CYBERNETICA
Information Security Research Institute

An Overview of Vulnerabilities and Mitigations of Intel SGX Applications

Version 1.0

Jaak Randmets

D-2-116 / 2021

Mailing address:
Cybernetica AS
Mäealuse 2/1
12618 Tallinn
Estonia

An Overview of Vulnerabilities and Mitigations of Intel SGX Applications

Jaak Randmets

Version 1.0

Abstract

This research report covers technical aspects of developing security critical applications using Intel Software Guard Extensions. We give a high-level overview of avenues that can be used to attack applications that use Software Guard Extensions as a privacy enhancing technology and cover some of the techniques to make these attack vectors more challenging to exploit.

Contents

1 Introduction

Intel Software Guard Extensions (SGX) is an extension of the instruction set of Intel processors which enables developing secure applications when even the host operating system is not trusted. SGX relies on three concepts to protect data: enclaves, attestation and data sealing.

SGX is a set of CPU instructions for creating and operating with memory partitions called *enclaves*. When an application creates an enclave, it provides a protected memory area with confidentiality and integrity guarantees. These guarantees hold even if privileged malware is present in the system, meaning that the enclave is protected even from the operating system that is running the enclave. With enclaves, it is possible to significantly reduce the attack surface of an application.

Remote attestation is used to prove to an external party that the expected enclave was created on a remote machine. During remote attestation, the enclave generates a report that can be remotely verified with the help of the Intel Attestation Service. Using remote attestation, an application can verify that a server is running trusted software before private information is uploaded.

Data sealing allows enclaves to store data outside of the enclave without compromising confidentiality and integrity of the data. The sealing is achieved by encrypting the data before it exits the enclave. The encryption key is derived in a way that only the specific enclave on that platform can later decrypt it.

SGX can be used to greatly enhance security of applications but it is important to highlight that organizational and human aspects of security are nearly always more important than technical aspects. Information may leak due to a human error without any malicious parties involved. For example, due to coding mistakes, lack of knowledge, or high-level decisions to not commit sufficient resources to security. Furthermore, no mitigation is helpful when the platform (SGX SDK and firmware) in not kept up to date. SGX requires some rather advanced techniques to be attacked successfully but automated tools that exploit such vulnerabilities (such as transient execution) are already out there in the wild [Cim20]. In SGX many of such vulnerabilities have been mitigated by microcode updates.

In Section 1.1 we give a short overview of Intel SGX technology. The overview is rather brief and the reader unfamiliar with the technology is welcome to consult introductory materials such as the Intel SGX developer guide [Int21]. In Section 1.2 we classify protected resources into categories: cryptographic keys, user data, and statistics/metadata. Each type of resource could be protected with different mitigations. In Chapter 2 we go over the following attack vectors that can be used to target applications protected by Intel SGX: cryptographic primitives and protocols (Section 2.1), enclave surface (Section 2.2), side channels (Section 2.3), speculative execution (Section 2.4), and output inference (Section 2.5). For each attack vector we cover some proof of concept attacks and give some guidelines for mitigations.

1.1 Protection mechanisms of Intel SGX

When talking about protections mechanisms provided by SGX the overarching assumption is that Intel is a trusted third party. Currently we have to assume that the hardware has no back doors and that Intel is an honest and uncompromised participant during remote attestation.

The internals of Intel CPU dies are nearly completely opaque and best guesses on how they internally function are via official Intel documentation and patent applications. Luckily, great strides have been recently made by security researchers by revealing security weaknesses and flaws in CPU design. While discovered flaws affect Intel the most (due to promised guarantees of SGX) they are not limited to Intel. Some of the timing side channels are inherent to (modern) microcontrollers and thus affect all modern CPUs (see *Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic* [BPS18]).

But, unfortunately, without fully transparent view and access to documentation of internals it is impossible to prove lack of security vulnerabilities. Even with full transparency building a formal specification, model and proving security of such complex architecture would be a monumental undertaking.

1.1.1 Privacy, integrity and isolation of data

SGX enclaves are isolated memory regions of code and data residing in main memory (RAM). Privacy, integrity and isolation of data between enclaves is achieved via authenticated encryption with the help of trusted hardware on the CPU-die [Gue16]. Unencrypted plain-text values only reside on the CPU-die in registers and various caches.

Enclave may store values in RAM region that is protected by encryption and any attempts to modify the values will be detected when the tampered region is read. When integrity (encrypted value is modified) or freshness (encrypted value is replayed) violation is detected the system will hang and will need to be re-booted (so-called *drop-and-lock* policy). An outside observer is not able to distinguish if encrypted values in the enclave memory refer to equal plaintext values or not. Additionally, an outside observer will not be able to tell if subsequent writes to the same (or different) memory address store the same plaintext values or not. While SGX protects the contents of memory cryptographically it does not hide the locations where memory stores and loads refer to. Many attacks exploit exactly that limitation in combination with transient execution behaviour of modern processors.

1.1.2 Remote attestation and integrity of code

The Intel SGX features *remote attestation* [AGJS13, JSR+16, KSC+18] allowing for clients to remotely establish trust of secure enclaves. Remote attestation is a cryptographic protocol that in the end establishes a secure communication channel between a secure enclave and a client. While Intel SGX offers more flexibility in this work we assume that remote attestation involves following three parties:

- a *server* hosting a secure application with its enclave,
- a *client* that is mistrustful of the server but wants to establish trust of the enclave, and
- the *Intel Attestation Service* (IAS) that verifies the enclave and provides the client the assurance of enclaves' authenticity.

If the client trusts SGX technology, and trusts Intel to not collude with the server (and not behave maliciously in other ways) then the client can assume that remote attestation establishes the following:

- Identity (cryptographic hash of code and data) of the enclaved application.
- That enclave has not been tampered with.
- Version of the hardware platform.
- The security patch level of hardware.
- Version of Intel SGX SDK (software library provided by Intel for developing SGX applications).

Further details of remote attestation scheme is beyond the scope of this work.

1.1.3 Attack surface minimization

Major protection mechanism of Intel SGX is attack surface minimization[1].

When a remote attacker is targeting a security critical application in classically hosted (personal server, cloud) environment often the initial point of entry is not the security application itself but some other application running (in user mode) on the same host. The other application acts as a gateway to deploy an exploit against operating system that, in turn, compromises all other applications on that host. This means that the attack surface may encompass many millions of lines of code. Such a large code base is infeasible to protect or audit.

SGX enclaves run in user mode and are protected from a potentially hostile operating system. By eliminating the direct threat of the operating system we reduce the attack surface by an order of magnitude against remote attackers and offer protection against attackers with privileged and even physical access.

Applications that use SGX are divided into two parts. A trusted component (a set of enclaves) and an untrusted component. The untrusted component is regular application code that interacts with enclave(s) in some way. From the enclave standpoint both the operating system and the rest of the application are to be viewed as untrusted.

To some degree most SGX enclaves need to communicate with the operating system (to manage files, to perform I/O, etc.) or non-enclaved part of the security application.

[1] `https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-1-foundation`

This interaction between barriers of trust can not be achieved via regular function calls. Instead, there are two special calls: ECALLs and OCALLs. ECALL is a (trusted) function call that enters an SGX enclave and OCALL is an (untrusted) function call that leaves an enclave. ECALLs are referred to as trusted calls in the sense that they invoke trusted (enclaved) code, and similarly OCALLs are referred to as untrusted calls in the sense that they execute code that the host of the server is able modify without the client being made aware of it. Other function calls made in enclave can not leave that same enclave. Using Intel SGX SDK the set of allowed OCALLs and ECALLs for each enclave is strictly specified using *enclave definition language* (or EDL) and this set of calls defines the direct attack surface of the enclave.

1.2 Protected resources

We consider three types of resources in the order of sensitivity: cryptographic keys, user data, and statistics/metadata.

1.2.1 Cryptographic keys

In order to protect user data enclaved applications have to generally manage some sort of cryptographic key material. For example, during remote attestation client and enclave could establish a symmetric session key that is used to encrypt further communication between the client and the enclave. If this session key is compromised the trusted channel is broken and any participant with appropriate access can play either the role of the client or the enclave or simply passively collect all client data. Side-channel vulnerabilities are particularly dangerous against algorithms that handle keys as even partially leaked information about a key can lead to a total compromise.

Cryptographic keys are the most sensitive resource to protect. When keys are compromised it is likely that user data can be decrypted and, in some extreme cases, all security guarantees of SGX enclave can be broken (see ZombieLoad [SLM+19]) if SGX internal keys are compromised. Using compromised cryptographic keys an attacker can forge messages and manipulate data in undetectable ways. Hence, most aggressive mitigations should be applied to protecting cryptographic keys and procedures should be developed to lessen the impact of compromised keys.

1.2.2 User data

The primary goal of SGX is to keep user data secure by protecting both privacy and integrity at runtime. Hence, as the most basic security consideration user data must be kept encrypted at rest using industry standard cryptographic primitives.

SGX automatically provides strong in-memory encryption via a specialized on-chip hardware *memory encryption engine* (see A Memory Encryption Engine Suitable for General Purpose Processors [Gue16] by Shay Gueron) and a convenient API for working with encrypted files. Both of these mechanisms also protect data against tampering (integrity protection). The memory encryption engine furthermore also offers data replay protection.

Many applications also use publicly available data while processing private information. When this public data is tampered it affects results of (private) data processing. For this reason the integrity of publicly available data should be protected. Thanks to its low cost overhead of SGX, it is often easy for public data to be protected by the same encryption and integrity protection mechanisms as private data.

Data processing algorithms are frequently difficult to protect against side-channel attacks (see Section 2.3). Attack mitigations can reduce performance significantly. For this reasons it is often not feasible to handle processing of public and private data uniformly. For example, the same functionality could be implemented in two distinct ways: one implementation to handle private data in a side-channel protected manner, and the second implementation to handle public data in a more performant (but non-side-channel-hardened) manner.

In conventional languages without a *security type system* [SM03] it is difficult to cleanly segregate data with different security levels and, thus, it is easy to introduce accidental data leakage when handling public and private data non-uniformly (e.g. calling algorithm that is not side-channel hardened on private data). When the language offers sufficiently powerful type system the information flow security can be encoded at the library level (for example [PVH19, BVR15]). Great care must be taken when handling data of different security levels.

1.2.3 Statistics/metadata

The result of processing private user data is usually some statistics intended for some participant. For example, distribution of salary information including minimum, maximum, average and quantiles. Statistical information is often provided for many subsets of records (for each age group, for each gender, etc). Output statistics can be very sensitive information and from a protection standpoint should be handled as carefully as regular data. Only the intended participants should have accesses to statistical results.

Side channels (see Section 2.3) usually do not directly leak input records or output statistics but rather they indirectly reveal some (statistical) information such as the number of records, distribution of record values, or when and how often a particular data is accessed. Depending of the structure of processed data this may also include sizes of individual records. We call this indirectly revealed statistical information *inferable* as a malicious party does not directly learn it. Instead, a malicious party must make some effort to be able to infer it via side channel analysis.

Inferable information is not immediately public to everyone. The amount of information leakage depends on the involved participants, the amount of risk potential attackers are willing to take, and resources they can afford to spend. For example:

- Honest (but curious) clients are only able to learn approximate timing of data processing applications. However, this can be done with no risks involved or resources spent.
- An honest host is able to see input sizes and also learn decently accurate timing of

the application[2].

Millisecond-level timings can often be directly learned from logs or by observing the process table. We consider such coarse grained measurements to not be a security risk. A host that wants to learn fine-grained timings to mount side channel or speculative execution attack has to take higher risks and expand more resources.

- A remote client with malicious intent will need to spend a great deal more resources to break into the application host, escalate privileges, and then mount a local attack against the security enclave.
- Furthermore, malicious third parties need even more resources to gain access to any inferable information and mostly likely need to take higher risks to doing so.

Without knowing the data processing algorithm implementation details and precise specification of the CPU it is impossible to accurately characterize what data can be indirectly inferred. For this reason indirectly inferable statistics can be the most difficult resource to protect.

As an example consider attempts to hide size information of tabular data. First of all, for every data column all cells must be padded to the same length. Next, all algorithms that subsequently process that data must not leak the real record sizes via timing or memory access patterns. To be sure of that, a developer: must have a clear understanding what data in the application at any point is sensitive, must be aware of how compilers transform high-level code to machine code, and has to also be aware of what data processing steps are constant time. This kind of know-how must reach down to low-level knowledge of some CPU internals in order to understand what instructions are side-channel safe.

Inferable data can occasionally be either not as important to protect or is already public information. For example, the number of patients suffering from a particular disease is not very sensitive information if that information is taken from general population. However, if that information can be sensitive if it applies to a small sub-group.

For an attacker to infer fine-grained sensitive statistical information it usually needs very high access levels. This means that the attacker has to be hosting the secure enclave and/or has significant resources to conduct a targeted attack remotely. For more details see Section 2.3.

2 Attack vectors

In this chapter we cover four of the attack vectors that can target applications that use Intel SGX: cryptographic primitives and protocols (Section 2.1), enclave surface (Section 2.2), side channels (Section 2.3), speculative execution (Section 2.4), and output inference (Section 2.5). Each section covering an attack vector is independent and gives a brief overview of the vector, covers some proof of concept attacks, and give some guidelines for mitigation. Sections do cross references each other as they are not completely independent. For example, speculative execution and side channel

[2]This information is usually logged without any confidentiality or integrity protection.

attacks are often closely related. The list of attack vectors that we cover is by no means comprehensive and many common vectors (credential compromise, phishing, denial of service, etc.) are beyond the scope of this document.

2.1 Cryptographic primitives and protocols

Attacks against cryptographic primitives and protocols are not specific to SGX. However, it is possible for SGX to make these vulnerabilities more difficult to find and exploit due to attack surface minimization. With a classical approach to security an attacker that has gained total local access would not need to attack cryptographic methods directly. SGX enabled applications may force attacker to resort to exploiting cryptography.

2.1.1 Access requirements

Design or implementation flaws in cryptographic protocols (like TLS) may lead to remote vulnerabilities [DKA+14]. Remote code execution and other similar attacks via buffer overflows are covered in Section 2.2; these means of entry exploit the enclave surface and not cryptography directly.

An attacker with local access is able to retain enclave state and thus apply long-term attacks on encrypted data. This includes brute-forcing encryption keys when weaknesses are eventually found in cryptographic primitives. Furthermore, local privileged attacker can also arbitrarily invoke enclave in a sequential or parallel manner in order to exploit weaknesses in any cryptographic protocols the enclave implements.

2.1.2 Countermeasures

Cryptographic primitives that SGX uses and recommends (AES128, SHA256, ECC P-256) have been well researched and are widely considered to offer strong security. While it is impossible to fully rule out the risk, a rapid advances in research that allows for these primitives to be easily broken are unlikely to happen.

2.1.2.1 Misuse errors

Perhaps simplest to exploit enclave surface attacks target the misuse of otherwise secure cryptographic primitives. This misuse can often stem from poorly designed API and bad library documentation [GS16]. For example misuse errors include:

- misunderstandings of properties a specific cryptographic primitive offers,
- unsatisfied preconditions for secure use of a primitive,
- regular coding errors.

We make the following recommendations to combat misuse errors:

- When available use high-level cryptographic API.

  - Prefer battle-tested and well-documented libraries (when available).
  - Only use low-level API when necessary and always provide a reason why a low-level interface is used.
  - Strongly prefer libraries that have simple and hard-to-misuse interfaces. Functions should have no combination of parameters that directly lead to vulnerabilities. Functions should offer sensible and secure default parameters.

- Use as simple as possible protocols for a given task. Large protocol suits have correspondingly larger attack surface.
- Do not implement custom cryptographic primitives and avoid using existing cryptographic primitives in non-standard ways.
- Establish a code review process.
- Train developers on secure programming and cryptography fundamentals.

2.1.2.2 Long term attacks

A potential attack scenario directly against cryptographic primitives is one where slow progress over years eventually reveals weaknesses that over time lead to practical attacks long (after the primitive has been phased out of use). For example, SHA1 hash has not been considered safe to use since an attack published in 2005 [RO05] showed fundamental weaknesses in design[3]. The first publicly known collision was announced over a decade later in 2017 [SBK+17]. At the time the attack required considerable computing power (6500 years CPU-time and 110 years GPU-time).

To exploit this vector an attacker has to retain encrypted data for a long time (years or even decades) waiting for advances in security research and/or for computing power to become more accessible and cheaper. This kind of long-term approach is available to state-level attackers.

Attacks of this type are one of the least probable and most costly. Most likely there are easier ways for an attacker to achieve the same results. Regardless, to make long term attacks more resource costly to conduct we make the following recommendations:

- Use well-researched cryptographic primitives and protocols and follow up-to-date recommendations [Bar20].
- Use perfect forward secrecy where applicable.
- Limit the amount of data encrypted using the same key.

  In general limit the use of each key. In particular avoid using the same key for different purposes. For example, in most cases it is a good idea to never use the same asymmetric key for both encryption and authentication.

[3]Despite recommendations to prefer replacements since 2010, web browsers only stopped accepting SHA-1 SSL certificates in 2017.

- Do not retain keys unless absolutely necessary.

  For example, when intermediate data processing results need to be stored on disk do not persist the encryption key. Either re-derive key when data is needed again or keep the key temporarily in memory.

- When secrets need to remain protected for long periods of time (say, decades) quantum resistant security becomes relevant.

  Intel SGX alone without other strong security protection is difficult to recommend as a solution for applications that require protection against state-level attackers and need for data to remain protected long periods of time.

2.2 Enclave surface

Enclave surface is the most probable attack avenue against enclaved applications. Attacking enclave via local access is similar to attacking web applications via remote access. In both cases there is a defined and rather limited interface that an attacker can work with. In both cases the goal is to exploit this interface to move an application into an unintended state and by doing so either learn some information or tamper with existing data.

Attack surface of an enclave developed using Intel SGX SDK is defined in three parts:

1. The set of ECALLs and OCALLs specified in EDL files (either switchless or not).

   Attacker with privileged local access is able to arbitrarily use the ECALL/OCALL mechanism. Attacker can replace OCALL implementations as they reside in untrusted part of the application. Attacker can also arbitrarily perform ECALLs to enclave. Using those means attacks can attempt to move enclave into some unintended state.

2. Reads and writes to shared public memory.

   Enclaves have the freedom to read and write non-encrypted (public) memory regions subject to OS restrictions. An attacker with privileged local access is able to arbitrarily manipulate public memory.

3. The CPU architecture and behaviour. Including instruction timings, memory access patterns and other side-channels.

   SGX is very complex and interacts with parts of the CPU in ways that are difficult to intuitively understand. While parts of Intel CPUs, like the instruction set, are very well documented the implementation details are often not specified. Some operations performed in an enclave will affect the CPU state in a way that is visible to a non-enclaved observer.

2.2.1 Notable proof-of-concept attacks

2.2.1.1 The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX

An attack [BCD+18] that allows various memory corruption vulnerabilities in enclave code to be easily exploited. They abuse the Intel SGX SDK to make attacks more effective. Their attack does not rely on elevated privileges or a malicious operating system.

If an enclave does not have vulnerable code then the described attack does not work.

2.2.1.2 AsyncShock: Exploiting Synchronization Bugs in Intel SGX Enclaves

An attack [WKPK16] that relies on enclaves to have memory vulnerabilities (like use-after-free). They show that these vulnerabilities in enclaves are easier to exploit in multi-threaded code and present a semi-automated tool for exploiting synchronization bugs.

If a synchronization bug exists in an enclave the compromised OS can schedule the enclave threads in a way to always trigger said bug. We consider this to be fundamental issue that stems from the design tradeoffs. This proof-of-concept attack shows the importance of developing bug-free enclaves. Avoiding multi-threaded enclaves also fully mitigates this attack vector.

2.2.2 Countermeasures

2.2.2.1 Enclave API design considerations:

- Simplify and minimize the set of OCALLs and ECALLs.

  Simpler interfaces with clear purpose are easier for developers to reason about. A smaller set of calls allows developers to keep a bigger proportion of the enclave surface in mind at the same time.

- Document the enclave surface.

  Clear and thorough documentation allows developers to reason about the enclave surface. Good documentation stands as an excellent starting point for formal specification and proofs if the security level requires that.

- When reasonable split a security application into multiple simple enclaves. While this introduces the complexity of parallel composition it can be worthwhile if enclaves can be compartmentalized [And20, Chapter 10].

  A system that is compromised of multiple enclaves can be more resilient against compromise of a single enclave. Consider splitting application to multiple enclaves if:

  - *each of the enclaves has a clear purpose and single responsibility; and*

  - *when one of the enclaves is compromised the security impact is clear and limited.*

- When security requirements are strict consider formal specification with correctness and security proofs.

2.2.2.2 Secure programming considerations:

- Train developers on secure software engineering to follow best practices.
- Establish a code review process.
- Use memory safe languages.

  To some degree using a memory safe language mitigates risks involved with SGX enclaves being able to read and write arbitrary memory. Memory safe languages reduce the risk of introducing memory safety issues that cause a large portion of all software vulnerabilities[4]*.*

- Use multiple threads in an enclave only if there are very compelling reasons to do so. In general enforce that enclave can only be entered by a single thread. This can be forced by configuring the enclave with `TCSNum` and `TCSMaxNum` parameters set to one.

  Parallel composition of protocols and interfaces is challenging to reason about due to exponentially increased number of possible state transitions. An attacker with local access can trigger any multi-threading bug as they fully control enclave scheduling.

- Prefer message passing to shared memory.

  Whenever trusted (enclaved) and untrusted parts of an application need to communicate either prefer the explicit ECALL/OCALL mechanism or some other form of message passing over shared memory.

- Avoid error recovery and fail fast (see *crash-only software*[5]).

  Error recovery paths are most likely to contain bugs as they get least amount of testing. To implement reliable error recovery developers have to not only reason about correct states of the application but also incorrect states.

Side-channel attack mitigations are covered in Section 2.3.4.

[4] `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/` Accessed: 2021-03-18

[5] `https://en.wikipedia.org/wiki/Crash-only_software` Accessed: 2021-03-18

2.3 Side channels

Side-channel attacks exploit implementation details of an algorithm to learn more information about input data that what is intended by the abstract description of the algorithm. An abstract description of an algorithm does not usually tell anything about how the program is evaluated in practice, what kind of hardware the code is executed on, or what kind of underlying data structures are used. All these implicit properties may give rise to hidden information leaks. Side-channel attacks may exploit timing information, instruction counts, network packet sizes, probed memory locations, or power consumption.

Side-channel vulnerabilities are an old topic of research spanning decades. Recently Meltdown and Spectre attacks exploited side channel vulnerabilities and allowed attackers with unprivileged local access to leak memory contents from one application to another. A remote variant of Spectre [SSL+19] also exists. SGX intends to protect against privileged local attackers that, compared to remote attackers, have a much larger set of possible side channels to exploit. In this security model any information leaked via side channels may mean compromised security. Side channels are one of the few known attack vectors against Intel SGX enclaves.

> "As opposed to protection against cryptographic attacks, protection against side channel attacks is never expected to be absolute: a determined attacker with a massive amount of resources will sooner or later be able to break an implementation. The engineering challenge is putting in enough countermeasures such that the attack becomes too expensive to be interesting. Products that offer a high level of security typically implement countermeasures on multiple levels." (The Keccak Team[6])

2.3.1 Sources of side channel vulnerabilities

The following code structures can lead to side channel vulnerabilities when they depend on secrets:

- Conditional branches and loop bounds.
- Array/memory look-ups.
- Variable latency CPU instructions.

Side channels are everywhere. For example, even computing the statistical mean of a data set (see Listing 1) leaks the approximate number of input elements via the running time – the longer the program runs the more input it was given. This side channel is even exploitable by a remote attacker. With local access attackers can determine the number of input elements more precisely. Fortunately, in most applications the number of data elements is not considered to be sensitive information. This example is artificial at best but does demonstrate how even the simplest piece of code that does not seem to have any obvious security flaws may reveal unintended information via side channels.

[6] `https://keccak.team/files/NoteSideChannelAttacks.pdf` Accessed: 2021-03-18

Listing 1: Side-channel vulnerable mean

```
float mean(float * arr, size_t n) {
    float s = 0.0f;
    for (size_t i = 0; i < n; ++ i) {
        s += arr[i];
    }

    return s / static_cast<float>(n);
}
```

Whether or not the code in Listing 1 actually has a vulnerability depends on if we consider the data set size to be sensitive information. In most cases it is not, but in some cases it might be. In particular when computing `mean` of many small subsets.

When we consider the number of inputs `n` to be private the code example leaks information in 3 distinct ways. It has a loop with an upper bound `n`, memory is accessed `n` times which also leak approximate size, and the division operation can leak some information both about final output `s` and input `n` (via instruction timing channel). If `n` is considered to be public then only the final division operation leaks some information about the magnitude of `s` to a highly motivated[7] local attacker.

Memory accesses may also lead to timing vulnerabilities. Each RAM read[8] will cache a small region of memory so that subsequent accesses to that region would be faster. It is a natural optimization as most memory accesses are consecutive. This way the high bandwidth of RAM can compensate for its relatively poor latency (compared to on-die registers and caches). Consider two consecutive memory accesses to the same memory object, the first with index `i` and the second with `j`[9]. Assuming that no part of the array has been previously cached then the speed of the second memory access depends on the distance between the two indices. Hence, timing information leaks if the index `i` is close to `j`.

Side-channel attacks can also exploit shared hardware resources. For example in Intel CPUs the L1 cache can be shared by multiple threads if these threads happen to be executed simultaneously on the same core. This is called *simultaneous multithreading* or SMT (Intel's proprietary implementation is called *hyper-threading* or HT). This means that in some situations processes can observe each others read and write locations. Even locations of reads performed by an enclave can be observed.

We take a conservative policy and assume that a privileged attacker is able to observe all memory read and write patterns. A consequence of this assumption is that many common algorithms leak a significant amount of information. For example, most sorting algorithms leak the structure of the input data. The exact information depends on the algorithm used but for instance if the algorithm (e.g. insertion sort, selection sort, quicksort) performs no memory writes the input must have been already sorted. In other

[7] Instruction-timing attacks require repeated measurements and high resolution timing information.

[8] Memory writes lead to similar issues via various caches and line fill buffers.

[9] This is a heavily simplified scenario. In practice these memory accesses may be executed out-of-order. To force these accesses to happen consecutively the second read location must depend on the result of the first read. For instance `j = mem[i]; k = mem[j];`.

Listing 2: Side-channel protected select

```
// Denotationally: select(b, x, y) == b ? x : y
unsigned select(bool b, unsigned x, unsigned y) {
    // well-defined, when b == 1 then m == UINT_MAX
    unsigned m = - (unsigned) b;
    return (x&m) | (y&~m);
}
```

words, attackers are able to keep track of the permutation between input and output. If an attacker happens to have a priori knowledge about the ordering of input data then that may already leak sensitive information about the output.

Consider a database of name-salary pairs that is initially sorted by names. When the database is reordered by salary then an attacker capable of tracking memory access patterns will be able to establish a decently accurate mapping between names and salaries. However, if the database is initially ordered randomly then only an approximate shape of the distribution of salaries leaks. Note that, when comparison-based sorting is used then the salary distribution itself does not leak because those sorting algorithms have identical memory accesses patterns under all order preserving transformations. For example sorting the array `[3,1,2]` yields same access pattern as sorting `[10000,9,370]`.

Last but not least we must note that not all Intel CPU instructions are constant time. There is a number of useful operations that have latencies that vary with input, most notably integer division executes in a fewer cycles on certain input ranges and can leak approximate information about how large the instruction inputs are (we will discuss techniques to avoid instruction-timing leaks later). Luckily extracting information with this side channel requires great effort from the adversary. It requires thorough code analysis and multiple precise measurements. Even then an attack may not be able to extract precise secret value but only some rough statistical information.

2.3.1.1 Maintenance and complexity of mitigations

Naive attempts at hardening side-channel vulnerable code may fail due to compiler optimizations. In many circumstances code that looks like it should not have conditional branches will have them on the machine code level. Code that makes use of Boolean logic operations is particularly vulnerable to this.

Consider the attempt to implement a side-channel safe procedure `select` (Listing 2) to pick one of two values based on a Boolean. This code is side-channel safe on most platform, compiler, and optimization flag combinations. But in some rare cases it might not be. For example, using clang (versions 3.0 to 8.0) to compile for 32-bit platform (`-m32 -march=i386`) produces machine code that is vulnerable to side-channel attacks due to a jump instruction.

When implementing side-channel safe primitives one must hence not only be careful implementing the algorithms but also continuously and diligently test and verify that the machine code output has remained safe over code changes and updates to compiler(s) across various supported CPU architectures.

Listing 3: Side-channel vulnerability from use of side-channel safe primitive

```
float fselect(bool b, float x, float y) {
    union { float f; unsigned u; } X, Y, Z;
    X.f = x;
    Y.f = y;
    Z.u = select(b, X.u, Y.u);
    return Z.f;
}
```

Listing 4: Assembly for side-channel vulnerable select (GNU syntax)

```
select:
        testl   %edi, %edi
        cmovel  %edx, %esi
        movl    %esi, %eax
        retq
fselect:
        testl   %edi, %edi
        jne     .LBB1
        vmovaps %xmm1, %xmm0
.LBB1:
        retq
```

Furthermore, a primitive that compiles to a side channel safe machine code might not remain safe under composition. The aforementioned function `select` will compile to using a `CMOV` instruction with clang compiler (with `-O2` optimization level for a 64-bit platform). This is a side-channel safe implementation on current Intel platforms. Surprisingly, if used to implement oblivious selection of floating-point numbers (Listing 3) a vulnerability is introduced.

Machine code that is produced when compiling `select` and `fselect` can be found in Listing 4. Notice how `fselect` performs a conditional jump. This is a very short jump but even those are distinguishable for a local attacker. This concrete issue can be solved by forcing the compiler to never inline `select`. But the example highlights the importance of side-channel safety review and testing. Great care must not only be taken to verify that primitives are safe but extra care must also be taken to make sure that composition of those primitives remains safe.

2.3.1.2 Practical attack resources and tools

In practical terms to track timing and memory access patterns a malicious host can use SGX-Step [BPS17]. It is a tool that allows OS level adversaries to interrupt victim enclaves after every single instruction allowing side-channel observations to be made at extremely fine-grained resolution. Hence, we must assume that local attackers are fully able to track the control flow path taken by enclaved code, including the ability to distinguish if-branches that execute instructions that in total have equivalent timings [BPS18]. All conditional branches leak to local attackers.

2.3.2 Access requirements

Size-information is exploitable via (remote) snooping attacks. Relatively coarse grained timing information also leaks remotely.

Most powerful side-channel attacks leverage local access. Control over operating system gives attacker the power to single-step enclaved applications and allows them to gather various side-channel information after each executed instruction. Access to physical hardware can facilitate even more attacks and allows attackers to bypass some software-based mitigations. Mitigations against local attackers are the most challenging and expensive to implement.

2.3.3 Notable proof-of-concept attacks

2.3.3.1 Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic

> "Nemesis-style interrupt timing attacks leak this information at an instruction-level granularity, which allows to precisely isolate (successive) data-dependent instruction timing measurements in a larger enclaved computation." (Jo Van Bulck, Frank Piessens, and Raoul Strackx)

Published in 2018 the paper [BPS18] describes very fine-grained attacks that leverage timing and interrupt counting side channels. The attack relies on the ability to externally interrupt enclaved execution. We consider this side-channel to be fundamental and unlikely to be mitigated in hardware in the near future. Timing-attack resistant processor design remains an active research topic.

They are able to extract the secret lookup key from a (timing vulnerable) binary search function of the Intel SGX SDK. Among other results, they clearly demonstrate that latencies of `DIV` and `IDIV` instructions increase with the number of significant digits in the dividend. Thus, (integer) division leaks secret input(s) magnitude.

2.3.3.2 High-resolution side channels for untrusted operating systems

The paper [HCP17] presents side-channel attack using timer interrupts and cache misses to partially recover images from libjpeg running in an SGX enclave. The attack incurs heavy overhead to the enclave process. They provide an example with an overhead of over 3000x (219 seconds compared to 62 milliseconds).

This is a clear and practical demonstration that running existing code (such as libjpeg) in SGX enclave without further mitigations does not provide sufficient protection against a malicious host. In the particular case it has been shown that an automated tool could be developed that can extract images from any enclave that uses libjpeg.

2.3.3.3 Single Trace Attack Against RSA Key Generation in Intel SGX SSL

The paper [WSB18] identifies a critical side-channel vulnerability in RSA key generation of Intel SGX SSL and the underlying OpenSSL library. The attack allows to recover 16 bits of one of the two prime factors of the public key by observing a single execution of the algorithm. The vulnerability was fixed in subsequent OpenSSL and Intel SGX SSL releases.

Such critical side-channel vulnerabilities are particularly dangerous when they occur in cryptographic primitives. Even a small number of leaked bits from a private key can compromise an entire enclave.

2.3.3.4 Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX

In this paper [WCP+17] memory side-channels of Intel SGX are classified. Additionally they propose schemes for making attacks more difficult to detect. To demonstrate effectiveness they attack the spelling engine Hunspell, the font rendered FreeType and EdDSA (from libgcrypt v1.7.6, not side-channel hardened) running in enclave. In the latter attack they are able to fully recover the secret session key.

The demonstrated memory side-channel attacks are not effective with appropriate countermeasures (see below).

2.3.3.5 MemJam: A False Dependency Attack against Constant-Time Crypto Implementations

MemJam [MWES19] demonstrates an attack against side-channel hardened software AES from Intel Integrated Performance Primitives (IPP). They demonstrate key recovery against two different implementations which are secure against cache timing attacks.

We have to note that the Intel SGX SDK does not use this AES implementation. Additionally, AES implemented using AES-NI instructions is not vulnerable to this type of attack. Finally, this attack requires hyper-threading.

While this attack is defeated by side-channel countermeasures (see below) they demonstrate that modern microarchitectures are so complex that countermeasures that have long been believed to be effective can turn out to not be so. In particular MemJam (as did CacheBleed) is able to track memory accesses within a same cache-line.

2.3.3.6 CacheBleed: A Timing Attack on OpenSSL Constant Time RSA

CacheBleed [YGH17] is not an attack that targets SGX enclaves. However, it is a relevant side channel attack that uses a timing information to extract a secret key from OpenSSL constant time RSA implementation. Parallel implementation of CacheBleed allows complete recovery of the secret key in minutes by observing decryption operations (60% of the key is recovered by observing 16000 decryptions).

Intel has historically promoted avoiding memory accesses at coarser than cache line granularity as a side-channel attack countermeasure. CacheBleed clearly demonstrated that this mitigation is not effective (on HT enabled platforms).

> "CacheBleed demonstrates that secret-dependent memory access at a finer than cache line granularity is vulnerable to timing attacks. In particular, the scatter-gather technique is not safe. Sensitive software should be written to eliminate all secret-dependent memory access." (Yuval Yarom, Daniel Genkin, and Nadia Heninge)

As with MemJam this attack also requires the victim and the attacker processes to be located on the same physical core. Disabling hyper-threading mitigates this attack fully.

2.3.3.7 SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control

SGX-Step [BPS17] is an open-source framework and a tool for side-channel attack research on Intel SGX. It facilitates the ability to interrupt the enclave execution at every single instruction.

2.3.3.8 Plundervolt: Software-based Fault Injection Attacks against Intel SGX

Up until recent research into undervolting-based fault-injection attacks the integrity of SGX enclaves was not compromised and vast majority of research was focused on breaching confidentiality.

> "We present the Plundervolt attack, in which a privileged software adversary abuses an undocumented Intel Core voltage scaling interface to corrupt the integrity of Intel SGX enclave computations. Plundervolt carefully controls the processor's supply voltage during an enclave computation, inducing predictable faults within the processor package. Consequently, even Intel SGX's memory encryption/authentication technology cannot protect against Plundervolt." ([MOG+20])

Because the attack is fully software controlled it was possible for Intel to mitigate the issue by modifying remote attestation to allow for clients to verify that software-based undervolting is disabled.

2.3.3.9 VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface

To overcome software-based mitigations against undervolting attacks hardware-based voltage control can be used.

> "To this end, we have built VoltPillager, a low-cost tool for injecting messages on the Serial Voltage Identification bus between the CPU and the voltage regulator on the motherboard. This allows us to precisely control the CPU core voltage. We leverage this powerful tool to mount fault-injection attacks that breach confidentiality and integrity of Intel SGX enclaves. We present proof-of-concept key-recovery attacks against cryptographic algorithms running inside SGX." ([CVM+21])

This threat is somewhat managed by the fact that undervolting attacks are not simple to carry out. As with many other side-channels attacks the targeted code has to be carefully isolated unless vulnerability is exploited in a reusable piece of code. Furthermore, undervolting can be unreliable. Too much undervolting leads to system instability that can be detected by remote clients. Too little undervolting makes the attack impossible as no faults are injected. The sweet spot is specific to hardware and can even depend on the concrete CPU binning.

Hardware-based side-channel attacks against Intel SGX have not yet been explored sufficiently and we expect this avenue to be a fruitful research direction.

2.3.4 Countermeasures

Because leakage strongly depends on the power of the attacker, the more privileged the attacker the more information they are able gain access to. Hence, the first line of defense must be strict access control to make local access as difficult to attain as possible.

It is not feasible to mitigate all possible side-channel vulnerabilities in a non-trivial data processing application. In most cases doing so would increase the cost of development prohibitively. To find the sweet spot between security and too excessive side-channel hardening it is important to clearly specify what (statistical or otherwise) information must be kept secret and what is allowed to leak to remote or privileged local attackers.

There are two approaches to mitigating side channel vulnerabilities. Either eliminate the release of information, or break data associations so that any leaked information is useless to an attacker.

2.3.4.1 Eliminate release of information

For protecting user (meta)data the most obvious approach is to make sure that no sensitive information leaks. To achieve this user data needs to be processed in a side-channel safe manner. Luckily, many simpler algorithms are already side channel-safe or only require conceptually minor modifications. On the other hand, many more complicated algorithms are highly challenging (area of active research) to make side channel safe. One such class of algorithms are cryptographic primitives.

For protecting cryptographic keys it is utmost important to use side-channel safe protocols and cryptographic algorithms. Choose by-design side-channel safe primitives when possible. When no such primitive is available opt for well-researched ones with implementations that have no known strong vulnerabilities. For example:

- Hardware based AES (Intel AES-NI [ADF+10]) is known to have strong side-channel safety guarantees. Only known attacks are via fault-injection [CVM+21] and extensive power-analysis [SRH16, LKO+21].
- Salsa20 [Ber08] stream cipher and its variant ChaCha are resistant to timing and cache attacks by design.
- Side-channel safety of RSA is thoroughly researched [MML+20] and has hardened implementations readily available. As demonstrated by CacheBleed [YGH17] full mitigation remains challenging.
- In comparison the safety of elliptic curve cryptography implementations is not quite as well researched. Well-known techniques from protecting RSA and software-based AES lead be used for more resilient implementations. Fully side-channel resilient implementations are challenging [LR21].

Avoid hand-rolling constant-time cryptographic algorithms. Implementations are very tricky to get right [Por] and vulnerabilities are found even in commonly used techniques thought to be secure [YGH17, MWES19, Rya19].

For protecting data processing code developers have to make sure that there are no secret-dependent conditional branches, loop bounds or memory accesses. As a rule of thumb an algorithm is side-channel safe if for all possible secret input values it executes exactly the same instructions and performs exactly the same set of memory accesses (in the same order). Some caveats apply. For example few arithmetic instructions, like `DIV` and `FSIN`, have a latency that depends on input values.

2.3.4.2 When possible use side-channel hardened high-level algorithms

Rather than implementing a custom solution for a specific domain use a combination of side-channel protected algorithms that are more general. Even if there are significant performance regressions from doing so. Premature optimization is the root of many data leaks and other security regressions. Optimize only once the need arises and performance bottleneck has been clearly determined.

2.3.4.3 Minimize side-effects in secret-processing code

Depending on the scope, side effects can be impossible to handle in a side-channel safe manner. For example, it is not possible to make effects that require operating system interaction non-observable. In most mainstream languages side effects can creep up surprisingly; even memory accesses are usually side-effectful.

Limited forms of side effects can be manageable. Like local enclave state transitions. Side effects that depend only on public inputs are safe. Of course, the side-effectful function not only has to exclusively take public parameters it must also not be called in a secret dependent way (all control flow paths that invoke the function are determinable from public information).

Listing 5: Side-channel protected one-armed conditional

```
auto const s1 = f(s);
s = select(cond, s1, s);
```

2.3.4.4 Eliminate branches that depend on secrets

Instead of branching on a secret and then executing either of the branches evaluate both branches and select one of the resulting values obliviously based on the secret bit. This approach does not work if either of the branches performs side effects (like transitioning the program state).

For example, instead of transforming program state `s` conditionally in-place

```
if (cond) { s = f(s); }
```

transform the state into copy `s1` and then commit the copy conditionally in an oblivious manner. In imperative code the state is usually implicit and as such, for this technique to work, needs to be made explicit. Depending on the size of the state and complexity of the function `f` this solution is significantly slower.

2.3.4.5 Avoid loops with secret-dependent bounds

Loops should be considered to always leak the bound.

If a loop bound is deemed to be confidential then determine a reasonable upper bound and always perform that number of iterations. The loop body needs to be transformed to handle extra iterations to preserve the semantics of the original. Frequently data that is being processed needs to be padded and the loop body has to be able to correctly and securely handle the extra padding.

To protect the previously seen function `mean` against some side channel attacks we can do the following (using side-channel safe `fselect`). The function now takes 3 parameters. The pointer to array `arr`, the actual size of the array `n` and an upper bound `bound` that is acceptable to be leaked. The assumption is that `arr` holds room for `bound` number of elements. We iterate over the entire capacity of `arr` and either increment the sum by the present element or by 0. We select which number to add obliviously. If we know beforehand that all padded elements are 0 then oblivious selection is not necessary.

In the above implementation care must be taken that `fselect` is not inlined and optimized away to a side-channel vulnerable machine code. One possible way to achieve this (at the expense of performance) is to make sure that `fselect` is not inlined at all and is a completely opaque function call to the compiler. Disabling link-time optimizations is also necessary.

In order to avoid comparisons `(i < n)` a sufficiently smart compiler may also split the loop into two ranges. One spanning from `i` to (excluding) `n` and one spanning from `n` to (excluding) `bound`. That would also lead to a side-channel vulnerable implementation.

Listing 6: Attempt at implementing side-channel protected mean

```
float mean(float * arr, size_t n, size_t bound) {
    float s = 0.0f;
    for (size_t i = 0; i < bound; ++ i) {
        s += fselect(i < n, arr[i], 0.0f);
    }

    return s / static_cast<float>(n);
}
```

Listing 7: Side-channel protected mean

```
float mean(float * arr, size_t n, size_t bound) {
    float s = 0.0f;
    for (size_t i = 0; i < bound; ++ i) {
        s += obl::fselect(obl::lt(i, n), arr[i], 0.0f);
    }

    return obl::fdiv(s, n);
}
```

To avoid this optimization we can move the comparison to a function that is opaque to the compiler.

The code still has an instruction-level timing side-channel via floating-point division. This can be remedied by implementing division in a side-channel protected manner. One approach is to approximate the result via a binary-search that performs a fixed number of iterations and manipulates the search bounds obliviously. The most complicated part of the implementation is handling special cases in an oblivious manner (infinities, zero, nan, subnormals) and porting various floating-point related functionality to be oblivious from the C++ standard library. In fact, this is a considerable engineering effort. In the end a fully safe (according to our knowledge) implementation of statistical mean is presented in Listing 7. Oblivious operations have been moved to the namespace `obl`.

2.3.4.6 Hide data record size

In many cases the size of data records can leak sensitive information. For example a person's name length is sensitive (especially in smaller data sets), the length of a salary string tells a lot about a person's wealth, and some diseases can be uniquely identified from the name length alone.

Avoid textual data and *stringly* typing in general. For instance store salary as 64-bit integers as opposed to strings. Operations (arithmetic, comparison) on register-wide integers do not leak information about magnitudes.

When applicable convert textual data to identifiers and store the mapping between identifiers and names in another table[10].

[10]Of course, that extra table has to be handled in side-channel protected manner.

When textual data cannot be avoided pad it to a reasonable upper bound. If no obvious upper bound exists data can be padded in rough increments. For example, in a coarsely increasing increments with the smallest padding starting at least the median string length. Adopt data processing algorithms to work on padded data in a side-channel safe manner while keeping the actual length (and the amount of extra padding) a secret.

2.3.4.7 Avoid secret-dependent array/memory accesses

Frequently it is important to look up information based on private information from some map, array or memory region. This need arises often in graph algorithms like social network analysis. These operations can lead to vulnerabilities as memory accesses are traceable by local adversaries and cache timing attacks can leak information to even remote adversaries.

One way to hide memory access patterns is via using *oblivious RAM* (ORAM) that is designed for this exact purpose. However, ORAM can be an order of magnitude slower than regular random access memory (even with SGX encryption and ECALL overhead). Most ORAM implementations are designed to work in a client-server model where actions that a client performs leak information about the memory access pattern.

Another way is by breaking data associations (see Section 2.3.4.12) when it is applicable.

2.3.4.8 Avoid calling variable latency instructions on secret-dependent values

As a good rule of thumb the instructions to avoid are division/reminder (both integer and floating-point) and various complex floating-point operations as square root, trigonometric operations, exponentiation, and logarithm. This is of course in addition to all instructions that take memory locations as arguments. Unfortunately, no comprehensive documentation on instruction latencies is provided by Intel, but excellent third party resources [Fog21] are available that give a decent indication whether or not an instruction could be safe to use on a given CPU architecture.

Unfortunately there is no easy way to work around this timing side-channel and the best approach is situational. There are few options to consider:

- Find a way to avoid using variable-latency instructions. For example, in a code that handles geometry trigonometric operations (sine, cosine) can usually be avoided.
- Transform input such that leaking those secrets becomes acceptable (see the next section).
- Use constant-time replacements. *Unfortunately software implementations of such low-level operations are often many times slower.*

Listing 8: Forcing all operations to flush denormalized output to zero

```
#include <xmmintrin.h>
void enable_denormal_flush_to_zero() {
    _mm_setcsr(_mm_getcsr() | 0x8000);
}
```

2.3.4.9 Assume that no floating-point operation is constant time

Majority of floating-point operations are not constant time. This includes all trigonometric functions, square-root and division[11]. Even multiplication operation is not constant time when denormals are present. Majority of standard library operations on floats are also not constant-time.

Hence, whenever writing any floating-point code expect it to not be constant time. Always thoroughly test that the code seems to inhabit constant time behaviour.

2.3.4.10 Avoid denormalized floating-point numbers

All processors that support Intel SGX also support *denormalized floating-point numbers*[12] that cause a **significant** [Daw] slowdown of floating-point operations and, as a result, may create a side-channel vulnerabilities. One way to avoid the possibility of this slowdown is to make sure that floating-point operations do not create denormal values. In modern Intel CPUs denormals are enabled as per default, but there is a *Flush-to-Zero* (FTS) mode that forces any floating-point operation that would produce a denormal as an output instead returns a zero. Either use the following code or some other means to disable denormalized numbers.

Avoid using `-ffast-math` optimization flag. According to GCC documentation "-ffast-math also may disable some features of the hardware IEEE implementation such as the support for denormals or flush-to-zero behavior."

When private floating-point input data is provided detect if any of the values are denormal and handle them appropriately. Either reject entire input or round denormals to 0 in a side-channel safe manner.

Be wary with custom-built side-channel safe floating-point operations that directly manipulate IEEE 754 representation. It is possible to accidentally introduce denormals that can cause side-channel vulnerabilities down the line. To renormalize a manually constructed float one can use following function:

Both clang and GCC compilers produce expected code but ICC optimizes (even when `-ffast-math` is not present) the addition away. In that case a more elaborate approach is needed.

[11] Division of single-precision floats is constant-time operation on later generation of CPUs, division of double-precision floats is not.

[12] `https://en.wikipedia.org/wiki/Denormal_number` Accessed: 2021-03-18

Listing 9: Floating-point value normalization

```
float normalize(float x) {
    return x + 0.0f;
}
```

2.3.4.11 Use masking-based techniques to harden critical components

The countermeasures we have described here do not fully eliminate extremely low-level hardware-based side-channels like electromagnetic (EM) radiation and power usage. While there has been no published attacks that exploit these types of side-channels against modern Intel CPUs[13] we can not rule out their possibility.

Standard way to mitigate against these types of attacks is via *masking*. The idea is to not manipulate value x directly but instead work with random shares $x_1, \ldots, x_n$ such that $x = x_1 \oplus \ldots \oplus x_n$ (this is Boolean masking, regular modular addition can also be used). For an attacker to recover original value x all shares x_i must be recovered. This technique is very similar to how secret-sharing based multi-party computation works. The number of shares correlates with the desired security level.

Because this mitigation is extremely technical to implement we recommend using it to only protect the most critical parts of the application. For instance, code involved in protecting and using private keys.

For a more thorough practical overview see *Note on side-channel attacks and their countermeasures*[14] by The Keccak Team. For a theoretical security analysis of masking-based techniques see [PR13]. Masking-based techniques also have some drawbacks in practice [Por] such as requirement of source of randomness.

2.3.4.12 Eliminate data associations

Occasionally it is easier to not eliminate side-channel leaks but to just make leaks unusable. This can be done by breaking data associations. Recall the example of sorting a database of name and salary records. The database is initially sorted by names and afterwards will be sorted by salaries. Standard sorting algorithm will reveal (at least partially) the permutation that maps the initial input to the ordered output. Hence, an attacker is able to (approximately) match names to salaries. However, if the initial name-ordered data set is randomly shuffled before sorting then the process of sorting by salaries no longer reveals the mapping between input and output. In this case the sorting algorithm does not have to be side-channel safe but the shuffling procedure has to be.

The ground-level primitive operation for breaking data associations is side-channel safe 2-swap. One possible implementation is found below. It can be used in combination with `RDRAND` to implement random swap. This is by no means the only or the best implementation and there is no guarantee that it will remain secure throughout compiler

[13] Existing EM attacks against modern processors can reveal instructions that the CPU is executing. We consider the enclaved program execution trace to already to be a publicly available information.

[14] `https://keccak.team/files/NoteSideChannelAttacks.pdf`

Listing 10: Side-channel resistant swap

```
void swap(bool b, unsigned x, unsigned y,
          unsigned & o1, unsigned & o2)
{
    unsigned const m1 = - (unsigned) b;
    unsigned const m2 = ~m1;
    o1 = x&m1 ^ y&m2;
    o2 = x&m2 ^ y&m1;
}
```

improvements and new revisions of processors. A faster implementation could use `CMOV` that, as of this writing, is constant time and does not modify the state of the branch predictor (see *Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations* [Int] and *Intel® 64 and IA-32 Architectures Optimization Reference Manual* [Int20]).

Breaking data associations in larger sequences can be built using this primitive. One possible approach is to use permutation networks such as the *Omega network* [Law75] or the *Clos network* [Erw41, Clo53] to construct a larger shuffle out of many 2-swaps. Unfortunately, the minimal number of layers in these networks to achieve cryptographic security is still an open problem and as such these shuffles should be viewed as additional layers of security that alone do not provide full resilience against side-channel attacks. Solutions that are decent in performance with logarithmic number of layers and cryptographic security proofs [MR14, CGLS18] are complex in implementation.

An alternative approach to shuffling multiple records is to add a new column filled with unique random numbers to the data set. Next, order the records by the new row. The ordering phase requires side-channel safe sorting of which best known implementations have $O(n \log^2 n)$ complexity and $O(\log^2 n)$ round complexity (sorting networks [Bat68] such as *bitonic mergesort* and *odd-even mergesort*).

One approach to generate a column of random numbers is to generate a random `AES` key and encrypt subsequent numbers with that key. This yields a cryptographically strong random-looking sequence and avoids duplicates. Hashing based approaches could also be used by generating a random key K and computing a sequence of HMAC-s as $\mathrm{HMAC}_K(1), \ldots, \mathrm{HMAC}_K(n)$. We recommend HMAC as it is a well established standard and resistant to extension attacks. Alternatively, one can simply generate sufficiently large random numbers and with high probability they are unique. When generating 2^{32} random 64-bit numbers then more than half of the time the sequence contains only distinct elements. While this approach does not generate all possible permutations uniformly it yields a sufficiently good shuffle when large random numbers are used. If a conservative approach is desired we recommend to not shuffle more than $\left\lfloor 2^{n/2-2} \right\rfloor$ elements using a randomly generated column of n-bit numbers. This lowers the probability of having duplicates in the generated sequence to just three percent.

Note that the commonly used shuffling algorithm *Fisher-Yates*[15] leaks entire permutation via side channels. Firstly, it performs random memory accesses. Secondly, the algorithm

[15] `https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle` Accessed: 2021-03-18

Listing 11: Fault-injection resilient squaring

```
unsigned checked_square(unsigned num) {
    volatile auto v = num;
    const auto r1 = v * v;
    const auto r2 = v * v;
    if (r1 != r2) [[unlikely]] {
        std::terminate();
    }
    return r2;
}
```

needs to sample numbers from arbitrary ranges and the typical methods of doing so (rejection sampling) are not side-channel safe. One can attempt to harden Fisher-Yates with ORAM but that leads to much worse performance compared to previous methods. Furthermore, ORAM implementations are commonly built upon side-channel safe sorting or shuffling.

2.3.4.13 Duplicate and compare

To protect against hardware-based fault-injection attacks the only known software-based mitigation is to compute intermediate results multiple times and verify that all duplicate computations yield the same result. This is difficult to implement manually in all but the most critical code paths. Letting compiler insert duplicated instructions or by rewriting (binary) code is feasible with appropriate tooling. The obvious down side of this mitigation is significant reduction of performance.

Implementing duplicate-compare checks in high-level language like C or C++ is made more difficult by compilers assuming that no hardware-faults can happen. Compilers are very good at eliminating common sub-expressions. A way to force re-computations to happen is to mark all involved variables as volatile. For example, following function computes square of input and offers some resilience against fault injection attacks. Modern compilers (tested with GCC, Clang, ICC, and MSVC) produce machine code that computes multiplication twice and checks for inequality.

2.4 Speculative execution

Speculative (or transient) execution attacks exploit exception or branch mis-prediction events that modify CPU state (like shared caches) in a secret-dependent way. The attacker is able to observe these state changes and may thus learn secret information. This is a notable subclass of side channel vulnerabilities that relies on hardware optimizations (like branch prediction) present in most microarchitectures.

Speculative execution attacks require enclave code to have particular exploitable code patterns. Unfortunately, those kinds of patterns occur surprisingly often in user code and occasionally also in reusable SGX libraries. In the worst case the exploitable code processes private keys and is thus a pathway to completely compromise both

Listing 12: Enclave code fragment vulnerable to speculative execution attack

```
1 void enclave_function(size_t untrusted_index) {
2     if (untrusted_index < array1_size) {
3         uint8_t secret_value = array1[untrusted_index];
4         temp &= array2[secret_value * 4096u];
5     }
6 }
```

confidentiality and integrity guarantees of SGX. If an exploitable function is found in a commonly reused component (such as in the SGX standard library) then easy-to-use exploit tools can be implemented.

In this section we will follow a code example where an attacker controls an untrusted index that it uses to speculatively load otherwise inaccessible memory that holds secret data. The secret is then leaked via a memory timing side-channel. For this attack to work the vulnerable (victim) code needs to perform two loads: first with the untrusted index and second with the previously loaded value. Do note that this is by far not the only way to exploit transient execution. For more variants and a much more thorough overview see *A survey of microarchitectural timing attacks and countermeasures on contemporary hardware* [GYCH18] and *A Systematic Evaluation of Transient Execution Attacks and Defenses* [CBS+19].

In the code example the conditional branch that is guarded by a bounds check can be speculatively executed by the CPU. This means that both load statements may be executed regardless of whether `untrusted_index` is in the bounds of `array1` or not. Next, the attacker is able to use a side-channel vulnerability to learn the value of the secret via a memory timing attack; after executing line 4 the attacker can probe, outside of enclave, all of `array2` to detect which part of the array was loaded into the CPU cache[16]. By doing that the attacker learns the secret value.

In summary, these types of attacks use speculative execution optimization and side-channel vulnerability in tandem to leak secret values. For more details about side-channel vulnerabilities see sections 2.3 and 2.4.2.1.

2.4.1 Access requirements

Theoretically this attack can be conducted remotely if the enclave has a web-facing interface and if a suitable side-channel is found. However, no such attacks have been demonstrated yet and those are the easiest to detect and the most resource-consuming for an attacker. Most dangerous attacks that exploit this vector require local and privileged access.

Local, privileged access enables the use of specialized tools and kernel modules that make some of the attacks of this kind even possible. For example, side-channel leaks are much more powerful if an attacker can force victim enclave and the attacker process to be executed on same physical core in an hyper-threading enabled system.

[16] Often this process needs to be repeated multiple times to learn the secret information with high probability. Such attacks can be very slow in practice.

2.4.2 Notable proof-of-concept attacks

2.4.2.1 RIDL: Rogue In-Flight Data Load

RIDL [vSMÖ+19] enables attacker to leak data across VM and SGX security boundaries despite mitigations against existing attacks. SGX enclaves are vulnerable to their cross-process attacks when SMT is enabled. The attack requires that victim and attacker processes are situated on the same physical core in which case reads and writes trivially leak to the attacker.

They demonstrated that contents of the `/etc/shadow` file can leak from one VM to another using a RIDL attack. The attack is relatively slow and takes 24 hours to leak 16 bytes of the file.

The vulnerability has since been fixed via hardware microcode security updates. Enclaves running on platforms with disabled SMT are not vulnerable to this attack.

2.4.2.2 FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution

Foreshadow [BMW+18] (also known as L1 Terminal Fault or L1TF) attack exploits a speculative execution bug in Intel processors to reliably leak enclave secrets from the CPU cache. They demonstrate the attack by extracting full cryptographic keys from enclaves and forging arbitrary local and remote attestation responses.

The vulnerability has since been fixed via hardware microcode security updates. The attack does not work on platforms with (in hardware) disabled SMT.

2.4.2.3 SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution

SgxPectre [CCX+18] exploits a CPU bug to break the confidentiality and integrity guarantees of SGX enclaves. They show that enclave code can be influenced by programs outside of the enclave such that the control flow of the enclave program can be temporarily altered to execute instructions that lead to observable cache-state changes. By observing these changes an attacker can learn secrets inside the enclave memory or its internal registers.

They applied the attack to extract sealing keys and attestation keys from Intel signed quoting enclaves. The sealing key allows persistent enclave data to be decrypted and the attestation key can be used to forge attestation signatures. This attack completely undermined the security guarantees of Intel SGX.

This attack relies on specific vulnerable code gadgets (patterns) to exist in enclaved code. If no such gadgets exist then the enclaved application is not vulnerable to this attack.

Intel firmware microcode updates have since fixed some more dangerous Spectre variants. However, some transient effects remain observable but these can be either eliminated (Section 2.4.4.2) or mitigated by avoiding side-channel leaks (Section 2.3.4).

2.4.2.4 ZombieLoad: Cross-Privilege-Boundary Data Sampling

In ZombieLoad [SLM+19] proof-of-concept attack it is demonstrated that recently loaded values can leak across logical cores. This means that on SMT (hyper-threading) enabled cores enclaves may leak information via a side-channel to processes situated on the same physical core. The attack broke security guarantees of Intel SGX. The vulnerability has since patched via a BIOS applied microcode update and the attack is also completely mitigated by disabling hyper-threading.

2.4.3 Complexity

Most of speculative vulnerabilities are difficult to exploit and require local privileged access.

One exception is if a speculative vulnerability is found in shared code. In that case automated exploit tools can be developed. Even then an attacker might need to observe thousands of program runs, and may slow the program execution down greatly or may need the ability to rerun programs multiple times on varied inputs.

Usually there are much easier attack vectors to exploit like classic software vulnerabilities (that lead to remote code execution) or attacks directed against data owners.

2.4.4 Countermeasures

The foremost countermeasure is to keep hardware microcode and the SGX SDK up to date. Secondarily, consider software based countermeasures.

Assuming that an attacker has privileged local access there are various avenues to reduce efficacy of such attacks against secure enclaves. To name a few:

- prevent speculation (Section 2.4.4.2)
- prevent reading of inaccessible data (Section 2.4.4.3)
- avoid side-channel leaks (Section 2.3.4)
- obfuscate and randomize (Section 2.4.4.4)

This list of mitigations is by no means complete, For a more detailed overview of Spectre and mitigations see *Spectre is here to stay* [MST+19].

All transient execution mitigations have downsides that contribute to higher relative costs of developing hardened software:

- **complexity** – it is difficult to find correct places to apply mitigations and every mitigation adds development complexity,
- **fragility** – hardware microcode changes, compiler changes, and modifications to dependencies can break existing mitigations,

- **testability** – it is challenging to test if applied mitigations are effective on a given platform, and
- **overhead** – many of the mitigations come with a significant performance impact.

2.4.4.1 When to harden against transient execution attacks?

In general we recommend applying software based mitigations only when it is deemed necessary.
In other cases we make the following recommendations:

- Always use software based mitigations when handling secret keys.

  In particular only use side-channel safe (or hardened) cryptographic primitives that are well researched and understood.

- Use transient execution hardened high-level data structures and algorithms for processing private user data.

  Do so even if it incurs significant performance cost. Do so only if such primitives are readily available as implementing them is challenging.

- Minimize the time that secrets reside in memory.

  The smaller the time span that a secret resides in (encrypted) memory the smaller is the amount of code that an attacker can exploit to extract it (say, via speculative execution).

- Avoid writing low-level code.

  This cannot always be avoided, but prefer using side-channel safe high-level operations. For example, instead of implementing high-performance array processing algorithms that directly manipulate indices implement the same logic using operations such as mapping, filtering, sorting, joining or using iterators and ranges. Often low-level code, in particularly one that manipulates pointers or array indices, is difficult to reason about and thus errors there are more challenging to find.

- While the body of research is still young and practical tools are not plentiful consider using static analysis and code instrumentation (fuzzing) tools that highlight potential speculative execution vulnerabilities.

  - SpecFuzz: Bringing Spectre-type vulnerabilities to the surface [OTSF19]
  - DIFFUZZ: Differential Fuzzing for Side-Channel Analysis [NNP19]
  - Respectre: The State of the Art in Spectre Defenses [OPE18]
  - oo7: Low-overhead Defense against Spectre Attacks [WCG+18]
  - SPECTRE Variant 1 scanning tool[17] (RedHat).
  - MSVC compiler has Spectre 1 pass via `/qspectre` option (Microsoft)

[17] `https://access.redhat.com/blogs/766093/posts/3510331` Accessed: 2021-03-18

Listing 13: Speculative execution resistant branching via a memory fence

```
void enclave_function(size_t untrusted_index) {
    if (untrusted_index < array1_size) {
        sgx_lfence(); // speculative memory load barrier
        uint8_t secret_value = array1[untrusted_index];
        temp &= array2[secret_value * 4096u];
    }
}
```

Listing 14: Speculative execution resistance via masking

```
volatile int poison;
void enclave_function(size_t untrusted_index) {
    if ((poison = (untrusted_index < array1_size))) {
        uint8_t secret_value = array1[untrusted_index] * poison;
        temp &= array2[secret_value * 4096u];
    }
}
```

2.4.4.2 Preventing speculation

One way to avoid speculative execution attacks is to make sure that speculative execution does not happen on branches that are conditional on untrusted inputs.

The most sure-fire way to achieve this is to disable speculation of all branches. This can be done with automated tools that insert speculation barriers into appropriate places. Such a heavy handed solution incurs a great performance cost. The V8 Javascript JIT engine runs the *octane benchmark*[18] 2-3 times slower with speculation barrier on every critical branch. Fortunately, in domain specific applications there are far fewer relevant branches and generally the run-time overhead would be much lower.

To mitigate our previous (Listing 12) example we can insert a memory fence (in Intel SGX SDK `sgx_lfence`) after the bounds check to make sure that the memory is not speculatively loaded on that branch.

For more details on this mitigation technique see *Intel® Software Guard Extensions (SGX) SW Development Guidance for Potential Bounds Check Bypass Side Channel Exploits* [Int18].

2.4.4.3 Preventing reading of inaccessible data

Speculative execution in essence is not the whole story of these types of attacks. A critical part is also accessing information that lies outside of the bounds of an array (or some other memory object). A way to mitigate this speculative execution attack is to simply mask the inaccessible data. These and some other mitigation techniques are explained in more detail in *Speculative Load Hardening* [Car] from LLVM documentation.

[18] `https://chromium.github.io/octane/` Accessed: 2021-03-18

Listing 15: Speculative execution resistance via index masking

```
volatile unsigned array1_mask = array1_size - 1;
void enclave_function(size_t untrusted_index) {
    if (untrusted_index < array1_size) {
        uint8_t secret_value =
            array1[untrusted_index & array1_mask];
        temp &= array2[secret_value * 4096u];
    }
}
```

For example, in our running example we can mask the `secret_value` by tracking a poison bit that is either set to `1` if the branch is actually taken and `0` otherwise. Even if `secret_value` is wrongly speculatively computed its value will always be `0`. Great care must be taken to make sure that the compiler does not optimize computation involving `poison` bit as it may easily undo this mitigation.

This approach has a much smaller running time overhead than explicit memory fences. Downsides are that it is also up to the developer to find and fix potential security critical pieces of code and additionally has to make sure that compiler optimizations will not undo this mitigation. Great care must be taken to test that new platform and compiler combinations remain secure.

When the array has power of two elements then an even faster alternative is masking array indices. This mitigation can have better performance but will increase memory usage if many (or large) arrays are forced to power-of-two length. Again, much care must be taken that the compiler does not elide the masking operation.

2.4.4.4 Obfuscation and randomization

There is a wide class of various obfuscation and randomization techniques available. Most of those only make attacks more costly and time consuming but do no fully mitigate their possibility. Regardless, these are still very useful hardening techniques.

- SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs [SLK+17]
- Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization [HLLP18]
- SGXBOUNDS: Memory Safety for Shielded Execution [KOA+17]
- DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization [BCD+17]
- SGXElide: enabling enclave code secrecy via self-modification [BWZL18]

2.5 Output inference

Output inference is not an attack that exploits implementation details or design weaknesses of SGX enclaves. Rather it is occasionally possible to infer more information from intended application output than was initially expected. In particular anonymization techniques have shown to often fail [NS08, NS09, GKdPC14] as much fewer bits of data can lead to full de-anonymization/re-identification as what is commonly expected. Output inference can be combined with other techniques (side channel attacks) to make attacks more effective. Because output inference attacks are a very general technique independent of trusted execution being used or not we only give a brief overview here.

2.5.1 Access requirements

At minimum output inference requires client level access. Multiple malicious clients in collaboration can infer more from joint output. Higher access levels provide access to side channels that enable further attacks in combination with output inference.

2.5.2 Notable proof-of-concept attacks

This class of attacks is not specific to SGX and there exists a body of literature on de-anonymization, re-identification, and database reconstruction attacks

2.5.2.1 Robust De-anonymization of Large Datasets (how to break anonymity of the Netflix prize dataset)

> "We apply our de-anonymization methodology to the Netflix Prize dataset, which contains anonymous movie ratings of 500,000 subscribers of Netflix, the world's largest online movie rental service. We demonstrate that an adversary who knows only a little bit about an individual subscriber can easily identify this subscriber's record in the dataset." ([NS08])

2.5.2.2 De-anonymizing Social Networks

> "To demonstrate its effectiveness on real-world networks, we show that a third of the users who can be verified to have accounts on both Twitter, a popular microblogging service, and Flickr, an online photo-sharing site, can be re-identified in the anonymous Twitter graph with only a 12% error rate." ([NS09])

2.5.2.3 Why Your Encrypted Database Is Not Secure

> "In this paper, we take a system-centric view of encrypted databases and investigate what an attacker would learn in a realistic scenario: stealing a disk, performing SQL injection, or rootkitting the OS. We demonstrate that a "snapshot" attacker, which is the main security model of most encrypted databases, is largely a myth. Modern DBMS's keep logs, caches, and data structures that, in any realistic snapshot attack, reveal information about past queries. This leakage is inherent in today's production environments because a DBMS must maintain caches and other metadata to adapt the system to the workload and help manage its performance." ([GRS17])

2.5.3 Countermeasures

First and foremost the amount of output and how often it is provided needs to be well thought out. These attacks usually stem from underestimating information leakage from the design phase of the application. Classical techniques such as statistical disclosure control [HDFF+10] should be applied.

Do not give input providers the control to run analysis and extract output on demand. This could lead to attacks where queries are repeatedly run on carefully chosen inputs to see how it affects the output. Differences in output can leak information about inputs that should be hidden from the attacker.

Differential privacy (DP) is a mitigation technique that has steadily gained use in practice [EPK14, Tea17] and recently the 2020 US Census is design to apply DP for publishing results [Abo18]. DP is still a very active research topic. A challenge with DP is parameter selection which is not always intuitive [TKB+17] and requires trade-offs between privacy and accuracy [Haw20].

References

[Abo18] John M. Abowd. The U.S. Census Bureau adopts differential privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18. Association for Computing Machinery, 2018.

[ADF+10] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. Breakthrough AES performance with Intel AES new instructions. *White paper*, page 11, 2010.

[AGJS13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, page 7, 2013.

[And20] Ross Anderson. *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2020.

[Bar20] Elaine Barker. Recommendation for Key Management: Part 1 - General. NIST special publication SP 800-57 Part 1 Rev. 5, May 2020.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[BCD+17] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017.

[BCD+18] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *USENIX Security Symposium*, pages 1213–1227. USENIX Association, 2018.

[Ber08] Daniel J. Bernstein. The salsa20 family of stream ciphers. In *The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2008.

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, pages 991–1008. USENIX Association, 2018.

[BPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX@SOSP*, pages 4:1–4:6. ACM, 2017.

[BPS18] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *ACM Conference on Computer and Communications Security*, pages 178–195. ACM, 2018.

[BVR15] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: mixing static and dynamic typing for information-flow control in haskell. In *ICFP*, pages 289–301. ACM, 2015.

[BWZL18] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. SGXElide: enabling enclave code secrecy via self-modification. In *CGO*, pages 75–86. ACM, 2018.

[Car] Chandler Carruth. Speculative load hardening. `https://llvm.org/docs/SpeculativeLoadHardening.html`. Accessed: 2021-03-01.

[CBS+19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, pages 249–266. USENIX Association, 2019.

[CCX+18] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. SgxPectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018.

[CGLS18] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Cache-oblivious and data-oblivious sorting and applications. In *SODA*, pages 2201–2220. SIAM, 2018.

[Cim20] Catalin Cimpanu. First fully weaponized spectre exploit discovered online. `https://therecord.media/first-fully-weaponized-spectre-exploit-discovered-online/`, 2020. Accessed: 2021-03-03.

[Clo53] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[CVM+21] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. VoltPillager: Hardware-based fault injection attacks against Intel SGX enclaves using the SVID voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association.

[Daw] Bruce Dawson. That's Not Normal–the Performance of Odd Floats. https://randomascii.wordpress.com/2012/05/20/thats-not-normalthe-performance-of-odd-floats/. Accessed: 2021-02-26.

[DKA+14] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of Heartbleed. In *Internet Measurement Conference*, pages 475–488. ACM, 2014.

[EPK14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *CCS*, pages 1054–1067. ACM, 2014.

[Erw41] Edson L Erwin. Telephone system, 1941. US Patent 2,244,004.

[Fog21] Agner Fog. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. https://www.agner.org/optimize/instruction_tables.pdf, 2021. Accessed: 2021-03-08.

[GKdPC14] Sébastien Gambs, Marc-Olivier Killijian, and Miguel Núñez del Prado Cortez. De-anonymization attack on geolocated data. *J. Comput. Syst. Sci.*, 80(8):1597–1614, 2014.

[GRS17] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *HotOS*, pages 162–168. ACM, 2017.

[GS16] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Secur. Priv.*, 14(5):40–46, 2016.

[Gue16] Shay Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptol. ePrint Arch.*, 2016:204, 2016.

[GYCH18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27, 2018.

[Haw20] Michael B. Hawes. Implementing differential privacy: Seven lessons from the 2020 United States census. *Harvard Data Science Review*, 2020.

[HCP17] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX Annual Technical Conference*, pages 299–312. USENIX Association, 2017.

[HDFF+10] Anco Hundepool, Josep Domingo-Ferrer, Luisa Franconi, Sarah Giessing, Rainer Lenz, Jane Longhurst, E Schulte Nordholt, Giovanni Seri, and P Wolf. Handbook on statistical disclosure control. *ESSnet on Statistical Disclosure Control*, 2010.

[HLLP18] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating branch-shadowing attacks on Intel SGX using control flow randomization. *CoRR*, abs/1808.06478, 2018.

[Int] Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. `https://software.intel.com/security-software-guidance/secure-coding/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations`. Accessed: 2021-02-25.

[Int18] Intel® Software Guard Extensions (SGX) SW Development Guidance for Potential Bounds Check Bypass (CVE-2017-5753) Side Channel Exploits. `https://software.intel.com/content/www/us/en/develop/download/intel-software-guard-extensions-sgx-sw-development-guidance-for-potential-bounds-check.html`, July 2018. Accessed: 2021-03-01.

[Int20] Intel® 64 and IA-32 Architectures Optimization Reference Manual. `https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html`, May 2020. Accessed: 2021-02-25.

[Int21] Intel® Software Guard Extensions (Intel® SGX) Developer Guide. `https://software.intel.com/content/www/us/en/develop/download/intel-software-guard-extensions-intel-sgx-developer-guide.html`, January 2021. Accessed: 2021-03-10.

[JSR+16] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper*, 1(1-10):119, 2016.

[KOA+17] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: memory safety for shielded execution. In *EuroSys*, pages 205–221. ACM, 2017.

[KSC+18] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating remote attestation with transport layer security. *CoRR*, abs/1801.05863, 2018.

[Law75] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. Computers*, 24(12):1145–1155, 1975.

[LKO+21] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[LR21] Victor Lomné and Thomas Roche. A side journey to titan. *IACR Cryptol. ePrint Arch.*, 2021:28, 2021.

[MML+20] Maria Mushtaq, Muhammad Asim Mukhtar, Vianney Lapotre, Muhammad Khurram Bhatti, and Guy Gogniat. Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA. *Inf. Syst.*, 92:101524, 2020.

[MOG+20] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[MR14] Ben Morris and Phillip Rogaway. Sometimes-recurse shuffle - almost-random permutations in logarithmic expected time. In *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 311–326. Springer, 2014.

[MST+19] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019.

[MWES19] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47(4):538–570, 2019.

[NNP19] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. DifFuzz: differential fuzzing for side-channel analysis. In *ICSE*, pages 176–187. IEEE / ACM, 2019.

[NS08] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, pages 111–125. IEEE Computer Society, 2008.

[NS09] Arvind Narayanan and Vitaly Shmatikov. De-anonymizing social networks. In *IEEE Symposium on Security and Privacy*, pages 173–187. IEEE Computer Society, 2009.

[OPE18] OPEN SOURCE SECURITY INC. Respectre: The state of the art in spectre defenses, 2018.

[OTSF19] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing spectre-type vulnerabilities to the surface. *CoRR*, abs/1905.10311, 2019.

[Por] Thomas Pornin. Why constant-time crypto? `https://www.bearssl.org/constanttime.html`. Accessed: 2021-03-02.

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2013.

[PVH19] James Parker, Niki Vazou, and Michael Hicks. Lweb: information flow security for multi-tier web applications. *Proc. ACM Program. Lang.*, 3(POPL):75:1–75:30, 2019.

[RO05] Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. In *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 58–71. Springer, 2005.

[Rya19] Keegan Ryan. Hardware-backed heist: Extracting ECDSA keys from qualcomm's trustzone. In *CCS*, pages 181–194. ACM, 2019.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In *CRYPTO (1)*, volume 10401 of *Lecture Notes in Computer Science*, pages 570–596. Springer, 2017.

[SLK+17] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *NDSS*. The Internet Society, 2017.

[SLM+19] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. *CoRR*, abs/1905.05726, 2019.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003.

[SRH16] Sami Saab, Pankaj Rohatgi, and Craig Hampel. Side-channel protections for cryptographic instruction set extensions. *IACR Cryptol. ePrint Arch.*, 2016:700, 2016.

[SSL+19] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *ESORICS (1)*, volume 11735 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2019.

[Tea17] Apple Differential Privacy Team. Learning with privacy at scale. `https://docs-assets.developer.apple.com/ml-research/papers/learning-with-privacy-at-scale.pdf`, 2017. Accessed: 2021-03-09.

[TKB+17] Jun Tang, Aleksandra Korolova, Xiaolong Bai, Xueqiang Wang, and XiaoFeng Wang. Privacy loss in apple's implementation of differential privacy on MacOS 10.12. *CoRR*, abs/1709.02753, 2017.

[vSMÖ+19] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *IEEE Symposium on Security and Privacy*, pages 88–105. IEEE, 2019.

[WCG+18] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *CoRR*, abs/1807.05843, 2018.

[WCP+17] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *ACM Conference on Computer and Communications Security*, pages 2421–2434. ACM, 2017.

[WKPK16] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *ESORICS (1)*, volume 9878 of *Lecture Notes in Computer Science*, pages 440–457. Springer, 2016.

[WSB18] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *AsiaCCS*, pages 575–586. ACM, 2018.

[YGH17] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptographic Engineering*, 7(2):99–112, 2017.