IPsec (Metzger and Simpson 1995). Preneel and Oorschot (1995, 1996) note in their analysis that envelope constructions are fundamentally vulnerable to a divide and conquer attack if it is possible to create message forgeries which have appended suffices. For the ATSHA204A, however, it is an unalterable constraint of the hardware that the calculated hashes are always ever performed over the static 88 bytes of data shown in Sect. 5.5; therefore these attacks are not applicable to our scenario.

In the construction described by us, the value that is the result of the HSM's MAC function is used as a PSK for TLS. It directly follows from assumption 1 that the attack 1 does not apply here: if an attacker is unable to determine the PSK from the exchanged messages and the PSK is the output of the MAC-function, then an attacker cannot recover m from intercepted traffic. If the attacker does not have m, she cannot draw inferences from it about the tuple (K, O) which led to the calculation of that m.

However, an attacker is free to modify arbitrarily S during the handshake phase; since it is transmitted in plain text as part of the PSK identity hint message, it is transmitted in plain text without any authentication. The attack scenario 2 still is not applicable here: The direct consequence of assumption 2 in conjunction with assumption 3 leads to the fact that the resulting PSK m changes in an avalanche fashion because of the used client nonce. This means an attacker cannot choose an a priori value S because m significantly depends on the value C which is only generated after S has already been received by the client.

5.5.5 Practical Analysis

To quantify the level of security, we now consider the worst case estimate. The assumption we make for an extremely strong attacker model. Namely, these assumptions are:

- An attacker has direct access to the result m of the MAC function (i.e., the PSK). The attacker knows the serial number N and the configuration options M of the ATSHA204A.
- The attacker can manipulate not only the server side nonce S but also the client-side nonce C at will.
- A divide and conquer attack on the cryptographic construction is possible (even though, as shown in Sect. 5.5.4, this is not the case in practice).
- The complexity of the divide and conquer attack would solely depend on the 11 bytes OTP suffix O; since the attack complexity of the divide and conquer attack is $2^{\frac{k}{2}}$ messages, this would be 2^{44} messages in our concrete case.
- There is no communication overhead towards the ATSHA204A; every execution of the MAC operation takes its typical execution duration of 12 ms.

If an attacker now tries to extract the module-internal key K, she would need to have access to 2^{44} messages with corresponding valid MAC value to have reasonable chances of success regarding a key recovery attack. Calculation of this number of messages alone would, however, take:

$$\omega \cdot 2^{44} \cdot 12\text{ms} \approx \omega \cdot 2.11 \cdot 10^{11} \text{ sec} \approx \omega 6690 \text{ yr}$$

For this type of attack, the inherent hardware limitation of the module itself is the bottleneck because the module is not capable of calculating the MAC command any faster. In practice, the attack is, due to real-world circumstances, much more difficult than our attack model and therefore not be applicable. However, consider a different type of attack, where an attacker does not need a hardware security module, but can rely on ASIC hardware. Since SHA-256 is used in BitCoin mining, much effort has been put into optimization of hash cracking ASICs. This means that we have very good real-world estimates of efficient hash-cracking systems. One such system is the BitMain AntMiner S7 (Bitmain Technologies Ltd. 2016). It delivers 4.73 TH/s at 1293 Watts, which equals an efficiency of $3.66\frac{\mathrm{GH}}{\mathrm{s\cdot W}}$. Consider therefore an alternative attack scenario:

- The attacker can make use of the birthday phenomenon, reducing the complexity of finding a collision in SHA-256 to 2^{128}.
- Power is available to the attacker for \$0.10 per kWh of energy.
- The attacker spends \$1000 per day on electricity; the hardware is free.
- The system's efficiency starts with $3.75\frac{\mathrm{GH}}{\mathrm{s\cdot W}}$, but automatically increases exponentially according to Moore's Law (i.e., it doubles every 18 months).
- The workload factor ω was chosen to give the weakest possible construction, i.e., $\omega = 1$.

This leads to the following calculation. The energy that is available to the attacker every day is:

$$E = \frac{\$1000}{\frac{\$0.10}{\mathrm{kWh}}} = 10000\ \mathrm{kWh} = 3.6 \cdot 10^{10}\ \mathrm{J}$$

Therefore, the system's initial throughput η_0 is:

$$\eta_0 = 3.75\frac{\mathrm{GH}}{\mathrm{s\cdot W}} \cdot \frac{E}{\mathrm{day}} = 1.35 \cdot 10^{20}\frac{\mathrm{H}}{\mathrm{day}}$$

Considering that η has exponential growth according to Moore's law and doubles every 18 months (i.e., 547 days), we get the throughput function in dependence of the elapsed time t:

$$\eta_{(t)} = \eta_0 \cdot 2^{\frac{t}{547\ \mathrm{day}}}$$

With those numbers we can now calculate the time t it would take for such an attack:

$$\begin{aligned} t \cdot \eta_{(t)} &= 2^{128}\ \mathrm{H} \\ t \cdot \eta_0 \cdot 2^{\frac{t}{547\ \mathrm{day}}} &= 2^{128}\ \mathrm{H} \\ t \cdot 1.35 \cdot 10^{20}\frac{\mathrm{H}}{\mathrm{day}} \cdot 2^{\frac{t}{547\ \mathrm{day}}} &= 2^{128}\ \mathrm{H} \end{aligned}$$

We define

$$t = x \cdot \mathrm{day}$$

to simplify the calculation:

$$x \cdot 1.35 \cdot 10^{20}\mathrm{H} \cdot 2^{\frac{x}{547}} = 2^{128}\ \mathrm{H}$$

$$x \cdot 2^{\frac{x}{547}} = 2.52 \cdot 10^{18}$$

Solving for x:

$$x = \frac{547}{\ln 2} \cdot W\left(\frac{2.52 \cdot 10^{18} \cdot \ln 2}{547}\right) \approx 789 \cdot W(3.19 \cdot 10^{15})$$

where W is the Lambert W function. This gives a solution of

$$x \approx 25426$$
$$t = x \cdot \mathrm{day} = 25426 \cdot \mathrm{day}$$

i.e., around 70 years and a total cost of about \$25 million. It can, therefore, be concluded that, notwithstanding our strong attacker model, as long as our basic assumptions hold true, this is a computationally infeasible type of attack.

5.6 Conclusion

With our work, we show that the integration of hardware security modules in preexisting security infrastructures is not only feasible but even practical. Our approach for incorporation of symmetric hardware security modules does not need any modification in the internal protocol structure of DTLS. Likewise, for our asymmetric hardware security module, we show that the implementation and resulting security considerations are even easier than in the symmetric case. This allows for flexible integration into the TLS infrastructure. For the symmetric case, all code that needs to be implemented can be written as part of the application without modification of the TLS library. For the asymmetric case, an engine API can be used; since this is commonly available to allow for cryptographic acceleration hardware, it is also a practical alternative — with the added benefit of creating a completely transparent integration of the HSM into the TLS handshake. Since the price of such modules is relatively low, this, therefore, enables applications in the low-cost segment to benefit from hardware that is specifically designed to withstand physical attacks. Our work, therefore, is a concrete contribution towards the hardening of these types of end nodes against physical attacks.

Chapter 6

DRAM Scrambling

As hard disk encryption, RAM disks, persistent data avoidance technology and memory-only malware become more widespread, memory analysis becomes more important. Cold-boot attacks are a software-independent method for such memory acquisition. However, on newer Intel computer systems the RAM contents are scrambled to minimize undesirable parasitic effects of semiconductors. While not intended to be of cryptographic protection, this RAM scrambling was largely considered prohibitive to performing raw memory acquisition by direct readout of the memory chips.

In this chapter, we present a descrambling attack that requires at most 128 bytes of known plaintext within the image to perform the full recovery. This attack is further refined by us using the mathematical relationships within the keystream to at most 50 bytes of known plaintext for a dual memory channel system. Therefore, we enable cold-boot attacks on systems employing Intel's memory scrambling technology.

The research in this chapter has been presented at the European Digital Forensic Research Workshop (DFRWS EU 2016) as a full paper and was published in Digital Investigation (Bauer, Gruhn, and Freiling 2016).

6.1 Introduction

For several reasons, the contents of volatile memory (RAM) are a valuable piece of digital evidence during a forensic investigation. Firstly, the keys for full disk encryption are usually stored in RAM. Extracting such keys from a memory snapshot, therefore, allows accessing contents of encrypted storage that would be inaccessible otherwise. Secondly, a plethora of other information about the current system state can be recovered from RAM, including ephemeral cryptographic communication keys, the list of running processes and the details of active network connections. Last but not least, new forms of memory-only malware can only be analyzed while they are active in memory. So overall, with the increasing use of encryption technology, cloud storage, and memory-only malware, forensic memory image acquisition has become increasingly important.

There has been a lot of debate on how to properly perform imaging of volatile memory since there exist a variety of options (Vömel and Freiling 2011). One commonly chosen option is runtime acquisition via software using specific memory acquisition tools like WinPmem. While this method appears convenient in many cases, memory imaging may be manipulated by malware that hides in inaccessible memory regions (Stüttgen and Cohen 2013), thus creating a memory image that is not forensically sound. Another problem of software acquisition methods is that they operate concurrent to regular system activity and therefore produce inconsistencies that do not occur in “perfect” memory snapshots (Vömel and Freiling 2012). A generic option that avoids these problems is to perform a so-called *cold boot attack*. These attacks exploit the *remanence effect* of modern RAM technology.

Modern RAM technology is commonly based on dynamic random access memory (DRAM), a type of RAM in which the cells which store data are constituted of an array of capacitors. Each capacitor is either charged or discharged, depending on whether the cell bit is set or cleared. Since capacitors have a leakage current, their data content slowly dissipates over time. Therefore, to effectively use DRAM, each and every cell has to be periodically refreshed. This is achieved by reading the contents and writing it back to the RAM chip. The time that DRAM keeps its contents without leakage affecting the content is referred to as *retention time*. The fact that the retention time is nonzero and that memory keeps its contents for a while nevertheless even when it is not actively refreshed is referred to as the *remanence effect*. It is well-known from electrical engineering that the leakage current of capacitors grows exponentially with their temperature (Wyns and Anderson 1989). Therefore, the retention time of RAM dramatically decreases with increased chip temperature.

Cold boot attacks exploit the remanence effect and can be executed in two ways: One way (Halderman et al. 2009) is to reset the target computer by using the reset button and boot from an alternative medium such as USB using a special imaging USB stick that contains only a minimal operating system together with imaging software such as memimage (ibid.). Ideally, the original contents of RAM are maintained and can be recovered, apart from those parts that have been overwritten by the acquisition software. Unfortunately, such an attack is easily thwarted by using trivial protection mechanisms like BIOS passwords.

The other way to perform cold boot attacks is to “transplant” the RAM module physically at runtime from the device under investigation into an acquisition computer and perform image extraction on that second computer. When the semiconductors are properly cooled before transplantation, they retain most of their content. It is, therefore, beneficial to freeze the DRAM modules using a cooling spray to increase the remanence effect. To the best of our knowledge, this second form of cold boot attack is paradigmatic for the class of memory acquisition procedures since it combines genericity, availability and offers the highest level of integrity and atomicity for the acquired memory images (Vömel and Freiling 2011, 2012).

While the remanence effect itself is already well-studied (Hamamoto, Sugiura, and Sawada 1998; J. Liu et al. 2013), Halderman et al. (2009) were the first to exploit it to attack full disc encryption systems on desktop PCs. However, it has been shown that cold boot attacks also affect a multitude of other devices such as smartphones (Müller and Spreitzenbarth 2013). The dominating RAM technology at that time was called DDR2. Recently, however, Gruhn and Müller (2013) reported that the results of Halderman et al. (2009) could not be repeated with the more modern DDR3 RAM technology. Even worse, the memory images obtained from cold booting DDR3 devices appeared to be random for reasons inherent in that technology.

With the increasing speed of semiconductors, their undesirable parasitic effects also grow in magnitude. Current spikes and electromagnetic interference in the speed categories of DDR3 start to affect reliability and regulatory compliance. To counteract this, RAM manufacturers in general and Intel, in particular, perform memory scrambling in DDR3 memory controllers. As we explain in this chapter (and as observed by Gruhn and Müller (2013)), these scramblers severely limit the potential of forensic image acquisition. While Lindenlauf, Höfken, and Schuba (2015) performed measurements of the remanence effect magnitudes with DDR3 memory, they excluded a discussion of memory scrambling. So, to the best of our knowledge, there is no published work which investigates the possibilities of performing cold boot attacks on modern DDR3 systems in general, i.e., possibilities to "descramble" the scrambler effects.

Zandwijk (2015) recently performed some related work using an analytic approach to descrambling NAND flash chips. Their approach, however, is unfortunately not easily transferable to RAM acquisition since they require error-free source bitstreams which cannot be guaranteed for a cold boot process. Faintly related is also work by Rahmati et al. (2012) which use the remanence effect constructively to increase the security of embedded systems; they use the remanence effect as a means of timekeeping. For completeness, we also mention work by Kim et al. (2014) because they also exploit physical properties of RAM semiconductors to provoke bit flips within the RAM modules.

To summarize, the ramification of memory scrambling is currently not well understood, and there is no general method of performing cold boot attacks on scrambled DDR3 memory. In this chapter, we present an in-depth analysis of DDR3 scrambling as performed by the Intel memory controller and we show how to use this knowledge to develop a practical method of descrambling such DDR3 memory in real-world scenarios.

6.1.1 Contributions

The contributions which we make in this chapter are as follows:

- We present a template attack on scrambled DDR3 memory systems which requires 64 bytes of known plaintext per memory channel (i.e., at most 128 bytes for a dual-channel system) within the memory snapshot to yield complete descrambling of the image.
- We refine this template attack by exploiting the mathematical relationships present within the keystream to reduce the number of known plaintext bytes to only 25 (i.e., 50 bytes for a dual-channel system).
- We present methods which can be used to deinterleave dual-channel memory and give an algorithm which can construct an interleaved keystream of arbitrary length from the subkeys for each channel.

We make the source code and documentation of our research freely available to the community at `https://www1.informatik.uni-erlangen.de/filepool/mem/ddr3descramble.tar.gz`.

6.1.2 Outline

We first give some necessary background information in Sect. 6.2, then precisely formulate the problem of descrambling DDR3 memory in Sect. 6.3. Then we show how the problem can be solved in Sect. 6.4 and present some experimental results confirming our findings in Sect. 6.5. Finally, our conclusion is given in Sect. 6.6.

6.2 Background

We now give the background required to understand the technical aspects of the topic by first explaining why scrambling is used and how it compares and contrasts to cryptographic stream cipher encryption in Sect. 6.2.1. Then we proceed to show the mathematical background of linear-feedback shift registers and their construction in hardware in Sect. 6.2.2. The topic of DRAM and different peculiarities when operating DRAM outside its specified operating range are covered in Sect. 6.2.3. Finally, in Sect. 6.2.4 we build a bridge between these three topics: We give concrete implementation details of Intel's memory scrambling scheme and how it works internally.

6.2.1 Scrambling

Storage of bitstreams which are strongly biased towards zero or one can lead to a multitude of practical problems: Modification of data within such a biased bitstream can lead to comparatively high peak currents when bits are toggled. These current spikes cause problems in electronic systems such as stronger electromagnetic emission and decreased reliability. In contrast, when streams without DC-bias are used, the current when working with those storage semiconductors is, on average, half of the expected maximum.

Scrambling, also referred to as *whitening*, can be applied to biased bitstreams to remove these undesirable side effects. In the simplest case, such a scrambler is a pseudo-random binary sequence (PRBS) that is added onto the input bitstream using the exclusive or (XOR) addition. We call these devices *additive* or *synchronous scramblers*. For the receiving side, the inverse operation, *descrambling*, must be applied to get the original content. A pleasant side effect of the XOR operation is that scrambling and descrambling is symmetric; it effectively is the same operation.

From a signal processing perspective, a whitened signal has a smooth power distribution over the allocated bandwidth. This means there are no special frequencies present which have spikes in the power distribution and instead every part of the spectrum is utilized equally well. Hardware constraints are sometimes also the reason to employ scrambling: The Ethernet physical layer standard, for example, enforces strong galvanic isolation of bus participants (IEEE Computer Society 2012). This is usually done using magnetics, i.e., magnetic transformers. Transformers can only transport AC from one primary to the secondary coil, however. Usage of transformers, in this case, is only possible because the standard guarantees a bias-free signal — in the case of Ethernet, 8b/10b encoding is used (Widmer and Franaszek 1983), but the principle remains the same.

Another reason to ensure signals are bias free is that this can enable clock recovery. If the receiver can safely assume that a certain amount of state transitions occurs for a given amount of transported data bits, then it is possible to omit the explicit clock and let the receiver regenerate the clock from the received data stream. This complicates the receiver but makes it possible that phase noise (jitter) is much less of a problem than with synchronously clocked systems. All high-speed buses (USB, SATA, DRAM) today use some form of clock recovery mechanism, albeit with much more sophisticated techniques (Altera Inc. 2015; Hewlett-Packard Inc. et al. 2013; Micron Technology Inc. 2008).

6.2.2 Linear-feedback Shift Registers

The computing term *register* refers to a set of D-type (data type) flip-flop memory cells which are clocked synchronously, i.e., latch the input data all at the same point in time. A *shift register* is a special type of register in which the input of each flip-flop is fed from the output of the next memory cell. Such a shift register with n stages has exactly one *feed* data input, n state bits $Q_0..Q_{n-1}$ and out output bit Q_0. Any bit that is fed into to the input cycles through all flip-flops with each clock cycle until it comes out of the last register after n cycles. Shift registers, therefore, have a first-in-first-out (FIFO) semantic.

If such a register is now *fed back* its output value, that is, if the output Q_0 is connected to the feed input of the flip-flop chain, the result is a *feedback shift register*. In such a register, the content just revolves around, and the periodicity is at most n. Instead of feeding back the output bit directly, one can feed back a linear combination of the internal state instead. This creates a *linear feedback shift register*. The name precisely describes its mode of operation: the feeding input is determined by a linear function of the internal state. In practice, the linear function which is most commonly used to combine the state bits is the exclusive or (XOR) function.

The maximum amount of different internal states such an LFSR cycles through when it is continuously clocked after initialization to a nonzero state is called the *period* of the LFSR. If an LFSR with a bit width of n bits has a period of $2^n - 1$ it is called a *maximum-length LFSR*. Every non-trivial initialization of such a maximum-length LFSR always causes the shift register to output the whole bitstream periodically; the initialization state only determines the phase within the bitstream at which the stream generation begins.

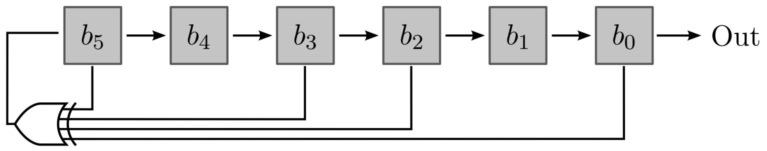


Figure 6.1: Fibonacci-LFSR with polynomial $x^6 + x^4 + x^3 + x + 1$

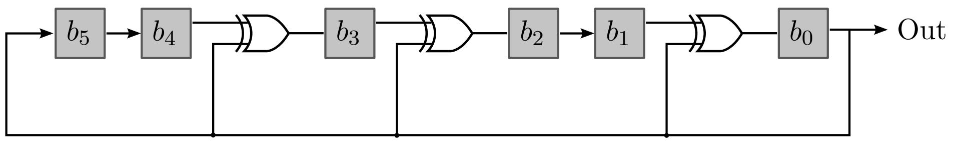


Figure 6.2: LFSR with polynomial $x^6 + x^4 + x^3 + x + 1$

Mathematically, this construction corresponds to polynomial division in finite fields. Let G be a polynomial in $\mathbb{F}_2$ which we refer to as the *generator*:

$$g(x) : c_0 + c_1 x + c_2 x^2 + c_3 x^3 + ... + c_n x^n \qquad c_i \in \mathbb{F}_2$$

Then select a second polynomial s which corresponds to the state of the LFSR. The output sequence of the LFSR is given by the coefficients of the quotient $b(x)$ of the two polynomials:

$$b(x) = \frac{s(x)}{g(x)}$$

Mathematically, LFSRs can be modeled as polynomials over $\mathbb{F}_2$ in which the state bits $b_0, \ldots, b_{n-1}$ represented the polynomial coefficients. Clocking corresponds to repeated multiplication of the whole polynomial with x and reduction modulo the *characteristic polynomial* P of the LFSR.

For implementation in hardware, there are generally two semantically equivalent representations of LFSRs. One is the Fibonacci variant, shown in Fig. 6.1 and the other the Galois variant, shown in Fig. 6.2. Note that both produce the same bitstream, but for the same internal state configuration, produce the sequence with some offset to each other. This is because the state bits $b_0..b_{n-1}$ mathematically only indirectly relate to the state bits of the numerator polynomial h. A detailed proof for this can be found with Golomb et al. (1981) or Goresky and Klapper (2002). The reason why this is mentioned here is the following: If for a given output sequence $a_0..a_{n-1}$ and known characteristic polynomial P the internal state needs to be determined, it is important to select the correct model depending on the used algorithm; otherwise, the used algorithm does not produce the desired results.

6.2.3 DRAM

As explained before, DRAM needs to be continuously refreshed to keep its content. For modern DRAM generations such as DDR3, this process is implemented by logic within the DRAM module itself (Micron Technology Inc. 2014). It is, however, *triggered* by the host. In Intel systems, this is the task of the memory controller hub (MCH). While the CPU and MCH were separate chips in early hardware generations, the MCH is completely contained within modern CPUs.

When the retention time is exceeded — for example, because the DRAM chip is unpowered — the stored charge of the capacitors slowly decays and the RAM takes its *ground state*. The ground state is not a trivial pattern of, for example, all zero bits, but depends on the physical construction of the chip itself. Namely, whether the statically connected sides of the cells are biased against GND or the positive supply voltage, V_{DD}. The memory pattern that a DRAM chip, therefore, shows when it far exceeded its retention time is heavily dependent on the physical semiconductor construction.

Another important aspect is the mapping between physical addresses and the physical storage location within memory modules. When more than one memory module is present in a computer, the RAM is usually operated in so-called *dual-channel* mode. In this mode, consecutive data is alternated in a certain pattern between modules. This is done for performance reasons, as it allows the use of two memory modules in parallel.

6.2.4 LFSR RAM Scrambling

As we have explained in Sect. 6.2.2, LFSRs are versatile building blocks that can be implemented efficiently in hardware. Since the output bitstream of a well-chosen LFSR has pseudo-random properties and is yet completely predictable when the internal state is known, they are ideal for to achieve a pseudo-random bitstream (PRBS) in which each bit appears with approximately equal probability. The resulting bitstream is said to be *DC-balanced* or without *DC bias.* Since the goal of scrambling is only to achieve bias removal, cryptographic requirements are not satisfied by a scrambling PRBS which has been generated by an LFSR.

Concretely, we take a look at the actual process implemented by the MCH construction of Intel. As the authors explain in the patent on the topic, their aim is to reduce both electromagnetic interference (EMI) and current spikes by scrambling (Falconer, Mozak, and Norman 2013; Mozak 2011).

To achieve this, they use a set of parallel LFSRs which generate a PRBS that is XORed with the data. Therefore, the data on the memory bus appears to be random and ideally exhibits no bit-bias. To be able to generate the PRBS for an arbitrary memory location, a short secret value, the so-called *seed*, is chosen on first power-up. Upon a read or write request to the RAM, this seed is mixed with the memory address to which the request was made. Fig. 6.3 illustrates this process schematically. This combined value then serves

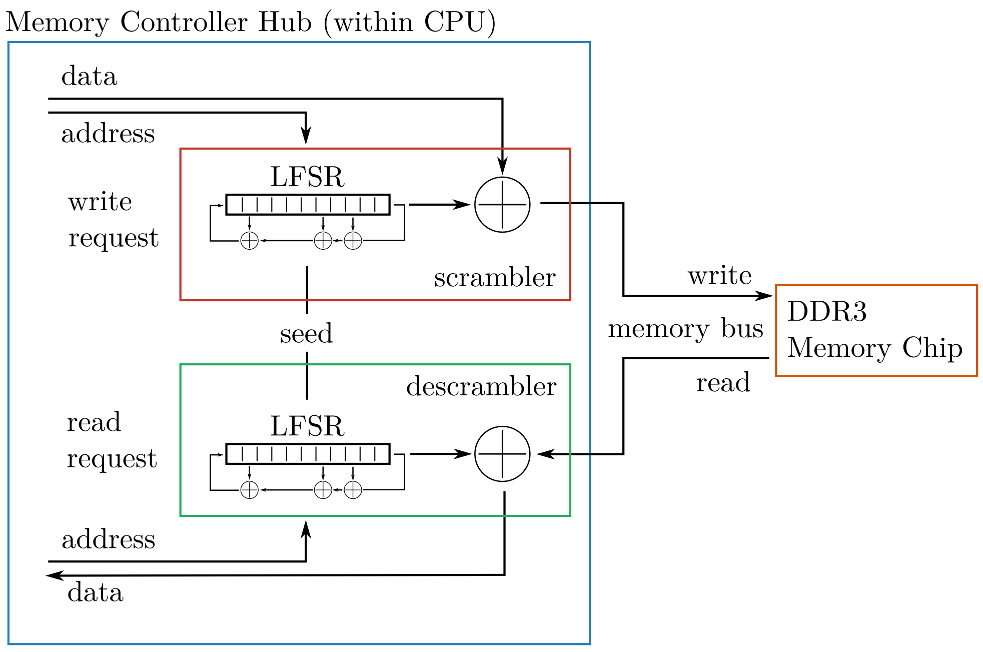


Figure 6.3: Schematic display of a Intel DDR3 scrambler

as the parameterization of the LFSR that generates the PRBS for the requested memory access. In this way, it is possible for the MCH to create the required PRBS for every random memory request. While the Intel patent (Falconer, Mozak, and Norman 2013) gives an example seed calculation in which only the lesser significant bits of the address are involved to parameterize the localized PRBS, it is unclear if this is indeed the method that is used in practice.

Note that when memory is operated in dual-channel mode, as explained in Sect. 6.2.3, each of the two channels has its scrambler and both scramblers also usually have distinct seed values. They are two completely independent hardware units.

During the boot process, the MCH is programmed by code that is part of the computer's firmware (i.e., the BIOS or UEFI). It is during this initialization that the MCH parameters — including the scrambler configuration and scrambler seed — are programmed as well. Therefore, a reseeding of the MCH scrambler can only occur when the computer is performing a cold start (i.e., transitions from unpowered to powered state). In our trials, the seed was never reset when the computer was merely rebooted using the reset button.

6.3 Problem Description

As mentioned in the introduction, one approach to performing image acquisition is to reset the target computer by using the reset button (thereby *not* reinitializing any memory scramblers) and boot from an alternative medium such as USB. Tools like the memimage imaging software (Halderman et al. 2009) have a sufficiently small memory footprint as to leave most of the memory contents intact. However, such attacks are easily thwarted using BIOS passwords for example. The other approach is to transplant the RAM module physically from the suspect's computer into an acquisition computer and perform image extraction on that second computer, as shown in Fig. 6.4. It is in this case that scrambling comes into full effect since the RAM chip contains only scrambled data and the acquisition computer cannot have the correct seed to replicate the original keystream.

As shown in Fig. 6.3, all data that is passed to or from the DRAM chips is first passed through the scrambler circuitry within the memory controller hub. It is transparently scrambled when writing to and transparently descrambled when reading from RAM. Therefore, at any point in time, the RAM module contains only a scrambled image M. All connected peripherals are not aware that scrambling is even happening, but only see the plain RAM image P. We refer to the scrambling data stream as K and the connection between the three is simply an XOR relationship, just as with a stream cipher: $M = P \oplus K$. When such an image is forensically acquired via means of cold booting, the captured image is referred to as I. While during normal operation of the computer the key K_0 was used by the scrambler, during the image acquisition phase this key might be K_1, where K_0 is not necessarily equal to K_1. Fig. 6.4 shows this formally: During the normal use, the plain image P is scrambled by K_0 and the memory M consists of the value $K_0 \oplus P$ which resides in RAM:

$$P \xrightarrow[\text{scramble}]{K_0} P \oplus K_0 = M \tag{6.1}$$

When the RAM module is transplanted to the analysis machine, generally the scrambling key is different between the two systems. We denote the new key by K_1. Subsequently, during the acquisition phase, the descrambler adds K_1 to the RAM image, yielding an image

$$I = P \oplus K_0 \oplus K_1 = M \oplus K_1 \xleftarrow[\text{descramble}]{K_1} M \tag{6.2}$$

Therefore, in a cold boot scenario the descrambler does not do what the name suggests (i.e., recover the original plaintext image), but instead actually adds another layer of scrambling on top of the already scrambled image. During normal operation of the computer, K_0 is equal to K_1 so that the keystreams cancel each other out and the descrambler recovers the plaintext image.

Intel's patent on their scrambler mechanics explains that a parallel LFSR is used to generate the scrambler bitstream K. Therefore, it seems that decrypting such a scrambled image would be a rather simple, straightforward task. Surprisingly enough, in practice, it turns out to be more complicated than initially anticipated. This has a number of reasons:

1. *Nonexistent public documentation*: All documentation that explains the registers which are used by the memory controller hub (MCH) — the component that contains

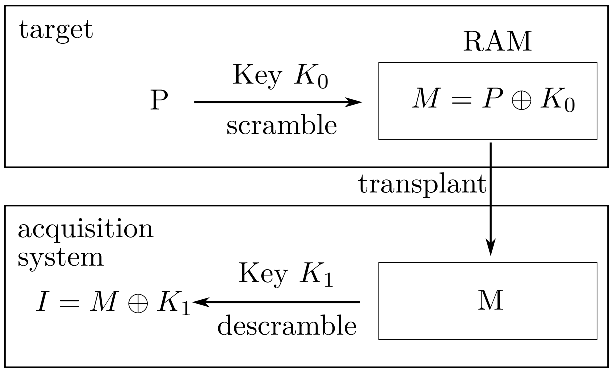


Figure 6.4: Scrambled storage of data and image acquisition

the scrambling unit — is non-public. Parts of the documentation which are publicly available, such as the patents Intel has filed on the issue (Falconer, Mozak, and Norman 2013; Mozak 2011), are worded as broadly as possible to include a plethora of different options. It is unclear which one is used in practice.

2. *Lossy image acquisition*: Forensic image acquisition when using cold boot techniques relies on the remanence effect of the semiconductors. This effect is neither guaranteed nor reliable. Bit flips frequently occur as the DRAM cells lose their content. When trying to reverse engineer the used scrambling mechanisms, this poses a problem since algorithms like Berlekamp-Massey (Massey 1969) for the synthesis of an LFSR from a given bitstream rely on perfect input data to produce correct results. When the algorithms are fed noisy input, they do not indicate failure, but instead, synthesize misleading output.

3. *Unknown ground state*: If the DRAM content of a chip which was inactive for a long time, i.e., the ground state of that chip G, were known, then it would be easy to determine the pure scrambler bitstream. We could perform a cold boot attack on a machine that had been turned off for a long time. While then assuming that $M = G$, we can determine $K = G \oplus I$, since for this machine the memory content C would be equal to G and it would run through the descrambler during forensic image acquisition. However, the ground state G is highly dependent on the actual constitution of the hardware itself and forms a nontrivial pattern. Therefore, it is difficult to gain access to the pure scrambling bitstream.

4. *Interleaved memory*: Lastly, most modern systems with more than one physical RAM chip are configured to use dual channel mode to improve system performance. This means that consecutive data is put on alternating RAM modules in an unknown pattern. Since each channel has its own, completely separate scrambler instance, it must be known which pieces of data have been scrambled by which unit to perform descrambling for such images.

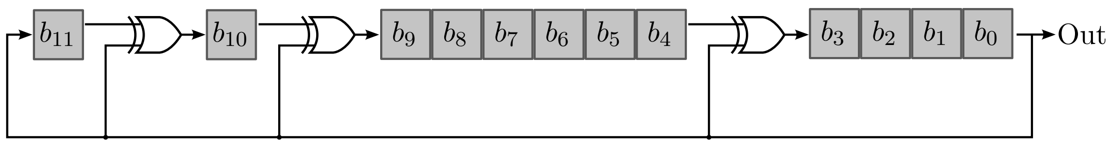


Figure 6.5: Galois-LFSR with polynomial $x^{12} + x^{11} + x^{10} + x^4 + 1$

6.4 Towards Descrambling

We now describe an approach to descramble the contents of a DDR3 memory image that was acquired using cold boot. The first steps which we describe are necessary to calculate certain parameters of the hardware that are necessary for descrambling to work. These steps can, however, also be performed *after* the memory image has been taken.

6.4.1 Practical LFSR Algorithms

During experimentation with the descrambler and to try to figure out how the internal MCH construction could look like, we used several algorithms which are useful to reproduce our findings. We briefly explain their function and how they are used.

```python
def clk_lfsr_galois(state, width, poly):
    assert(isinstance(state, int))
    assert(isinstance(width, int))
    assert(isinstance(poly, int))
    outbit = state & 1
    if not outbit:
        newstate = state >> 1
    else:
        newstate = (state ^ poly) >> 1
    return (outbit, newstate)
```

Listing 6.1: Python example for a Galois LFSR

As explained in Sect. 6.2.2, the Galois and Fibonacci construction of LFSRs are both equivalent representations. However, to produce the same bit sequence, they need to be initialized differently. In software, the Galois construction is easier to implement and has better performance. For practical reasons we, therefore, only work with the Galois representation. Note that our algorithms need to be adapted if you want to transfer our results to Fibonacci LFSRs. List. 6.1 shows the basic clocking function which for a given internal state of an LFSR with a given bit width n and polynomial returns the output bit when clocked and the resulting next state.

The polynomial is given in integer representation in this case. Consider the Galois LFSR given previously in Fig. 6.2 and a different one is shown in Fig. 6.5. The width of the polynomial, i.e., 6 or 12 bits, for the shown LFSRs, respectively, determines the term with the highest power. The other terms have a coefficient of one in those powers in which a tap exists. For the first example, this is the case for bits $4, 3, 1, 0$ while for the second case the tap exponents are $11, 10, 4, 0$. The resulting polynomial p in integer representation for a list of exponents E is given as

$$p = \sum 2^e \quad \forall e \in E$$

To illustrate, p for the first LFSR would be `0x5b` and it would be `0x1c11` for the second one.

```
state = 0xada
width = 12
poly = sum(2 ** exponent for exponent in [ 12, 11, 10, 4, 0 ])
assert(poly == 0x1c11)
sequence = [ ]
for i in range(48):
    (outbit, state) = clk_lfsr_galois(state, width, poly)
    sequence.append(outbit)
sequence_str = "".join(str(x) for x in sequence)
assert(sequence_str == "010111101010001110101100011000100111010110101110")
```

Listing 6.2: Usage of the LFSR functions

How this function can be used is shown in List. 6.2. For a given initialization value of `0xada`, and the polynomial for the LFSR shown in Fig. 6.5, it produces the shown bit sequence.

The reverse operation is performed by the function shown in List. 6.3. Once the constitution — bit length n and polynomial — of an LFSR are known, a bit sequence of length n can be fed to the function to recover the internal state at which the LFSR produces that sequence.

The usage of this recovery function is shown in List. 6.4. Note that the first twelve bits which were generated are now fed into the recovery function, and it successfully recovers the original state `0xada`. Keep in mind that a Fibonacci LFSR initialized to `0xada` produces a different sequence, and the recovery function would have to be changed to accommodate for this fact.

```python
def galois_recover_state(width, poly, desired_seq):
    assert(isinstance(width, int))
    assert(isinstance(poly, int))
    assert(len(desired_seq) == width)
    assert(all(value in [0, 1] for value in desired_seq))
    recovered_state = 1
    mask = ((1 << width) - 1)
    for attempt in range(width + 1):
        # Generate from the current state attempt
        state = recovered_state
        synth_seq = [ ]
        for i in range(width):
            (outbit, state) = clk_lfsr_galois(state, width, poly)
            synth_seq.append(outbit)

        # Calculate the disparity between desired output
        # and generated output
        discrepancy = [ (x ^ y) for (x, y)
                                in zip(desired_seq, synth_seq) ]
        try:
            # Find the bit offset of the first discrepancy
            discr_bit = discrepancy.index(1)
        except ValueError:
            # No discrepancy found, state is correct
            break

        # Invert the state bits that cause the discrepancy
        recovered_state ^= mask << discr_bit
    else:
        raise Exception("No state found that generates \
                     given bit sequence")
    recovered_state &= mask
    return recovered_state
```

Listing 6.3: Algorithm to recover internal state of a Galois LFSR

6.4.2 Calculating Memory Offsets

As mentioned before, the scrambler LFSR is parameterized by a global seed and (parts of) the memory address that is accessed. It is, therefore, vital to know the exact physical memory address of every byte in the acquired image. Unfortunately, not all acquisition software works reliably; in fact, there are many examples in which areas of memory

```
(width, poly) = (12, 0x1c11)
sequence_str = "010111101010"
sequence = [ int(x) for x in sequence_str ]
state = galois_recover_state(width, poly, sequence)
assert(state == 0xada)
```

Listing 6.4: Usage of the function to recover the internal LFSR state

which are inaccessible are simply skipped instead of being correctly filled with padding data (Vömel and Stüttgen 2013). When scanning through plaintext images to locate cryptographic keys, this is not a problem. For our purposes, however, it is not only important to get the data, but also important to be able to pinpoint the exact storage location of that data. Only then can we select the correct keystream offset with which we can descramble the image. To work around inherent limitations of acquisition software, we wrote a custom *data placer* program to store the 64-bit physical address every 8 bytes throughout all available memory. Then a soft reset was performed as not to reseed the memory scrambler and a forensic image was created with the same software which would later also capture the data images.

Note that using this approach we would also be able to determine the effects of memory address scrambling when performing acquisition on transplanted memory. While this was not the case in our experiments, there are indeed hints in literature that this would be something that could be expected in the future (Gould 2009).

By examining these dumped images, it was easy to identify the locations where the acquisition process was discontinuous. These discontinuities are referred to as *hidden memory* regions (Stüttgen 2015; Stüttgen, Vömel, and Denzel 2015). This is expected, as the BIOS memory map (which the acquisition software usually relies on) only loosely correlates with the intricate details of the actual memory mapping (which the MCH uses). As a consequence, such forensic images may contain holes within where hidden memory regions were present. Capturing exactly where these discontinuities were located for any given combination of computer and DRAM allowed us to calculate the actual address in the original physical memory of a given offset within the dumped memory image file.

6.4.3 Distinguishing the Scrambler Type

Now we know which addresses in physical memory map to an acquired image file offset, but we do not yet know about the scrambling behavior of the device under test at all. We now show how it is possible to determine how the scrambler is configured by the computer's firmware.

Here is the procedure to distinguish the different scrambler types (see Fig. 6.6):

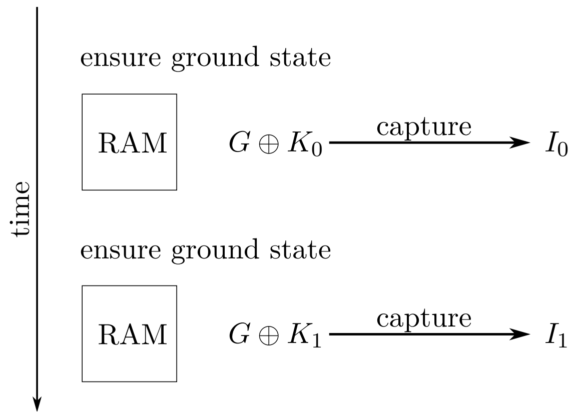

time
ensure ground state
RAM
$G \oplus K_0 \xrightarrow{\text{capture}} I_0$
ensure ground state
RAM
$G \oplus K_1 \xrightarrow{\text{capture}} I_1$

Figure 6.6: Experimental setup for image recovery.

1. Turn the device completely off and leave it off for an extended period (e.g., 1 minute). This ensures that the DRAM content is at its ground state G.
2. Turn the device on and immediately perform cold boot image acquisition.
3. Repeat these two steps twice to capture two independent cold boot images I_0 and I_1.

To determine the scrambler type, it is now sufficient to investigate the image content which has not been modified by the BIOS or dumping software. (Note that the amount of RAM that is overwritten by the BIOS on reboot needs to be evaluated on an individual basis.) When analyzing this memory, there are three possible outcomes:

1. The two captured images I_0 and I_1 are identical (except for noise) and look non-random. Long sequences of consecutive 0 and 1 bits are expected to be present in the output. The details of the pattern depend on the physical hardware wiring of the respective memory cells. This finding implies that *scrambling* is *disabled* on the machine.
2. The captured images are identical (except for noise), but look random (i.e., equal distribution of all bytes with approximately identical probability). This implies that *scrambling with a constant seed* is used on the machine. Note that disabled scrambling is a special case of constant scrambling, where the constant scrambling keystream $K_C = 0$.
3. The captured images look completely different, and also both look random. This implies that *scrambling with a random seed* is used on the machine.

In our experiments, we did not find any machine which disabled the scrambling feature altogether. However, there were machines of either of the two latter types. Which type a machine belongs to is determined by the system firmware, as explained earlier in Sect. 6.2.4. For example, we found a system consisting of an MSI H55M-P33 mainboard with an Intel Core i5-760 CPU to use constant scrambling, while an Intel Core i3-3225 CPU within an MSI B75MA-P45 mainboard used random scrambling.

6.4.4 Attacking Constant Scrambling

Assume a machine that performs constant scrambling. For such a computer, descrambling the memory contents is not necessary for most scenarios, since the scrambling and descrambling key K is, as the name suggest, constant over power cycles. Therefore, $K = K_0 = K_1$ and therefore $I = P \oplus K_0 \oplus K_1 = P$. The intuitive reason is that scrambling and descrambling cancel each other out on the same system when the keystream remains the same for both. Therefore, if the forensic image acquisition is performed on the same computer which also wrote the data into the RAM, the system can be treated as if there were no scrambling used at all. This also applies to RAM that was transplanted from one system to another one which uses the same firmware and therefore same, constant, scrambler seed.

Unfortunately, in most practical cases the machine with which the image recovery was performed is different from the computer which contains the forensically interesting data. In such cases, things become more complicated. Fortunately, for all hardware that we tested, the basic principle of how the scrambler works was always identical, so there is a reason to assume that there currently is only one generation of scrambling hardware available. This is also the case which we assume and deal with here. If the two computers use different scrambler generations, the results could vary greatly depending on the exact scrambler mechanisms which are employed.

Assuming that the two computers use at least the same scrambler generation and merely differ in the parameterization (e.g., the host system uses constant scrambling with K_0, but the acquisition system uses a different keystream K_1), then the captured image can simply be treated as if it were created on a randomized scrambling system, as described next.

6.4.5 Attacking Randomized Scrambling

Consider again the two images I_0 and I_1 from Fig. 6.6. They both capture the same, unknown, ground state G with different scrambling keys applied to them. In other words,

$$I_0 = K_0 \oplus G \quad \text{and} \quad I_1 = K_1 \oplus G.$$

We can, therefore, apply a differential approach by constructing the image D

$$D = I_0 \oplus I_1 = K_0 \oplus G \oplus K_1 \oplus G = K_0 \oplus K_1$$

By eliminating the unknown ground state from the equation we now only deal with the differential of two unknown keystreams. Since we know that the keystream is periodic, as explained in Sect. 6.2.2, it can be written as the repeated concatenation of some unknown partial keystream S (the *subkey*):

$$D = S^x$$

We then inspect chunks from this D of varying size (concretely, we used powers of two from $32 \dots 1024$). Using autocorrelation on these chunks we can identify the periodicity π of S within D. To do this, we define an equality function on two bitstreams X, Y of equal length:

$$X \approx Y \iff \frac{\mathrm{H}(X \oplus Y)}{|X|} < \epsilon$$

Here, H is the Hamming weight of a bit vector. Intuitively, we consider two bitstreams X, Y to be approximately equal if and only if the average Hamming weight of their bitwise difference is below a certain threshold ϵ. We group n of these chunks into an equivalence class:

$$\{C_0, C_1, \dots, C_{n-1}\} \quad \text{with} \quad C_i \approx C_j \, \forall \, i, j \in \{0, \dots, n-1\}$$

Once the periodicity π is selected correctly, only one equivalence class emerges with lots of approximately equal differential subkey candidates C_i, all of length π. During our experiments, we determined the smallest value for π at which this occurred to be 64 bytes.

Due to bit flips during the lossy acquisition, we still do not, however, know S. Under the assumption that all candidates C_i are just deviations from S caused by random noise during acquisition, we can calculate S by performing a majority vote on each bit s_j of S:

$$s_j = \begin{cases} 0 & \text{if } \sum\limits_{i=0}^{n-1} C_{i_j} < \frac{n}{2} \\ 1 & \text{otherwise} \end{cases}$$

As a result, we have the most likely subkey candidate S, where $D = K_0 \oplus K_1 = S^x$. We can now use this information to recover P using a known plaintext attack.

6.4.6 Stencil Attack

From our measurements, we found that on all machines we investigated, the differential of two keys K_0 and K_1 exhibited a 64-byte periodicity (i.e., $\pi = 64$). This directly enables what we refer to as the *stencil attack* for DDR3 descrambling. The attack works as follows:

1. Perform forensic recovery of the image that shall be descrambled. Without loss of generality, the memory image content M of the image P is $M = P \oplus K_0$. Here, K_0 is the keystream that is applied to the data by the scrambler unit on the target system.
2. The captured image is $I = P \oplus K_0 \oplus K_1$, with both K_0 and K_1 unknown. Note that the descrambled image P is the information of actual forensic interest and K_1 is the keystream added by the descrambler unit on the acquisition system.

3. Therefore, $I = P \oplus (K_0 \oplus K_1) = P \oplus D$, where D is still unknown. However, we know from Sect. 6.4.5 the periodicity π of D. Therefore, scan through the image I at π-byte boundaries and cluster π-byte chunks together using the approximative equality function described above. Select a partition that has lots of candidates: this is likely to be a pattern of all `0x00`. Construct the maximum likelihood candidate S by majority vote.
4. Construct $P = I \oplus S^x \oplus T^x$ where T is the known plaintext. If the known plaintext was a chunk of `0x00`, i.e., $T = 0$ then $P = I \oplus S^x$.

To reconstruct P, we therefore only need a known plaintext of length π, i.e., 64 bytes in our case.

6.4.7 Mathematical Approach

The stencil attack allows an attack to be mounted against scrambled DDR3 memory, effectively yielding the original image with relatively few pieces of known plaintext required. However, we now look at the mathematical relationships within the differential subkey stream with the purpose of constructing a keystream from less known plaintext.

In our approach we are limited to examination of a differential keystream $K_0 \oplus K_1$. This is as an inherent limitation of performing acquisition with systems which contain an active descrambler. During analysis, we found some interesting congruencies within this differential keystream. First, we partitioned the 64-byte differential keystream stencil into 32 values of 2 bytes each. Each value was interpreted as a little endian integer. We ended up with 32 16-bit integer values $v_0, \ldots, v_{31}$. We then were able to find three 16-byte polynomials p_0, p_1, p_2 for which 24 congruencies hold for $0 \le i \le 7$ and $0 \le j \le 2$:

$$((v_{4i+j} >> 1) \oplus p_j) \;\&\; \texttt{0x7fff} = v_{4i+j+1}$$

If you recall Sect. 6.2.2, this relationship can immediately be recognized as a LFSR relationship: Two related values, v_{4i+j} and its adjacent word v_{4i+j+1} can be constructed from each other when the first one is shifted right by one bit ($>> 1$) and then has the LFSR polynomial added onto it ($\oplus p_j$). Note however that in this congruence we can only determine 15 of the 16 bits of the adjacent word (& `0x7fff`). This is because it is lost when the difference between two LFSR output streams is constructed.

These relationships are a useful additional method to confirm the validity of keystreams and aid the search for a known plaintext in the differential memory snapshot D. It reduces the number of known plaintext bits to less than 40 % of the original stencil attack: in our case, instead of 64 bytes we now need only $(8+3) \cdot 2 + 3 = 25$ bytes if we exploit the mathematical inter-stream relationships. Only the eight initial states v_{4i} (16 bytes), the three polynomials $p_0 \ldots p_3$ (6 bytes) and the three most significant bits in every group of 8 bytes (total of 3 bytes) need to be known to construct the entire differential keystream.

6.4.8 Deinterleaving of Memory

As stated before in Sects. 6.2.3 and 6.2.4 a system with more than one DRAM module usually operates in dual-channel mode to improve system performance. Each memory channel has an independent scrambler, so the attack as described in Sect. 6.4.6 still works when it is known which part of the memory image needs to be descrambled with which channel key. In our experiments, we determined how the algorithm to split data between channels works in Intel systems. Consider two channel subkeys A and B of 64 bytes each. The two basic interleaved streams Q_1 and R_1, each of length 256 bytes, are defined to be:

$$Q_1 = A^2 \,||\, B^2$$
$$R_1 = B^2 \,||\, A^2$$

All further keystreams are then defined recursively:

$$Q_n = Q_{n-1} \,||\, R_{n-1}^2 \,||\, Q_{n-1}$$
$$R_n = R_{n-1} \,||\, Q_{n-1}^2 \,||\, R_{n-1}$$

This definition can be applied until one finds a Q_n of sufficient length. This Q_n is then the complete keystream K.

Since every channel subkey is 64 bytes in length, the length of an interleaved keystream Q_i is exactly:

$$|Q_i| = 64 \cdot 4^i$$

Solving for i with given $|Q_i|$:

$$i = \log_4 \frac{|Q_i|}{64}$$

i.e., for a dual-channel memory system of 4 GiB, one would choose $i = 13$.

The interleaving pattern for $i = 6$, i.e., for 4096 streams of 64 bytes each, is graphically shown in Fig. 6.7. It is exactly 64 by 64 tiles in size, and every color of the tile indicates whether stream A or stream B is in effect. The stream order is shown left-to-right, top-to-bottom.

With this knowledge, we can perform the stencil attack even when RAM is accessed in dual channel mode. The acquired image I has to be deinterleaved into two channel images I_A and I_B which can be treated independently before interleaving them back together to form a plain image P.

6.5 Experimental Evaluation

6.5.1 Investigated Machines

Our measurements were performed predominantly on the Intel Core i3-3225 with an MSI B75MA-P45 mainboard. We took care to verify that the results also apply to different machines. In the process, we confirmed that our results also apply to the following combination of CPUs and mainboards:

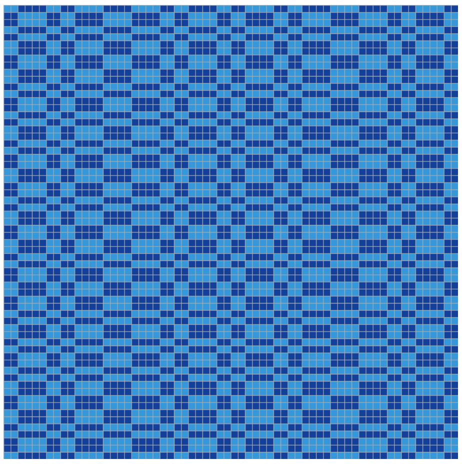


Figure 6.7: Dual channel interleaving graphically

- Intel i5-760/MSI H55M-P33
- i5-2520M/Dell 03PH4G
- i5-2400/Esprimo P900 E90+

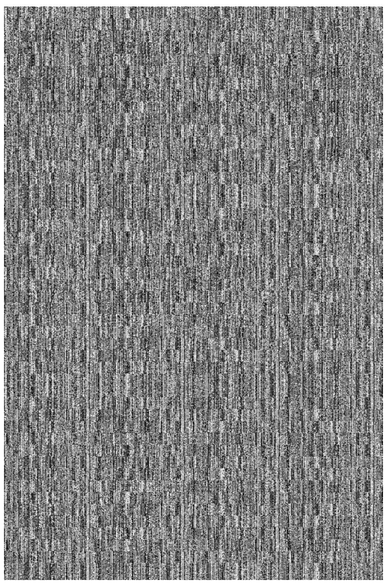
6.5.2 Applying the Stencil Attack

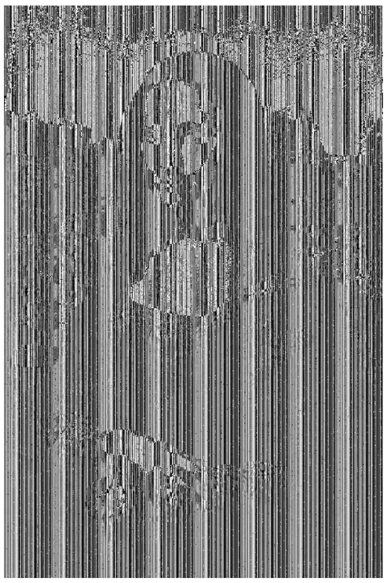
In our experiments, we first started out with a single memory module present on the target computers. We then used our data placer code to place 512x775 pixel grayscale images of Mona Lisa at every 1 MiB boundary. At the space in between images, we placed an easily recognizable, distinct pattern block.

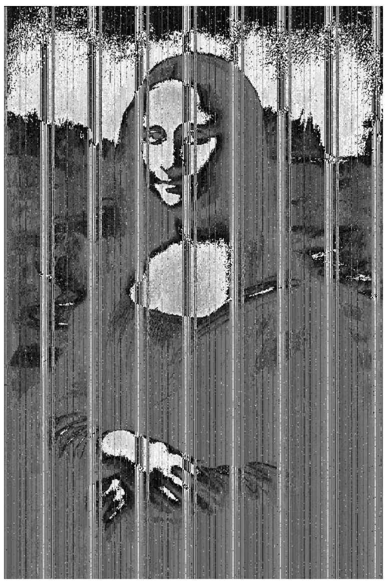
We then froze the memory module by applying cooling spray to it until it had reached around -30 °C. Then we cut power to the system by shutting it completely off and restarted immediately afterward. The latency that our targets took from complete shutdown to new boot-up ranged from around 2 to up to 5 seconds. We then drew memory images using the memimage toolkit. By using the techniques described in Sect. 6.4.6 we were able to recover the original memory image.

You can see the results of our experiments in Fig. 6.8. The first image, Fig. 6.8a shows an image acquisition that was performed at operating temperature (about +30 °C). No data could be recovered from this test, as everything was completely decayed.

On the second image, Fig. 6.8b, the image is shown when it was drawn from the target after cooling to about -30 °C was applied. The basic shape of Mona Lisa is still visible, but it is distorted by a repeating pattern. This repeating pattern is exactly the subkey stencil which we need to apply to descramble the memory images. When this key is unknown (because there is no known plaintext or at least no known plaintext yet) we did a related data experiment, shown in Fig. 6.8c. For this, we make the assumption that


(a) Scrambled image captured at +30 °C


(b) Scrambled image captured at -30 °C


(c) Related-data descrambling

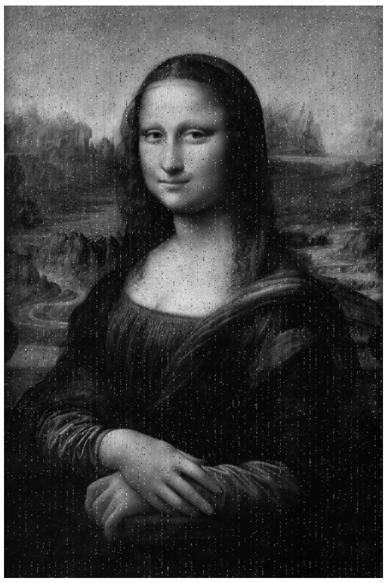

(d) Stencil descrambling

Figure 6.8: Images of descrambling single-channel memory

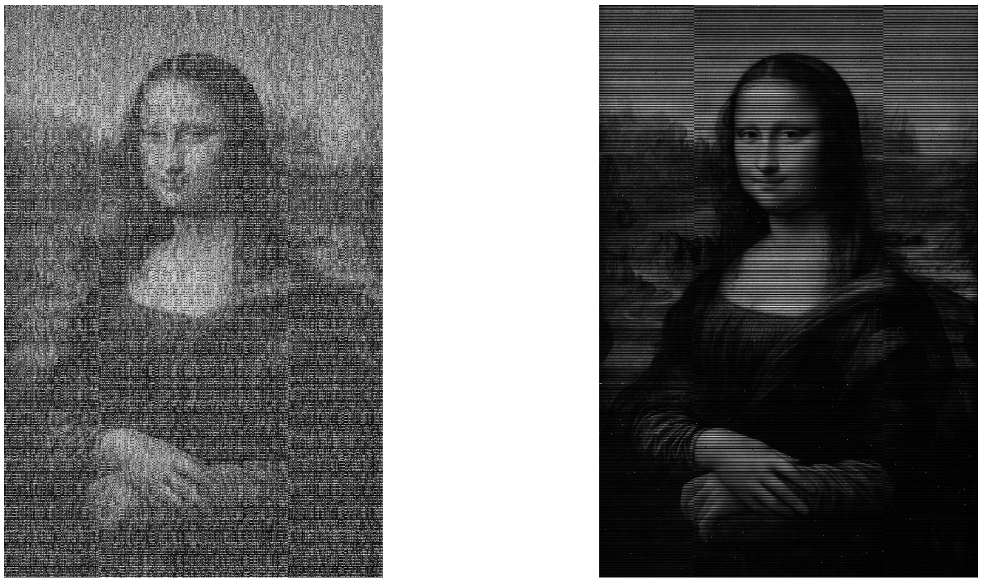

(a) Interleaved dual channels (b) Deinterleaved masked image

Figure 6.9: Images of descrambling dual-channel memory

consecutive 64-byte plaintexts — at least to some degree — repeat within the plaintext. Therefore, the first 64-byte block was chosen as a stencil subkey. You can see clearly that the image looks a lot better than the completely scrambled variant, but still has lots of distortions.

Finally, we used the method described in Sect. 6.4.6 and recovered the most probable key using majority vote and known plaintext. This key was then applied to the captured image, yielding a result that was, except for the occasional bit error, very close to the original image.

6.5.3 Dual Channel Mode and Decay Rate

During our experiments, we found the high decay rate of DDR3 RAM when at operating temperature curious. To verify our hypothesis that the retention time of memory was indeed much less than what can be seen in DDR2 counterparts, we performed an experiment: We placed two memory modules in our target PC and took care that they were operated in dual-channel mode. Then we placed the Mona Lisa images in RAM using our data placer program. At this point in time, roughly every second 64-byte block of the image is on one RAM module and every other 64-byte block is on the other RAM module. We then inserted a rectangular piece of 500µm PET (polyethylene terephthalate) in between the two RAM modules to achieve thermal isolation between them. One of the two modules — but not the other — was then frozen by us before performing a cold boot attack.

This experiment served a dual purpose: First, it allowed us to confirm that the thermal dependence of DDR3 is as critical as we assumed it was. This is because the only difference in the process was the temperature of the modules — all other parameters like the power-down time were exactly identical. Secondly, it allowed us to confirm that our algorithm to decode dual-channel memory, as explained in Sect. 6.4.8, worked as expected.

The results are depicted in Fig. 6.9. On the left side, Fig. 6.9a shows the interleaved, descrambled, memory image. The results verified our hypothesis: Approximately every other row was completely decayed and shows up as white noise in the image. Even though the image in Fig. 6.9a gives a rather noisy visual impression this noise is only present in the parts that were acquired from the warm RAM module. This becomes obvious when the noisy channel is masked out according to the algorithm we presented in Sect. 6.4.8. The result is Fig. 6.9b, a successfully descrambled one-channel image of a RAM module that was operated in dual-channel mode. The image shows virtually no noise because our algorithm correctly masks out only the module which had decayed content.

6.5.4 Remanence Effect of DDR3 Memory

In their original paper, Halderman et al. (2009) found that DDR2 RAM exhibits a comparatively strong remanence effect. They performed tests at operating temperature and even without externally applied cooling to the chips, some DDR2 chips had decay times of up to 35 seconds before showing complete data loss. To determine these values for DDR3 memory, Lindenlauf, Höfken, and Schuba (2015) did similar decay measurements. They found an astonishingly low bit error rate when the modules were cooled to a temperature between -30 °C and -35 °C and kept the RAM modules unpowered for up to 50 seconds. It is not apparent, however, if these long decay times were also performed with DDR3 memory or just with DDR2, and while they do show the dependence of the decay rate on the die temperature for DDR2 memory they omit how these results transfer to DDR3 memory.

In our experiments, we found DDR3 memory to be much less forgiving during cold booting. Much in contrast to DDR2 memory it was essential for us to keep modules always at low temperatures (around -30 °C) to produce usable results. We have observed retention times of about 10 seconds before total decay occurred even when we applied such cooling. At operating temperature (around +30 °C) we were not able to acquire a single usable image because all data content had dissipated.

It is our assumption that this can be explained by the different types of memory modules that were used by Lindenlauf, Höfken, and Schuba (ibid.) compared to ours. While they used modules that were produced in 2011, we used slightly more recent modules (produced 2013) that were also a bit faster (666 and 833 MHz types).

6.6 Conclusion and Outlook

Memory acquisition of DDR3 memory in the real world is more complicated than with a laboratory setup: While in a lab setup researchers can choose systems which work for their demonstration — systems which usually use constant scrambling — this luxury is not available in a real-world scenario. On top of the intricacy of descrambling images come practical aspects like dual-channel decoding. Both are obstacles that are not in place to deter cold boot attacks, but they still complicate memory acquisition significantly in practice.

We have demonstrated that our explanations and assumptions about the internal construction of the Intel DRAM scrambler are in line with the observations we made from our experimental results. Cracking a dual-channel system requires only 128 bytes of known plaintext to apply our stencil method and only 50 bytes if the mathematical approach is chosen. This is a negligible amount of data compared to the huge amount of RAM that is present in the computer systems of today. Large chunks of the RAM usually are set to zero during regular operation of a computer, be it either by the operating system or by any running application. We further demonstrated that we could correctly deinterleave RAM images. This is a prerequisite to correct descrambling of dual-channel systems, as each channel has an independent scrambler.

Since the operation of memory scramblers is transparent for the system during normal operation, it could well be possible that newer MCH revisions choose to use different mechanisms for scrambling. Of particular interest for a forensic investigation would be if our results can be applied to DDR4 memory as well. This is something we would like to explore in future work.

To improve on our attack, it would be most interesting to mathematically attack the generated keystream itself. Since our approach only works with differential streams we at no point in time could reconstruct the original keystream — we only ever reconstruct differential keystreams. Our ideas for future work are to utilize custom-built hardware around an FPGA development board to be able to read out the raw keystream from a cold booted DDR3 memory module. This would then enable brute forcing of keystreams by trying different seeds, but it would also be an attack that would be significantly more difficult than what we show in this chapter.

Investing this time would be interesting not only from an academical standpoint but also because of the potential real-world implications. The reason that scramblers are present in the first place is that Intel deemed it necessary to limit excessive current spikes on the memory bus and in the memory modules. This leads us to believe that there could be possibly exploitable detrimental effects if one could purposefully produce these excessive current spikes. If the scrambling keystream were known to an attacker, this could be leveraged from any unprivileged application to mount an attack which aims to distort RAM integrity. Since disturbing RAM-integrity is a relevant topic and is receiving increased attention after the inspiring row hammer attacks of Kim et al. (2014), this might prove to be a worthwhile investment of time after all.

Chapter 7

Conclusion

Protecting hardware against physical attacks is a difficult task. Even though we can try to protect secrets inside a microchip against unauthorized extractions — such secrets can be cryptographic keys or intellectual property — then a sufficiently motivated and equipped adversary is likely to be able to extract them eventually. Techniques such as those reviewed and described by Giannuzzi and Stevie (1999) — utilization of a Focused Ion Beam (FIB) — are tools which are extraordinarily powerful for an attacker. While they are exceedingly expensive for a hobbyist budget, any decently equipped hardware laboratory that does hardware analysis needs to have one to perform day-to-day tasks effectively. Naturally, state-level actors are always in possession of this type of equipment.

So while an attacker might not be able to afford to buy one, such tools are not out of reach for use. Moreover, even in the cases in which they are not in reach, small budget attacks have become increasingly sophisticated as well. Attacks like that of C. Miller and Valasek (2015) are executed exceptionally well. In their work, they hack the systems of a Jeep Cherokee to the point where they can remotely control its driving. The publication got much attention, primarily because the connected safety issues are immediately apparent. A hacker with the possibility to drive your car into a ditch remotely can be a dangerous adversary. However, it also serves as an example of the type of hackers which systems need to be defended against; while undoubtedly skilled at what they do, the resources they used to implement their hack were surprisingly undemanding. They needed no fancy equipment, but just time, competence and determination.

It almost seems like attackers can compensate their lack of budget with creativity and new approaches to crack systems open. For physical attacks, the adversary has an arbitrary amount of time to prepare and think about ways to break into a system. Time, in this case, works against the defensive side. An additional issue is that emergent technologies can often be vulnerable by new attack vectors; one such example is the work of Jang and Ghosh (2016). In their work, they explore attacks on RAM technology that internally relies on ferromagnetic effects. Such RAM can be externally stimulated by magnetic fields to show disturbances. It is easily imaginable how such hardware constraints can be exploited by an adversary to extract cryptographic keys from a system that relies on this technology.

7.1 Absence of a Security Silver Bullet

At the beginning of this thesis, we wrote about the difficulty of estimating a level of security and designing a security system that fits the needs of its designer. Moreover, again, the same that goes for all disciplines of engineering holds up here as well: There is no silver bullet for the design of a system to become secure automatically. However, it is easy to get the false impression that hardware security modules are such a silver bullet. Unfortunately, this is too good to be true; while hardware security measures may have the greatest theoretical potential for offering protection, it is difficult to decide objectively what is fact and what are just hollow marketing claims. Since vendors of such hardware deliberately try to keep the internals of their product a secret, adding to the difficulty of getting accurate information. Unless a customer is prepared to buy a large quantity of such security hardware, getting the right information is a painstakingly tedious process.

Another issue that surrounds the commercial implementation of hardware countermeasures is that they are encumbered with patents like few other areas of security. This makes employing such countermeasures costly for the vendor because of licensing fees. Instead of paying the price, there is a shocking number of devices which are marketed under the label “high security”, but which do not, for example, provide countermeasures against differential power analysis. There is no denying that attacks like power analysis are becoming more frequent and easy to conduct; the value of the work of O’Flynn and Chen (2014) cannot be overstated to underline this fact. Consequently, this could mean that a general-purpose microcontroller with software countermeasures against PA might offer better protection than a supposedly “high security” device which does not offer these countermeasures.

Also, real-world requirements of industry production may not be underestimated. Secure hardware is one part of the jigsaw puzzle, but the surrounding equipment and processes must at least meet that same level of protection. This means secure — usually proprietary — hardware programming devices have to be acquired, and the firmware has to be stored in them securely. These proprietary programming adapters can be costly and difficult to acquire since the person who tries to order them usually needs to adhere strictly to the silicon vendor’s protocol. For industry production, having a second source is also a good argument against proprietary hardware security modules. Few, if any, proprietary interfaces are the same in this sector. The manufacturer, therefore, puts all eggs in one basket as soon as the particular security IC is part of the design. The whole production relies on that chip being available by that one supplier. This dependency is the sword of Damocles hanging over the manufacturing process because any delayed delivery of the HSM ICs can grind the whole production to a halt.

All in all, hardware security modules surely are not the solution that automatically ensures a system becomes immune to attacks. From a technical perspective, they do have the advantage regarding potential defensive capabilities. However, there are other constraints on the design of a secure system which might take priority. What it ultimately boils down to is the type of attacker and the type of attack against which we are trying to defend. Only as soon as this is clear, we can decide sensibly which measures are appropriate to fulfill our constraints.

7.2 Security Silver Linings

While there is no simple “one size fits all” solution to defend against the exceeding number of different attacks, there however still is a silver lining. Instead of surrendering to the fact that even hobbyist hackers today are capable enough to hack systems of large corporations, the focus should be to *raise the bar* for these attacks. In other words, while the employed countermeasures are never able to defend against all attacks, they still can be considered a significant improvement if they deter at least most of the attackers. An example of this is the implementation of Sony’s PlayStation 3 security. While their security was severely flawed and ultimately broken into by bushing et al. 2010 (sic), the various layers of defense employed by Sony held up about four years. Considering that the time between new console models is around six years, the countermeasures that were used by Sony have been useful to at least some reasonable degree.

Again, it depends on the types of attacks that a system needs to be defended against and choosing the appropriate countermeasures for that particular attacker model. To underline this, we provide practical measures that can be used to harden the resistance of systems against physical attacks. An approach like the one we showed in Chap. 2 is, for example, useful to defend against passive attacks like power analysis. The concrete primitives which we used to obfuscate power emission are just to be taken as one possible example; the interesting part of our contribution is the versatility of our minimal virtual machine approach. It can effortlessly be augmented by different types of code substitutions and it can likewise be used for any architecture which allows code execution from runtime writable memory since the actual VM code is completely agnostic to the underlying assembly dialect. This means that our actual code to run the VM can be reused and only the tools which run on the host and output the VM bytecode need to be adapted.

While it may seem trivial, timing attacks also offer great attack surface if the system designed does not carefully avoid timing side channels. It, therefore, is advisable for the defensive side to evaluate if such problems are present in the own design. To aid this, we show how cycle accurate timing simulation can help to detect such issues. The Cortex-M core emulator we introduce in Chap. 3 is meant to be used for exactly that purpose. Our work highlights the complexity of modern microcontrollers and how nonintuitive their timing behavior can be. Since it, therefore, is not a viable option to manually check relevant code, our emulator takes that tedious task away from the user. It enables the implementer to focus on the aspects which cannot automatically be determined: Which functions are allowed to have variable runtime and which need to exhibit constant runtime behavior in order not to compromise the system security.

As soon as the need arises to incorporate proprietary external hardware into a design to further strengthen the overall security niveau, the difficulty of consolidating APIs, which are frequently vastly different, arises. How we can solve such an issue from a technical aspect is something we demonstrated in Chap. 5. While the concrete implementation is something that is unique to both the protocol and hardware we are trying to use — in our case, TLS-PSK with an Atmel HSM — the general ideas which we show apply to a wide variety of different protocols and hardware. These ideas may serve as a template for similar scenarios in which the designer of a system does not want to rely entirely on the security of a general purpose microcontroller alone.

Lastly, one thing that was important for us to highlight is the duality of added complexity and functionality on one hand and the relevance of secure external peripherals on the other. For this, we explore the former in Chap. 4 in which we show how entirely benign anti-EMI functionality could be abused by an adversary to create a covert channel which is difficult to detect because it operates on the analog-digital signal encoding ambiguity. The external peripherals which we analyze are DRAM chips within PC systems with an Intel CPU – for these systems, as shown in Chap. 6, the memory content is usually stored in scrambled form on the DRAM module itself. However, since the memory controller is not meant to prevent descrambling, but just aims to provide basic debiasing of the input data, we were able to descramble to memory contents and potentially use the knowledge to leverage disturbance errors in RAM.

It is our belief that this work achieves two goals: Firstly, we would like to raise awareness about the increasing simplicity with which physical attacks on embedded nodes can be performed today. In our opinion, physical attacks are still an underestimated and rising security risk and often unfairly disregarded as something that can only be accomplished by experts. Secondly, we would like to see our ideas be used constructively to harden embedded systems against physical attacks in scenarios in which — primarily due to cost constraints — no similar hardening techniques could have been used. Since in the upcoming years the number of deeply embedded nodes has been estimated to rise tremendously this likewise means a great increase in additional attack surface. Protection of this attack surface is something that we should not ignore — more importantly, it is something we simply cannot afford to ignore.

Bibliography

Adrian, David, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, John Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann (2015). "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice". In: *22nd ACM Conference on Computer and Communications Security—CCS '15.* Denver, Colorado, USA: ACM, pp. 5–17. ISBN: 978-1-450-33832-5. DOI: 10.1145/2810103.2813707. URL: https://dx.doi.org/10.1145/2810103.2813707.

Agosta, Giovanni, Alessandro Barenghi, and Gerardo Pelosi (2012). "A code morphing methodology to automate power analysis countermeasures". In: *49th Annual Design Automation Conference—DAC '12.* New York, NY, USA: ACM, pp. 77–82. DOI: 10.1145/2228360.2228376. URL: https://dx.doi.org/10.1145/2228360.2228376.

Agosta, Giovanni, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale (2015). "The MEET Approach: Securing Cryptographic Embedded Software Against Side Channel Attacks". In: *IEEE Transactions on CAD of Integrated Circuits and Systems* 34.8, pp. 1320–1333. DOI: 10.1109/TCAD.2015.2430320. URL: https://dx.doi.org/10.1109/TCAD.2015.2430320.

Agrawal, Dakshi, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi (2003). "The EM Side-channel(s): Attacks and Assessment Methodologies". In: *4th International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2002.* Ed. by Burton S. Kaliski, Çetin K. Koç, and Christof Paar. Berlin, Heidelberg, Germany: Springer, pp. 29–45. ISBN: 978-3-540-36400-9. DOI: 10.1007/3-540-36400-5_4. URL: https://dx.doi.org/10.1007/3-540-36400-5_4.

Al Fardan, Nadhem J and Kenneth G Paterson (2013). "Lucky thirteen: Breaking the TLS and DTLS record protocols". In: *IEEE Symposium on Security and Privacy—SP 2013.* IEEE, pp. 526–540. DOI: 10.1109/SP.2013.42. URL: https://dx.doi.org/10.1109/SP.2013.42.

Altera Inc. (2015). *External Memory Interface Handbook Volume 1: Altera Memory Solution Overview and Design Flow.* URL: https://www.altera.com/literature/hb/external-memory/emi.pdf.

American National Standards Institute (2001). *ANSI X9.63 Public Key Cryptography for the Financial Services Industry: Elliptic Curve Key Agreement and Key Transport Schemes.*

Antipa, Adrian, Daniel Brown, Alfred J Menezes, René Struik, and Scott A Vanstone (2002). "Validation of elliptic curve public keys". In: *6th International Workshop on*

Practice and Theory in Public Key Cryptography—PKC 2003. Ed. by Yvo G. Desmedt. Berlin, Heidelberg, Germany: Springer, pp. 211–223. ISBN: 978-3-540-36288-3. DOI: `10.1007/3-540-36288-6_16`. URL: `https://dx.doi.org/10.1007/3-540-36288-6_16`.

Aoki, Kazumaro, Jian Guo, Krystian Matusiewicz, Yu Sasaki, and Lei Wang (2009). "Preimages for step-reduced SHA-2". In: *Advances in Cryptology—ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security*. Ed. by Mitsuru Matsui. Berlin, Heidelberg, Germany: Springer, pp. 578–597. ISBN: 978-3-642-10366-7. DOI: `10.1007/978-3-642-10366-7_34`. URL: `https://dx.doi.org/10.1007/978-3-642-10366-7_34`.

Appelbaum, Jacob, Judith Horchert, and Christian Stöcker (2013). "Shopping for Spy Gear: Catalog Advertises NSA Toolbox". In: *Spiegel Online International*. URL: `http://www.spiegel.de/international/world/catalog-reveals-nsa-has-back-doors-for-numerous-devices-a-940994.html`.

ARM Ltd. (2010). *Cortex-M4 Technical Reference Manual Revision r0 Part p0*. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439b/DDI0439B_cortex_m4_r0p0_trm.pdf`.

ARM Ltd. (2014). *ARMv7-M Architecture Reference Manual DDI 0403E.b (ID 120114)*. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0403e.b/`.

Atmel Inc. (2015a). *ATECC508A CryptoAuthentication summary datasheet*. URL: `http://www.atmel.com/images/atmel-8923s-cryptoauth-atecc508a-datasheet-summary.pdf`.

Atmel Inc. (2015b). *ATSHA204A CryptoAuthentication datasheet*. URL: `http://www.atmel.com/Images/Atmel-8885-CryptoAuth-ATSHA204A-Datasheet.pdf`.

Atmel Inc. (2016). *ATECC108A CryptoAuthentication summary datasheet*. URL: `http://www.atmel.com/Images/Atmel-8895S-CryptoAuth-ATECC108A-Datasheet-Summary.pdf`.

Aumasson, Jean-Philippe (2006). *On the pseudo-random generator ISAAC*. IACR Cryptology ePrint Archive, Report 2006/438. URL: `http://eprint.iacr.org/2006/438`.

Aviram, Nimrod, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, John Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt (2016). "DROWN: Breaking TLS using SSLv2". In: *25th USENIX Security Symposium—USENIX Security 16*. URL: `https://drownattack.com/drown-attack-paper.pdf`.

Babar, Sachin, Antonietta Stango, Neeli Prasad, Jaydip Sen, and Ramjee Prasad (2011). "Proposed embedded security framework for Internet of Things (IoT)". In: *2nd Inter-*

national Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology—Wireless VITAE 2011. IEEE, pp. 1–5. DOI: `10.1109/WIRELESSVITAE.2011.5940923`. URL: `https://dx.doi.org/10.1109/WIRELESSVITAE.2011.5940923`.

Badra, M. (2009). *Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode (RFC5487)*. DOI: `10.17487/rfc5487`. URL: `https://tools.ietf.org/rfc/rfc5487.txt`.

Bagci, Ibrahim Ethem, Mohammad Pourmirza, Shahid Raza, Utz Roedig, and Thiemo Voigt (2012). "Codo: Confidential data storage for wireless sensor networks". In: *9th International Conference on Mobile Adhoc and Sensor Systems—MASS 2012*. IEEE, pp. 1–6. DOI: `10.1109/MASS.2012.6708508`. URL: `https://dx.doi.org/10.1109/MASS.2012.6708508`.

Barker, Elaine (2016). *Recommendation for Key Management - Part 1: General (Revision 4)*. NIST Special Publication 800-57 Part 1, Revision 4. URL: `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf`.

Bauer, Johannes and Felix C Freiling (2015). "Schutz eingebetteter Systeme gegen physische Angriffe". In: *DACH Security 2015 – Bestandsaufnahme - Konzepte - Anwendungen - Perspektiven*. Bonn, Germany: syssec-Verlag, pp. 387–396. ISBN: 978-3-000-49965-4.

Bauer, Johannes and Felix C Freiling (2016). "Towards Cycle-Accurate Emulation of Cortex-M Code to Detect Timing Side Channels". In: *11th International Conference on Availability, Reliability and Security—ARES 2016*. IEEE. DOI: `10.1109/ARES.2016.94`. URL: `https://dx.doi.org/10.1109/ARES.2016.94`.

Bauer, Johannes, Michael Gruhn, and Felix C Freiling (2016). "Lest we forget: Cold-boot attacks on scrambled DDR3 memory". In: *Digital Investigation* 16, S65–S74. DOI: `10.1016/j.diin.2016.01.009`. URL: `https://dx.doi.org/10.1016/j.diin.2016.01.009`.

Bauer, Johannes, Sebastian Schinzel, Felix C Freiling, and Andreas Dewald (2016a). "Information Leakage behind the Curtain: Abusing Anti-EMI Features for Covert Communication". In: *IEEE International Symposium on Hardware Oriented Security and Trust—HOST 2016*, pp. 130–134. DOI: `10.1109/HST.2016.7495570`. URL: `https://dx.doi.org/10.1109/HST.2016.7495570`.

Bauer, Johannes, Sebastian Schinzel, Felix C Freiling, and Andreas Dewald (2016b). *Information Leakage behind the Curtain: Abusing Anti-EMI Features for Covert Communication*. Tech. rep. CS-2016-03. University of Erlangen, Department of Computer Science 1. URN: `urn:nbn:de:bvb:29-opus4-71576`. URL: `https://nbn-resolving.org/urn:nbn:de:bvb:29-opus4-71576`.

Bayrak, Ali Galip, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne (2015). “Automatic Application of Power Analysis Countermeasures”. In: *IEEE Transactions on Computers* 64.2, pp. 329–341. ISSN: 0018-9340. DOI: 10.1109/TC.2013.219. URL: https://dx.doi.org/10.1109/TC.2013.219.

Bernstein, Daniel J (2005). *Cache-timing attacks on AES.* URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

Beurdouche, Benjamin, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue (2015). “A messy state of the union: Taming the composite state machines of TLS”. In: *IEEE Symposium on Security and Privacy—SP 2015.* IEEE, pp. 535–552. DOI: 10.1109/SP.2015.39. URL: https://dx.doi.org/10.1109/SP.2015.39.

Bitmain Technologies Ltd. (2016). *AntMiner S7 Manual.* URL: https://www.bitmaintech.com/files/download/Antminer+S7+user+guide.pdf.

Blake-Wilson, Simon, Nelson Bolyard, Vipul Gupta, Chris Hawk, and Bodo Möller (2006). *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) (RFC4492).* DOI: 10.17487/rfc4492. URL: https://tools.ietf.org/rfc/rfc4492.txt.

Bogdanov, Andrey, Dmitry Khovratovich, and Christian Rechberger (2011). “Biclique cryptanalysis of the full AES”. In: *Advances in Cryptology—ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security.* Ed. by Dong Hoon Lee and Xiaoyun Wang. Berlin, Heidelberg, Germany: Springer, pp. 344–371. ISBN: 978-3-642-25385-0. DOI: 10.1007/978-3-642-25385-0_19. URL: https://dx.doi.org/10.1007/978-3-642-25385-0_19.

Bracewell, Ronald (1999). *The Fourier Transform and its Applications.* 3rd. McGraw-Hill Publishing Company. ISBN: 978-0-071-16043-8.

Brachmann, Martina, Sye Loong Keoh, Óscar García-Morchón, and Sandeep S Kumar (2012). “End-to-end transport security in the IP-Based Internet of Things”. In: *21st International Conference on Computer Communications and Networks—ICCCN 2012.* IEEE, pp. 1–5. DOI: 10.1109/ICCCN.2012.6289292. URL: https://dx.doi.org/10.1109/ICCCN.2012.6289292.

Brier, Eric, Christophe Clavier, and Francis Olivier (2004). “Correlation power analysis with a leakage model”. In: *6th International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2004.* Ed. by Marc Joye and Jean-Jacques Quisquater. Berlin, Heidelberg, Germany: Springer, pp. 16–29. ISBN: 978-3-540-28632-5. DOI: 10.1007/978-3-540-28632-5_2. URL: https://dx.doi.org/10.1007/978-3-540-28632-5_2.

Bucci, Marco, Michele Guglielmo, Raimondo Luzzi, and Alessandro Trifiletti (2004). “A power consumption randomization countermeasure for DPA-resistant cryptographic processors”. In: *14th International Workshop on Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation—PATMOS 2004*. Ed. by Enrico Macii, Vassilis Paliouras, and Odysseas Koufopavlou. Springer, pp. 481–490. ISBN: 978-3-540-30205-6. DOI: `10.1007/978-3-540-30205-6_50`. URL: `https://dx.doi.org/10.1007/978-3-540-30205-6_50`.

bushing, marcan, segher, and sven (2010). “Console Hacking 2010 — PS3 Epic Fail”. In: *27th Chaos Communication Congress—27C3*. URL: `https://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf`.

Cabuk, Serdar, Carla E Brodley, and Clay Shields (2004). “IP covert timing channels: Design and detection”. In: *11th ACM Conference on Computer and Communications Security—CCS ’04*. Washington DC, USA: ACM, pp. 178–187. ISBN: 978-1-581-13961-7. DOI: `10.1145/1030083.1030108`. URL: `https://dx.doi.org/10.1145/1030083.1030108` (visited on 09/21/2012).

Clavier, Christophe, Jean-Sébastien Coron, and Nora Dabbous (2000). “Differential power analysis in the presence of hardware countermeasures”. In: *2nd International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2000*. Ed. by Çetin K. Koç and Christof Paar. Berlin, Heidelberg, Germany: Springer, pp. 252–263. ISBN: 978-3-540-44499-2. DOI: `10.1007/3-540-44499-8_20`. URL: `https://dx.doi.org/10.1007/3-540-44499-8_20`.

Copeland, Jack (2010). *Colossus: The secrets of Bletchley Park’s code-breaking computers.* Oxford University Press. ISBN: 978-0-199-57814-6.

Coron, Jean-Sébastien (1999). “Resistance against differential power analysis for elliptic curve cryptosystems”. In: *1st International Workshop on Cryptographic Hardware and Embedded Systems—CHES 1999*. Ed. by Çetin K. Koç and Christof Paar. Berlin, Heidelberg, Germany: Springer, pp. 292–302. ISBN: 978-3-540-48059-4. DOI: `10.1007/3-540-48059-5_25`. URL: `https://dx.doi.org/10.1007/3-540-48059-5_25`.

Coron, Jean-Sébastien and Louis Goubin (2000). “On Boolean and Arithmetic Masking against Differential Power Analysis”. English. In: *2nd International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2000*. Ed. by Çetin K. Koç and Christof Paar. Vol. 1965. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, pp. 231–237. ISBN: 978-3-540-41455-1. DOI: `10.1007/3-540-44499-8_18`. URL: `https://dx.doi.org/10.1007/3-540-44499-8_18`.

Croucher, John S (2006). “Collecting coupons — a mathematical approach”. In: *Australian Senior Mathematics Journal* 20.2, pp. 31–35. ISSN: 0819-4564. DOI: `1959.14/14137`. URL: `https://dx.doi.org/1959.14/14137`.

Daemen, Joan and Vincent Rijmen (1999). “Resistance against implementation attacks: A comparative study of the AES proposals”. In: *The Second AES Candidate Conference*, pp. 122–132. URL: http://csrc.nist.gov/archive/aes/round1/conf2/papers/daemen.pdf.

Dhem, Jean-François, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems (1998). “A practical implementation of the timing attack”. In: URL: https://www.uclouvain.be/crypto/services/download/publications.pdf.ba2a6ad854f479a8.7064663137332e706466.pdf.

Dhem, Jean-François, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems (2000). “A practical implementation of the timing attack”. In: *Third International Conference on Smart Card Research and Applications—CARDIS ’98*. Ed. by Jean-Jacques Quisquater and Bruce Schneier. Springer, pp. 167–182. ISBN: 978-3-540-44534-0. DOI: 10.1007/10721064_15. URL: https://dx.doi.org/10.1007/10721064_15.

Dierks, Tim and Eric Rescorla (2008). *The Transport Layer Security (TLS) Protocol Version 1.2 (RFC5246)*. DOI: 10.17487/rfc5246. URL: https://tools.ietf.org/rfc/rfc5246.txt.

Dietrich, Christian J, Christian Rossow, Felix C Freiling, Herbert Bos, Maarten van Steen, and Norbert Pohlmann (2011). “On Botnets that use DNS for Command and Control”. In: *7th European Conference on Computer Network Defense—EC2ND ’11*. Washington, DC, USA: IEEE, pp. 9–16. ISBN: 978-0-769-54762-6. DOI: 10.1109/EC2ND.2011.16. URL: https://dx.doi.org/10.1109/EC2ND.2011.16.

Diffie, Whitfield and Martin E Hellman (1976). “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6, pp. 644–654. DOI: 10.1109/TIT.1976.1055638. URL: https://dx.doi.org/10.1109/TIT.1976.1055638.

Duong, Thai and Juliano Rizzo (2011). *Here come the ⊕ ninjas*. URL: http://netifera.com/research/beast/beast_DRAFT_0621.pdf.

Eisenbarth, Thomas, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T Manzuri Shalmani (2008). “On the power of power analysis in the real world: A complete break of the KeeLoq code hopping scheme”. In: *Advances in Cryptology—CRYPTO 2008: 28th Annual International Cryptology Conference*. Berlin, Heidelberg, Germany: Springer, pp. 203–220. ISBN: 978-3-540-85174-5. DOI: 10.1007/978-3-540-85174-5_12. URL: https://dx.doi.org/10.1007/978-3-540-85174-5_12.

Eronen, Pasi and Hannes Tschofenig (2005). *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS) (RFC4279)*. DOI: 10.17487/rfc4279. URL: https://tools.ietf.org/rfc/rfc4279.txt.

Falconer, Maynard C., Christopher P. Mozak, and Adam J. Norman (2013). *Suppressing power supply noise using data scrambling in double data rate memory systems.* U.S. Patent 8,503,678. URL: `http://www.google.com.ar/patents/US8503678`.

Fan, Zhong, Russell J. Haines, and Parag Kulkarni (2014). "M2M communications for E-health and smart grid: an industry and standard perspective". In: *IEEE Wireless Communications* 21.1, pp. 62–69. DOI: `10.1109/MWC.2014.6757898`. URL: `https://dx.doi.org/10.1109/MWC.2014.6757898`.

Farag, Mohammed M., Lee W. Lerner, and Cameron D. Patterson (2012). "Interacting with Hardware Trojans over a network". In: *IEEE International Symposium on Hardware Oriented Security and Trust—HOST 2012.* IEEE, pp. 69–74. ISBN: 978-1-467-32341-3. DOI: `10.1109/HST.2012.6224323`. URL: `https://dx.doi.org/10.1109/HST.2012.6224323`.

Flajolet, Philippe, Danièle Gardy, and Loÿs Thimonier (1992). "Birthday paradox, coupon collectors, caching algorithms and self-organizing search". In: *Discrete Applied Mathematics* 39.3, pp. 207–229. DOI: `10.1016/0166-218X(92)90177-C`. URL: `https://dx.doi.org/10.1016/0166-218X(92)90177-C`.

Gandolfi, Karine, Christophe Mourtel, and Francis Olivier (2001). "Electromagnetic analysis: Concrete results". In: *3rd International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2001.* Ed. by Çetin K. Koç, David Naccache, and Christof Paar. Berlin, Heidelberg, Germany: Springer, pp. 251–261. ISBN: 978-3-540-44709-2. DOI: `10.1007/3-540-44709-1_21`. URL: `https://dx.doi.org/10.1007/3-540-44709-1_21`.

Genkin, Daniel, Lev Pachmanov, Itamar Pipman, and Eran Tromer (2015). *Stealing Keys from PCs using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation.* IACR Cryptology ePrint Archive, Report 2015/170. URL: `http://eprint.iacr.org/2015/170`.

Genkin, Daniel, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom (2016). *ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels.* IACR Cryptology ePrint Archive, Report 2016/230. URL: `http://eprint.iacr.org/2016/230`.

Genkin, Daniel, Itamar Pipman, and Eran Tromer (2014). "Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs". In: *16th International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2014.* Ed. by Lejla Batina and Matthew Robshaw. Berlin, Heidelberg, Germany: Springer, pp. 242–260. ISBN: 978-3-662-44709-3. DOI: `10.1007/978-3-662-44709-3_14`. URL: `https://dx.doi.org/10.1007/978-3-662-44709-3_14`.

Genkin, Daniel, Adi Shamir, and Eran Tromer (2014). “RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis”. In: *Advances in Cryptology—CRYPTO 2014: 34th Annual International Cryptology Conference*. Ed. by Juan A. Garay and Rosario Gennaro. Berlin, Heidelberg, Germany: Springer, pp. 444–461. ISBN: 978-3-662-44371-2. DOI: `10.1007/978-3-662-44371-2_25`. URL: `https://dx.doi.org/10.1007/978-3-662-44371-2_25`.

Giannuzzi, Lucille A. and Frederick A. Stevie (1999). “A review of focused ion beam milling techniques for TEM specimen preparation”. In: *Micron* 30.3, pp. 197–204. DOI: `10.1016/S0968-4328(99)00005-0`. URL: `https://dx.doi.org/10.1016/S0968-4328(99)00005-0`.

Gianvecchio, Steven and Haining Wang (2007). “Detecting covert timing channels: An entropy-based approach”. In: *14th ACM Conference on Computer and Communications security—CCS ’07*. Alexandria, Virginia, USA: ACM, pp. 307–316. ISBN: 978-1-595-93703-2. DOI: `10.1145/1315245.1315284`. URL: `https://dx.doi.org/10.1145/1315245.1315284` (visited on 11/29/2012).

Giffin, John, Rachel Greenstadt, Peter Litwack, and Richard Tibbetts (2003). “Covert messaging through TCP timestamps”. In: *2nd International Conference on Privacy Enhancing Technologies—PET ’02*. Ed. by Roger Dingledine and Paul Syverson. San Francisco, CA, USA: Springer, pp. 194–208. ISBN: 978-3-540-36467-2. DOI: `10.1007/3-540-36467-6_15`. URL: `https://dx.doi.org/10.1007/3-540-36467-6_15`.

Gilbert, Henri and Helena Handschuh (2004). “Security analysis of SHA-256 and sisters”. In: *10th Annual International Workshop on Selected Areas in Cryptography—SAC 2003*. Ed. by Mitsuru Matsui and Robert J. Zuccherato. Berlin, Heidelberg, Germany: Springer, pp. 175–193. ISBN: 978-3-540-24654-1. DOI: `10.1007/978-3-540-24654-1_13`. URL: `https://dx.doi.org/10.1007/978-3-540-24654-1_13`.

Golomb, Solomon W, Lloyd R Welch, Richard M Goldstein, and Alfred W Hales (1981). *Shift register sequences*. Aegean Park Press, Laguna Hills, CA. ISBN: 978-0-894-12048-0.

Goodspeed, Travis (2008). “A side-channel timing attack of the MSP430 BSL”. In: *Black Hat 2008*. Las Vegas, NV, USA. URL: `https://www.blackhat.com/presentations/bh-usa-08/Goodspeed/BH_US_08_Goodspeed_Side-channel_Timing_Attacks_White_Paper.pdf`.

Goresky, Mark and Andrew M Klapper (2002). “Fibonacci and Galois representations of feedback-with-carry shift registers”. In: *IEEE Transactions on Information Theory* 48.11, pp. 2826–2836. DOI: `10.1109/TIT.2002.804048`. URL: `https://dx.doi.org/10.1109/TIT.2002.804048`.

Gould, Geoffrey A. (2009). *Address scrambling to simplify memory controller's address output multiplexer*. U.S. Patent 7,493,467. URL: `http://www.google.com/patents/US7493467`.

Gruhn, Michael and Tilo Müller (2013). "On the Practicability of Cold Boot Attacks". In: *8th International Conference on Availability, Reliability and Security—ARES 2013*. IEEE Computer Society, pp. 390–397. ISBN: 978-0-769-55008-4. DOI: `10.1109/ARES.2013.52`. URL: `https://dx.doi.org/10.1109/ARES.2013.52`.

Halderman, John Alex, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten (2009). "Lest we remember: cold-boot attacks on encryption keys". In: *Communications of the ACM* 52.5, pp. 91–98. DOI: `10.1145/1506409.1506429`. URL: `https://dx.doi.org/10.1145/1506409.1506429`.

Hamamoto, Takeshi, Soichi Sugiura, and Shizuo Sawada (1998). "On the retention time distribution of dynamic random access memory (DRAM)". In: *IEEE Transactions on Electron Devices* 45.6, pp. 1300–1309. DOI: `10.1109/16.678551`. URL: `https://dx.doi.org/10.1109/16.678551`.

Herbst, Christoph, Elisabeth Oswald, and Stefan Mangard (2006). "An AES Smart Card Implementation Resistant to Power Analysis Attacks". In: *4th International Conference on Applied Cryptography and Network Security—ACNS 2006*. Ed. by Jianying Zhou, Moti Yung, and Feng Bao. Berlin, Heidelberg, Germany: Springer, pp. 239–252. ISBN: 978-3-540-34704-0. DOI: `10.1007/11767480_16`. URL: `https://dx.doi.org/10.1007/11767480_16`.

Hewlett-Packard Inc., Intel Inc., Microsoft Inc., Renesas Electronics Inc., ST-Ericsson, and Texas Instruments Inc. (2013). *USB 3.1 Specification Revision 1.0*. URL: `http://www.usb.org/developers/docs/usb_31_052016.zip`.

Iakymchuk, Taras, Maciej Nikodem, and Krzysztof Kepa (2011). "Temperature-based covert channel in FPGA systems". In: *6th International Workshop on Reconfigurable Communication-centric Systems-on-Chip—ReCoSoC 2011*. IEEE, pp. 1–7. ISBN: 978-1-457-70640-0. DOI: `10.1109/ReCoSoC.2011.5981510`. URL: `https://dx.doi.org/10.1109/ReCoSoC.2011.5981510`.

IEEE Computer Society (2012). *IEEE 802 Part 3-2008: IEEE Standard for Ethernet, section 1*. IEEE Computer Society. URL: `http://standards.ieee.org/getieee802/download/802.3-2012_section1.pdf`.

Intel Inc. (2012a). *Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 2: Instruction Set Reference*.

Intel Inc. (2012b). *Transistors to Transformations – From Sand to Circuits – How Intel Makes Chips*. URL: `http://www.intel.com/content/dam/www/public/us/en/documents/corporate-information/museum-transistors-to-transformations-brochure.pdf`.

Jang, Jae-Won and Swaroop Ghosh (2016). "Performance Impact of Magnetic and Thermal Attack on STTRAM and Low-Overhead Mitigation Techniques". In: *16th ACM/IEEE International Symposium on Low Power Electronics and Design—ISLPED '16*. San Francisco, California, USA: ACM.

Ji, Liping, Wenhao Jiang, Benyang Dai, and Xiamu Niu (2009). "A Novel Covert Channel Based on Length of Messages". In: *International Symposium on Information Engineering and Electronic Commerce—IEEC '09*. IEEE, pp. 551–554. ISBN: 978-0-769-53686-6. DOI: `10.1109/IEEC.2009.122`. URL: `https://dx.doi.org/10.1109/IEEC.2009.122` (visited on 10/08/2012).

Johnson, Don, Alfred J Menezes, and Scott A Vanstone (2001). "The elliptic curve digital signature algorithm (ECDSA)". In: *International Journal of Information Security* 1.1, pp. 36–63. ISSN: 1615-5262. DOI: `10.1007/s102070100002`. URL: `https://dx.doi.org/10.1007/s102070100002`.

Josefsson, S. (2003). *The Base16, Base32, and Base64 Data Encodings (RFC3548)*. DOI: `10.17487/rfc3548`. URL: `https://tools.ietf.org/rfc/rfc3548.txt`.

Jr., Robert J. Jenkins (1996). "ISAAC". In: *3rd International Workshop on Fast Software Encryption—FSE 1996*. Ed. by Dieter Gollmann. Vol. 1039. Lecture Notes in Computer Science. Springer, pp. 41–49. ISBN: 978-3-540-60865-3. DOI: `10.1007/3-540-60865-6_41`. URL: `https://dx.doi.org/10.1007/3-540-60865-6_41`.

Kelsey, John, Bruce Schneier, David Wagner, and Chris Hall (1998). "Side channel cryptanalysis of product ciphers". In: *5th European Symposium on Research in Computer Security—ESORICS '98*. Ed. by Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann. Springer, pp. 97–110. DOI: `10.1007/BFb0055858`. URL: `https://dx.doi.org/10.1007/BFb0055858`.

Kemmerer, Richard A. (1983). "Shared Resource Matrix Methodology: An Approach to Identifying Storage and Timing Channels". In: *ACM Transactions on Computer Systems* 1.3, pp. 256–277. ISSN: 0734-2071. DOI: `10.1145/357369.357374`. URL: `https://dx.doi.org/10.1145/357369.357374`.

Khovratovich, Dmitry, Christian Rechberger, and Alexandra Savelieva (2012). "Bicliques for preimages: attacks on Skein-512 and the SHA-2 family". In: *19th International Workshop on Fast Software Encryption—FSE 2012*. Ed. by Anne Canteaut. Springer, pp. 244–263. ISBN: 978-3-642-34047-5. DOI: `10.1007/978-3-642-34047-5_15`. URL: `https://dx.doi.org/10.1007/978-3-642-34047-5_15`.

Kim, Yoongu, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu (2014). “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”. In: *41st International Symposium on Computer Architecuture—ISCA 2014*. IEEE, pp. 361–372. DOI: `10.1109/ISCA.2014.6853210`. URL: `https://dx.doi.org/10.1109/ISCA.2014.6853210`.

King, Samuel T., Joseph Tucek, Anthony Cozzie, Chris Grier, Weihang Jiang, and Yuanyuan Zhou (2008). “Designing and implementing malicious hardware”. In: *1st USENIX Workshop on Large-scale Exploits and Emergent Threats—LEET ’08*. Berkeley, CA, USA: USENIX Association, 5:1–5:8. URL: `http://dl.acm.org/citation.cfm?id=1387709.1387714`.

Kipling, Rudyard (1897). “Recessional”. In: *The Times*. July 17, 1897.

Koblitz, Neal (1987). “Elliptic curve cryptosystems”. In: *Mathematics of computation* 48.177, pp. 203–209. ISSN: 1088-6842. DOI: `10.1090/S0025-5718-1987-0866109-5`. URL: `https://dx.doi.org/10.1090/S0025-5718-1987-0866109-5`.

Koblitz, Neal and Alfred J Menezes (2015). *A riddle wrapped in an enigma.* IACR Cryptology ePrint Archive, Report 2015/1018. URL: `http://eprint.iacr.org/2015/1018`.

Kocher, Paul C (1995). “Cryptanalysis of Diffie-Hellman, RSA, DSS, and Other Systems Using Timing Attacks (Extended Abstract)”. In: *Advances in Cryptology—CRYPTO ’95: 15th Annual International Cryptology Conference*. Springer, pp. 21–31.

Kocher, Paul C (1996). “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems”. In: *Advances in Cryptology—CRYPTO ’96: 16th Annual International Cryptology Conference*. Ed. by Neal Koblitz. Springer, pp. 104–113. ISBN: 978-3-540-68697-2. DOI: `10.1007/3-540-68697-5_9`. URL: `https://dx.doi.org/10.1007/3-540-68697-5_9`.

Kocher, Paul C, Joshua Jaffe, and Benjamin Jun (1999). “Differential power analysis”. In: *Advances in Cryptology—CRYPTO ’99: 19th Annual International Cryptology Conference*. Ed. by Michael Wiener. Vol. 1666. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, pp. 388–397. ISBN: 978-3-540-48405-9. DOI: `10.1007/3-540-48405-1_25`. URL: `https://dx.doi.org/10.1007/3-540-48405-1_25`.

Kong, Jingfei, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou (2008). “Deconstructing new cache designs for thwarting software cache-based side channel attacks”. In: *2nd ACM Workshop on Computer Security Architectures—CSAW ’08*. New York, NY, USA: ACM, pp. 25–34. DOI: 10.1145/1456508.1456514. URL: https://dx.doi.org/10.1145/1456508.1456514.

Krawczyk, Hugo, Mihir Bellare, and Ran Canetti (1997). *HMAC: Keyed-Hashing for Message Authentication (RFC2104)*. DOI: 10.17487/rfc2104. URL: https://tools.ietf.org/rfc/rfc2104.txt.

Lampson, Butler W. (1973). "A note on the confinement problem". In: *Communications of the ACM* 16.10, pp. 613–615. DOI: 10.1145/362375.362389. URL: https://dx.doi.org/10.1145/362375.362389 (visited on 10/08/2012).

Law, Laurie, Alfred J Menezes, Minghua Qu, Jerry Solinas, and Scott A Vanstone (2003). "An efficient protocol for authenticated key agreement". In: *Designs, Codes and Cryptography* 28.2, pp. 119–134. ISSN: 1573-7586. DOI: 10.1023/A:1022595222606. URL: https://dx.doi.org/10.1023/A:1022595222606.

Lindenlauf, Simon, Hans Höfken, and Marko Schuba (2015). "Cold Boot Attacks on DDR2 and DDR3 SDRAM". In: *10th International Conference on Availability, Reliability and Security—ARES 2015*. IEEE, pp. 287–292. DOI: 10.1109/ARES.2015.28. URL: https://dx.doi.org/10.1109/ARES.2015.28.

Liu, Jamie, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu (2013). "An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms". In: *ACM SIGARCH Computer Architecture News* 41.3, pp. 60–71. DOI: 10.1145/2508148.2485928. URL: https://dx.doi.org/10.1145/2508148.2485928.

Liu, Yali, Dipak Ghosal, Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Stefan Katzenbeisser (2009). "Hide and Seek in Time — Robust Covert Timing Channels". In: *14th European Symposium on Research in Computer Security—ESORICS 2009*. Ed. by Michael Backes and Peng Ning. Vol. 5789. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, pp. 120–135. ISBN: 978-3-642-04443-4. DOI: 10.1007/978-3-642-04444-1. URL: https://dx.doi.org/10.1007/978-3-642-04444-1.

Mangard, Stefan (2004). "Hardware countermeasures against DPA — a statistical analysis of their effectiveness". In: *Topics in Cryptology—CT-RSA 2004: The Cryptographers' Track at the RSA Conference 2004*. Ed. by Tatsuaki Okamoto. Springer, pp. 222–235. ISBN: 978-3-540-24660-2. DOI: 10.1007/978-3-540-24660-2_18. URL: https://dx.doi.org/10.1007/978-3-540-24660-2_18.

Mangard, Stefan, Elisabeth Oswald, and Thomas Popp (2007). *Power analysis attacks: Revealing the secrets of smart cards*. Boston, MA, USA: Springer, pp. 119–165. ISBN: 978-0-387-38162-6. DOI: 10.1007/978-0-387-38162-6_6. URL: https://dx.doi.org/10.1007/978-0-387-38162-6_6.

Massey, James L (1969). “Shift-register synthesis and BCH decoding”. In: *IEEE Transactions on Information Theory* 15.1, pp. 122–127. DOI: 10.1109/TIT.1969.1054260. URL: https://dx.doi.org/10.1109/TIT.1969.1054260.

McGrew, D. and D. Bailey (2012). *AES-CCM Cipher Suites for Transport Layer Security (TLS) (RFC6655).* DOI: 10.17487/rfc6655. URL: https://tools.ietf.org/rfc/rfc6655.txt.

McGrew, D., K. Igoe, and M. Salter (2011). *Fundamental Elliptic Curve Cryptography Algorithms (RFC6090).* DOI: 10.17487/rfc6090. URL: https://tools.ietf.org/rfc/rfc6090.txt.

Menezes, Alfred J, Paul C van Oorschot, and Scott A Vanstone (1996). *Handbook of Applied Cryptography.* CRC Press. ISBN: 978-0-849-38523-0. URL: http://cacr.uwaterloo.ca/hac/.

Messerges, Thomas S (2001). “Securing the AES Finalists Against Power Analysis Attacks”. In: *7th International Workshop on Fast Software Encryption—FSE 2000.* Ed. by Gerhard Goos, Juris Hartmanis, and Bruce van Leeuwen Jan and Schneier. Berlin, Heidelberg, Germany: Springer, pp. 150–164. ISBN: 978-3-540-44706-1. DOI: 10.1007/3-540-44706-7_11. URL: https://dx.doi.org/10.1007/3-540-44706-7_11.

Messerges, Thomas S, Ezzat A Dabbish, and Robert H Sloan (2002). “Examining smart-card security under the threat of power analysis attacks”. In: *IEEE Transactions on Computers* 51.5, pp. 541–552. DOI: 10.1109/TC.2002.1004593. URL: https://dx.doi.org/10.1109/TC.2002.1004593.

Metzger, P. and W. Simpson (1995). *IP Authentication using Keyed MD5 (RFC1828).* DOI: 10.17487/rfc1828. URL: https://tools.ietf.org/rfc/rfc1828.txt.

Micron Technology Inc. (2008). *TN-04-56: Dealing with DDR2/DDR3 Clock Jitter Introduction.* URL: https://www.micron.com/~/media/documents/products/technical-note/dram/tn0456_clock_jitter.pdf.

Micron Technology Inc. (2014). *DDR3 SDRAM Datasheet for MT41J256M4, MT41J128M4 and MT41J64M4.* URL: https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/1gb_ddr3_sdram.pdf.

Miller, Charlie and Chris Valasek (2015). *Remote exploitation of an unaltered passenger vehicle.* URL: http://illmatics.com/Remote%5C%20Car%5C%20Hacking.pdf.

Miller, John Milton (1919). “Dependence of the input impedance of a three-electrode vacuum tube upon the load in the plate circuit”. In: vol. 351. Scientific papers of the Bureau of Standards. United States Government Printing Office, pp. 367–385. DOI:

10.1016/S0016-0032(19)90474-1. URL: https://dx.doi.org/10.1016/S0016-0032(19)90474-1.

Miller, Victor (1986). "Use of elliptic curves in cryptography". In: *Advances in Cryptology—CRYPTO '85: 5th Annual International Cryptology Conference.* Berlin, Heidelberg, Germany: Springer, pp. 417–426. ISBN: 978-3-540-39799-1. DOI: 10.1007/3-540-39799-X_31. URL: https://dx.doi.org/10.1007/3-540-39799-X_31.

Modadugu, Nagendra and Eric Rescorla (2004). "The Design and Implementation of Datagram TLS". In: *Network and Distributed System Security Symposium—NDSS 2004.* The Internet Society. ISBN: 978-1-891-56218-1.

Möller, Bodo, Thai Duong, and Krzysztof Kotowicz (2014). *This POODLE bites: exploiting the SSL 3.0 fallback.* Google Inc. URL: https://www.openssl.org/~bodo/ssl-poodle.pdf.

Moore, Simon W, Ross Anderson, Paul Cunningham, Robert Mullins, and George Taylor (2002). "Improving smart card security using self-timed circuits". In: *8th International Symposium on Asynchronous Circuits and Systems—ASYNC 2002.* IEEE, pp. 211–218. DOI: 10.1109/ASYNC.2002.1000311. URL: https://dx.doi.org/10.1109/ASYNC.2002.1000311.

Moskowitz, Ira S. and Myong H. Kang (1994). "Covert channels–here to stay?" In: *9th Annual Confererence on Computer Assurance—COMPASS '94*, pp. 235–243. DOI: 10.1109/CMPASS.1994.318449. URL: https://dx.doi.org/10.1109/CMPASS.1994.318449 (visited on 09/21/2012).

Motorola, Inc. (2003). *SPI Block Guide V03.06.* URL: http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf.

Mozak, Christopher P. (2011). *Suppressing power supply noise using data scrambling in double data rate memory systems.* U.S. Patent 7,945,050. URL: https://www.google.com.ar/patents/US7945050.

Müller, Tilo and Michael Spreitzenbarth (2013). "FROST: Forensic Recovery of Scrambled Telephones". In: *11th International Conference on Applied Cryptography and Network Security—ACNS 2013.* Ed. by Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini. Springer, pp. 373–388. ISBN: 978-3-642-38980-1. DOI: 10.1007/978-3-642-38980-1_23. URL: https://dx.doi.org/10.1007/978-3-642-38980-1_23.

Murdoch, Steven J (2007). *Covert channel vulnerabilities in anonymity systems.* Tech. rep. UCAM-CL-TR-706. University of Cambridge, Computer Laboratory. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-706.pdf.

Murdoch, Steven J and Stephen Lewis (2005). “Embedding covert channels into TCP/IP”. In: *7th International Workshop on Information Hiding—IH 2005*. Ed. by Mauro Barni, Jordi Herrera-Joancomartí, Stefan Katzenbeisser, and Fernando Pérez-González. Berlin, Heidelberg, Germany: Springer, pp. 247–261. ISBN: 978-3-540-31481-3. DOI: 10.1007/11558859_19. URL: https://dx.doi.org/10.1007/11558859_19.

National Institute of Standards and Technology (1999). *Recommended Elliptic Curves for Federal Government Use*. URL: http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf.

National Institute of Standards and Technology (2001). *FIPS Publication 197: Announcing the Advanced Encryption Standard (AES)*. URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

National Institute of Standards and Technology (2002). *FIPS 180-2, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-2*. URL: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

Needham, Roger M and David J Wheeler (1997). *TEA extensions*. Tech. rep. Cambridge University, Cambridge, UK. URL: http://www.cix.co.uk/~klockstone/xtea.pdf.

Von Neumann, John (1951). “Various Techniques Used in Connection with Random Digits”. In: *Journal of Research of the National Bureau of Standards Applied Math Series* 12, pp. 36–38. URL: https://dornsifecms.usc.edu/assets/sites/520/docs/VonNeumann-ams12p36-38.pdf.

NXP Semiconductors N.V. (2011). *2N7002 product datasheet*. URL: http://www.nxp.com/documents/data_sheet/2N7002.pdf.

NXP Semiconductors N.V. (2014a). *MFRC522 Standard 3V MIFARE reader solution (112138) Rev 3.8*. URL: http://www.nxp.com/documents/data_sheet/MFRC522.pdf.

NXP Semiconductors N.V. (2014b). *UM10204* I^{2}C*-bus specification and user manual*. URL: http://www.nxp.com/documents/user_manual/UM10204.pdf.

O’Flynn, Colin and Zhizhang (David) Chen (2014). “ChipWhisperer: An Open-Source Platform for Hardware Embedded Security Research”. English. In: *Constructive Side-Channel Analysis and Secure Design—COSADE 2014*. Ed. by Emmanuel Prouff. Vol. 8622. Lecture Notes in Computer Science. Springer, pp. 243–260. ISBN: 978-3-319-10174-3. DOI: 10.1007/978-3-319-10175-0_17. URL: https://dx.doi.org/10.1007/978-3-319-10175-0_17.

O’Flynn, Colin and Zhizhang (David) Chen (2015). “Side channel power analysis of an AES-256 bootloader”. In: *28th Canadian Conference on Electrical and Computer Engineering—CCECE 2015*. IEEE, pp. 750–755. DOI: 10.1109/CCECE.2015.7129369. URL: https://dx.doi.org/10.1109/CCECE.2015.7129369.

Open Mobile Alliance (2016). *Lightweight Machine to Machine Technical Specification Candidate Version 1.0.* URL: `http://www.openmobilealliance.org`.

Osvik, Dag Arne, Adi Shamir, and Eran Tromer (2006). "Cache attacks and countermeasures: the case of AES". In: *Topics in Cryptology—CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006.* Ed. by David Pointcheval. Springer, pp. 1–20. ISBN: 978-3-540-32648-9. DOI: `10.1007/11605805_1`. URL: `https://dx.doi.org/10.1007/11605805_1`.

Oswald, David and Christof Paar (2011). "Breaking Mifare DESFire MF3ICD40: power analysis and templates in the real world". In: *13th International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2011.* Ed. by Bart Preneel and Tsuyoshi Takagi. Springer, pp. 207–222. ISBN: 978-3-642-23951-9. DOI: `10.1007/978-3-642-23951-9_14`. URL: `https://dx.doi.org/10.1007/978-3-642-23951-9_14`.

Oswald, Elisabeth, Stefan Mangard, Christoph Herbst, and Stefan Tillich (2006). "Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers". In: *Topics in Cryptology—CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006.* Ed. by David Pointcheval. Berlin, Heidelberg, Germany: Springer, pp. 192–207. ISBN: 978-3-540-32648-9. DOI: `10.1007/11605805_13`. URL: `https://dx.doi.org/10.1007/11605805_13`.

Page, Dan (2002). *Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel.* IACR Cryptology ePrint Archive, Report 2002/169. URL: `https://eprint.iacr.org/2002/169`.

Philips Semiconductors (1997). I^2S *bus specification.* URL: `https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf`.

Preneel, Bart and Paul C van Oorschot (1995). "MDx-MAC and building fast MACs from hash functions". In: *Advances in Cryptology—CRYPTO '95: 15th Annual International Cryptology Conference.* Ed. by Don Coppersmith. Springer, pp. 1–14. ISBN: 978-3-540-44750-4. DOI: `10.1007/3-540-44750-4_1`. URL: `https://dx.doi.org/10.1007/3-540-44750-4_1`.

Preneel, Bart and Paul C van Oorschot (1996). "On the security of two MAC algorithms". In: *Advances in Cryptology—EUROCRYPT '96: International Conference on the Theory and Application of Cryptographic Techniques.* Ed. by Ueli Maurer. Springer, pp. 19–32. ISBN: 978-3-540-68339-1. DOI: `10.1007/3-540-68339-9_3`. URL: `https://dx.doi.org/10.1007/3-540-68339-9_3`.

Quisquater, Jean-Jacques and David Samyde (2001). "Electromagnetic analysis (EMA): Measures and counter-measures for smart cards". In: *International Conference on Research in Smart Cards—E-smart 2001.* Ed. by Isabelle Attali and Thomas Jensen. Berlin, Heidelberg, Germany: Springer, pp. 200–210. ISBN: 978-3-540-45418-2. DOI:

10.1007/3-540-45418-7_17. URL: https://dx.doi.org/10.1007/3-540-45418-7_17.

Rahmati, Amir, Mastooreh Salajegheh, Dan Holcomb, Jacob Sorber, Wayne P. Burleson, and Kevin Fu (2012). “TARDIS: Time and Remanence Decay in SRAM to Implement Secure Protocols on Embedded Devices without Clocks”. In: *21st USENIX Security Symposium—USENIX Security 12*. Bellevue, WA: USENIX, pp. 221–236. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rahmati.

Ravi, Srivaths, Anand Raghunathan, and Srimat Chakradhar (2004). “Tamper resistance mechanisms for secure embedded systems”. In: *17th International Conference on VLSI Design—ICVD 2004*. IEEE, pp. 605–611. DOI: 10.1109/ICVD.2004.1260985. URL: https://dx.doi.org/10.1109/ICVD.2004.1260985.

Raza, Shahid, Hossein Shafagh, Kasun Hewage, René Hummen, and Thiemo Voigt (2013). “Lithe: Lightweight Secure CoAP for the Internet of Things”. In: *Sensors Journal, IEEE* 13.10, pp. 3711–3720. DOI: 10.1109/JSEN.2013.2277656. URL: https://dx.doi.org/10.1109/JSEN.2013.2277656.

Raza, Shahid, Daniele Trabalza, and Thiemo Voigt (2012). “6LoWPAN compressed DTLS for CoAP”. In: *8th International Conference on Distributed Computing in Sensor Systems—DCOSS 2012*. IEEE, pp. 287–289. DOI: 10.1109/DCOSS.2012.55. URL: https://dx.doi.org/10.1109/DCOSS.2012.55.

Rescorla, Eric (2008). *TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM) (RFC5289)*. DOI: 10.17487/rfc5289. URL: https://tools.ietf.org/rfc/rfc5289.txt.

Rescorla, Eric and Nagendra Modadugu (2006). *Datagram Transport Layer Security (RFC4347)*. DOI: 10.17487/rfc4347. URL: https://tools.ietf.org/rfc/rfc4347.txt.

Reshadi, Mehrdad and Nikil Dutt (2005). “Generic Pipelined Processor Modeling and High Performance Cycle-Accurate Simulator Generation”. In: *Conference on Design, Automation and Test in Europe—DATE ’05*. Vol. 2. Washington, DC, USA: IEEE, pp. 786–791. ISBN: 978-0-769-52288-3. DOI: 10.1109/DATE.2005.166. URL: https://dx.doi.org/10.1109/DATE.2005.166.

Rivest, Ronald L, Adi Shamir, and Len Adleman (1978). “A method for obtaining digital signatures and public-key cryptosystems”. In: *Communications of the ACM* 21.2, pp. 120–126. DOI: 10.1145/359340.359342. URL: https://dx.doi.org/10.1145/359340.359342.

Rivest, Ronald L, Adi Shamir, and Len Adleman (1983). *Cryptographic communications system and method.* U.S. Patent 4,405,829. URL: `http://www.google.com/patents/US4405829`.

Rosenfeld, Paul, Elliott Cooper-Balis, and Bruce Jacob (2011). "DRAMSim2: A Cycle Accurate Memory System Simulator". In: *IEEE Computer Architecture Letters* 10.1, pp. 16–19. ISSN: 1556-6056. DOI: `10.1109/L-CA.2011.4`. URL: `https://dx.doi.org/10.1109/L-CA.2011.4`.

Rowland, Craig H. (1997). "Covert Channels in the TCP/IP Protocol Suite". In: *First Monday* 2.5. DOI: `10.5210/fm.v2i5.528`. URL: `https://dx.doi.org/10.5210/fm.v2i5.528`.

Sadasivan, Shyam (2006). *White paper: An Introduction to the ARM Cortex-M3 Processor.* ARM Ltd.

Schoof, René (1985). "Elliptic curves over finite fields and the computation of square roots mod p". In: *Mathematics of Computation* 44.170, pp. 483–494. DOI: `10.2307/2007968`. URL: `https://dx.doi.org/10.2307/2007968`.

Schoof, René (1995). "Counting points on elliptic curves over finite fields". In: *Journal de théorie des nombres de Bordeaux* 7.1, pp. 219–254. DOI: `10.5802/jtnb.142`. URL: `https://dx.doi.org/10.5802/jtnb.142`.

Schuhmacher, Frank (2014). *DPA contest v4 – Hall of Fame (AES-256 RSM Implementation).* URL: `http://www.dpacontest.org/v4/rsm_hall_of_fame.php`.

Shah, Gaurav and Matt Blaze (2009). "Covert channels through external interference". In: *3rd USENIX Workshop on Offensive Technologies—WOOT '09.* Berkeley, CA, USA: USENIX Association, pp. 3–3. URL: `http://dl.acm.org/citation.cfm?id=1855876.1855879`.

Shockley, William (1952). "A unipolar 'field-effect' transistor". In: *Proceedings of the IRE* 40.11, pp. 1365–1376. DOI: `10.1109/JRPROC.1952.273964`. URL: `https://dx.doi.org/10.1109/JRPROC.1952.273964`.

Silverman, Joseph H. (1986). *The Arithmetic of Elliptic Curves.* Vol. 106. Graduate Texts in Mathematics. New York, NY, USA: Springer. ISBN: 978-1-475-71922-2. DOI: `10.1007/978-1-4757-1920-8`. URL: `https://dx.doi.org/10.1007/978-1-4757-1920-8`.

Skorobogatov, Sergei Petrovich (2004). "Semi-invasive attacks: a new approach to hardware security analysis". PhD thesis. University of Cambridge, Darwin College. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf`.

STMicroelectronics N.V. (2011a). *PM0081: STM32F40xxx and STM32F41xxx Flash Programming Manual.* URL: `http://www.bdtic.com/DownLoad/ST/PM0081.pdf`.

STMicroelectronics N.V. (2011b). *RM0090: Reference manual STM32F405xx, STM32F407xx, STM32F415xx and STM32F417xx advanced ARM-based 32-bit MCUs.* URL: `http://www.st.com/resource/en/reference_manual/DM00031020.pdf`.

STMicroelectronics N.V. (2012). *DM37051: ARM Cortex-M4 STM32F405xx STM32F407xx datasheet.* URL: `http://www.st.com/resource/en/datasheet/stm32f407vg.pdf`.

STMicroelectronics N.V. (2015a). *DM88500: STM32F030x4 STM32F030x6 STM32F030x8 STM32F030xC datasheet.* URL: `http://www.st.com/resource/en/datasheet/stm32f030c8.pdf`.

STMicroelectronics N.V. (2015b). *Releasing your creativity: STM32F0 series Mainstream 32-bit MCUs.* URL: `http://www.st.com/resource/en/brochure/brstm32f0.pdf`.

STMicroelectronics N.V. (2015c). *RM0360 Reference manual STM32F030x4/6/8/C and STM32F070x6/B advanced ARM-based 32-bit MCUs.* URL: `http://www.st.com/resource/en/datasheet/stm32f030f4.pdf`.

Stüttgen, Johannes (2015). "On the Viability of Memory Forensics in Compromised Environments". PhD thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Department of Computer Science 1, p. 134. URN: `urn:nbn:de:bvb:29-opus4-63160`. URL: `https://nbn-resolving.org/urn:nbn:de:bvb:29-opus4-63160`.

Stüttgen, Johannes and Michael Cohen (2013). "Anti-forensic resilient memory acquisition". In: *Digital Investigation* 10, S105–S115. DOI: `10.1016/j.diin.2013.06.012`. URL: `https://dx.doi.org/10.1016/j.diin.2013.06.012`.

Stüttgen, Johannes, Stefan Vömel, and Michael Denzel (2015). "Acquisition and analysis of compromised firmware using memory forensics". In: *Digital Investigation* 12.Supplement 1, S50–S60. DOI: `10.1016/j.diin.2015.01.010`. URL: `https://dx.doi.org/10.1016/j.diin.2015.01.010`.

Tate, John T (1974). "The arithmetic of elliptic curves". In: *Inventiones mathematicae* 23.3, pp. 179–206. ISSN: 1432-1297. DOI: `10.1007/BF01389745`. URL: `https://dx.doi.org/10.1007/BF01389745`.

Tehranipoor, Mohammad and Farinaz Koushanfar (2010). "A Survey of Hardware Trojan Taxonomy and Detection". In: *IEEE Design & Test of Computers* 27.1, pp. 10–25. DOI: `10.1109/MDT.2010.7`. URL: `https://dx.doi.org/10.1109/MDT.2010.7`.

Tillich, Stefan and Johann Großschädl (2007). "Power Analysis Resistant AES Implementation with Instruction Set Extensions". English. In: *9th International Workshop on Cryptographic Hardware and Embedded Systems—CHES 2007.* Ed. by Pascal Paillier and Ingrid Verbauwhede. Vol. 4727. Lecture Notes in Computer Science. Berlin, Heidelberg, Germany: Springer, pp. 303–319. ISBN: 978-3-540-74734-5. DOI: 10.1007/978-3-540-74735-2_21. URL: https://dx.doi.org/10.1007/978-3-540-74735-2_21.

Tsudik, Gene (1992). "Message authentication with one-way hash functions". In: *ACM SIGCOMM Computer Communication Review* 22.5, pp. 29–38. DOI: 10.1145/141809.141812. URL: https://dx.doi.org/10.1145/141809.141812.

Tsunoo, Yukiyasu, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi (2003). "Cryptanalysis of DES implemented on computers with cache". In: *5th International Workshop of Cryptographic Hardware and Embedded Systems—CHES 2003*. Ed. by Colin D. Walter, Çetin K. Koç, and Christof Paar. Springer, pp. 62–76. ISBN: 978-3-540-45238-6. DOI: 10.1007/978-3-540-45238-6_6. URL: https://dx.doi.org/10.1007/978-3-540-45238-6_6.

Tunstall, Michael, Carolyn Whitnall, and Elisabeth Oswald (2014). "Masking Tables—An Underestimated Security Risk". In: *20th International Workshop on Fast Software Encryption—FSE 2013*. Ed. by Shiho Moriai. Berlin, Heidelberg, Germany: Springer, pp. 425–444. ISBN: 978-3-662-43933-3. DOI: 10.1007/978-3-662-43933-3_22. URL: https://dx.doi.org/10.1007/978-3-662-43933-3_22.

Ukil, Arijit, Soma Bandyopadhyay, Abhijan Bhattacharyya, and Arpan Pal (2013). "Lightweight Security Scheme for Vehicle Tracking System Using CoAP". In: *International Workshop on Adaptive Security—ASPI '13*. Zurich, Switzerland: ACM, 3:1–3:8. ISBN: 978-1-450-32543-1. DOI: 10.1145/2523501.2523504. URL: https://dx.doi.org/10.1145/2523501.2523504.

Ukil, Arijit, Soma Bandyopadhyay, Abhijan Bhattacharyya, Arpan Pal, and Tulika Bose (2014). "Auth-Lite: Lightweight M2M Authentication reinforcing DTLS for CoAP". In: *International Conference on Pervasive Computing and Communications Workshops—PERCOM Workshops 2014*. IEEE, pp. 215–219. DOI: 10.1109/PerComW.2014.6815204. URL: https://dx.doi.org/10.1109/PerComW.2014.6815204.

Van Herrewege, Anthony, Vincent van der Leest, André Schaller, Stefan Katzenbeisser, and Ingrid Verbauwhede (2013). "Secure PRNG seeding on commercial off-the-shelf microcontrollers". In: *3rd International Workshop on Trustworthy Embedded Devices—TrustED '13*. New York, NY, USA: ACM, pp. 55–64. ISBN: 978-1-450-32486-1. DOI: 10.1145/2517300.2517306. URL: https://dx.doi.org/10.1145/2517300.2517306.

Van Herrewege, Anthony and Ingrid Verbauwhede (2014). "Software Only, Extremely Compact, Keccak-based Secure PRNG on ARM Cortex-M". In: *51st Annual Design Automation Conference—DAC '14*. San Francisco, CA, USA: ACM, 111:1–111:6. ISBN: 978-1-450-32730-5. DOI: 10.1145/2593069.2593218. URL: https://dx.doi.org/10.1145/2593069.2593218.

Vömel, Stefan and Felix C Freiling (2011). "A Survey of Main Memory Acquisition and Analysis Techniques for the Windows Operating System". In: *Digital Investigation* 8.1, pp. 3–22. ISSN: 1742-2876. DOI: `10.1016/j.diin.2011.06.002`. URL: `https://dx.doi.org/10.1016/j.diin.2011.06.002`.

Vömel, Stefan and Felix C Freiling (2012). "Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition". In: *Digital Investigation* 9.2, pp. 125–137. ISSN: 17422876. DOI: `10.1016/j.diin.2012.04.005`. URL: `https://dx.doi.org/10.1016/j.diin.2012.04.005`.

Vömel, Stefan and Johannes Stüttgen (2013). "An Evaluation Platform for Forensic Memory Acquisition Software". In: *Digital Investigation* 10, S30–S40. DOI: `10.1016/j.diin.2013.06.004`. URL: `https://dx.doi.org/10.1016/j.diin.2013.06.004`.

Wang, Zhenghong and Ruby B Lee (2007). "New cache designs for thwarting software cache-based side channel attacks". In: *ACM SIGARCH Computer Architecture News* 35.2, pp. 494–505. DOI: `10.1145/1273440.1250723`. URL: `https://dx.doi.org/10.1145/1273440.1250723`.

Wendzel, Steffen and Jörg Keller (2012). "Systematic Engineering of Control Protocols for Covert Channels". In: *13th International Conference on Communications and Multimedia Security—CMS 2012*. Ed. by Bart De Decker and David W. Chadwick. Berlin, Heidelberg, Germany: Springer, pp. 131–144. ISBN: 978-3-642-32805-3. DOI: `10.1007/978-3-642-32805-3_11`. URL: `https://dx.doi.org/10.1007/978-3-642-32805-3_11`.

Widmer, Albert X. and Peter A. Franaszek (1983). "A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code." In: *IBM Journal of Research and Development* 27.5, pp. 440–451. DOI: `10.1147/rd.275.0440`. URL: `https://dx.doi.org/10.1147/rd.275.0440`.

Wyns, Philippe and Richard L Anderson (1989). "Low-temperature operation of silicon dynamic random-access memories". In: *IEEE Transactions on Electron Devices* 36.8, pp. 1423–1428. DOI: `10.1109/16.30954`. URL: `https://dx.doi.org/10.1109/16.30954`.

Ye, Wu, Narayanan Vijaykrishnan, Mahmut Kandemir, and Mary Jane Irwin (2000). "The Design and Use of Simplepower: A Cycle-accurate Energy Estimation Tool". In: *37th Annual Design Automation Conference—DAC '00*. Los Angeles, California, USA: ACM, pp. 340–345. ISBN: 978-1-581-13187-1. DOI: `10.1145/337292.337436`. URL: `https://dx.doi.org/10.1145/337292.337436`.

Yiu, Joseph (2009). *The Definitive Guide to the ARM Cortex-M3*. 2nd. Newton, MA, USA: Newnes. ISBN: 978-1-856-17963-8.

Yourst, Matt T. (2007). “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator”. In: *IEEE International Symposium on Performance Analysis of Systems Software—ISPASS 2007*, pp. 23–34. DOI: `10.1109/ISPASS.2007.363733`. URL: `https://dx.doi.org/10.1109/ISPASS.2007.363733`.

Zander, Sebastian, Grenville Armitage, and Philip Branch (2007). “A survey of covert channels and countermeasures in computer network protocols”. In: *IEEE Communications Surveys & Tutorials* 9.3, pp. 44–57. DOI: `10.1109/COMST.2007.4317620`. URL: `https://dx.doi.org/10.1109/COMST.2007.4317620` (visited on 10/08/2012).

Zandwijk, Jan Peter van (2015). “A mathematical approach to NAND flash-memory descrambling and decoding”. In: *Digital Investigation* 12, pp. 41–52. DOI: `10.1016/j.diin.2015.01.003`. URL: `https://dx.doi.org/10.1016/j.diin.2015.01.003`.