Česká společnost uživatelů otevřených systémů EurOpen.CZ
Czech Open System Users' Group
www.europen.cz

52. konference
Sborník příspěvků

Hotel Černý orel, Žatec
13.–16. května 2018

Programový výbor:
Jakub Urbanec,
Petr Jiroušek,
Jiří Sitera,
Jan Panoch,
Zdeněk Šustr

Sborník příspěvků z 52. konference EurOpen.CZ, 13.–16. 5. 2018

Plzeň 2017. První vydání.
Editor: Zdeněk Šustr
Sazba a grafická úprava: Zdeněk Šustr, vydavatelství, Praha
e-mail: `vydavatelstvi@sustr.net`

Tisk: Typos, tiskařské závody, s.r.o.
Podnikatelská 1160/14, Plzeň

Příspěvky neprošly redakční ani jazykovou úpravou.

ISBN 978-80-86583-33-4

Obsah

CPU bugs and vulnerabilities

Vít Šembera

1 When Speculation Is Risky: Understanding Meltdown and Spectre

For several last days of 2017, rumors circulated about a serious vulnerability in Intel processors. It wasn't until January 3 that the official disclosure of the Meltdown and Spectre vulnerabilities was made, and it became clear how serious the problems were. To summarize, Meltdown and Spectre both allow malicious code to read memory that they would normally not have permission to.

The vulnerability can allow an attacker to steal information such as passwords, encryption keys, or essentially **anything** that the affected system has processed. Unfortunately, the desire to improve performance significantly compromised the existing security building blocks of mainstream operating systems (Windows, Linux, and Mac OS at least) that protect the privacy of user data. Proof of concept code for both attacks has been made public; no attacks are believed to have used these newly discovered vulnerabilities yet.

This article explains what these vulnerabilities are, how these vulnerabilities arose in the first place, and what vendors are doing to mitigate this threat.

2 Background: What is Memory Isolation

Memory virtualization is one of the basic security concepts in current operating systems. It brings memory isolation between user processes, ensuring one user cannot access and modify another user's data. This is implemented with paging nowadays. There are large and complex kernel tree structures for each process called page tables (PT), describing the mapping between virtual and physical addresses and also defining access privileges. Supporting hardware called the memory management unit

(MMU) in each modern CPU ensures this translation will be as efficient as possible. There are several layers of caches, called the translation lookaside buffer (TLB), that contain translation data. Each TLB miss leads to a time-consuming lookup in the PT hierarchy executed by the page fault trap handler.

Current operating systems like Windows, Linux, and macOS (including their mobile derivatives like Android and iOS) all use the same concept. Initially, it was expected that kernel privileged code and data would be mapped to the same virtual address of each process to simplify kernel code access to user data. Each change of virtual mapping (switching PT) usually leads to TLB flushing and is inevitable during process context switches. There was another reason for mapping kernel space to user virtual space – to reduce TLB misses when changing a virtual address mapping during the switch from user mode to the kernel code and back. This was so common that Intel recommended it as a best practice and allocated a single register to keep a pointer to the actual PT. This register is then expected to be populated during context switches. The ARM architecture, by contrast, has two PT pointer registers instead.

Kernel code/data is protected from direct access by user code with MMU access control. Many kinds of attacks have been discovered exploiting kernel code bugs, leading to privilege escalation and system control. To carry out these attacks, the attacker would need to know two things: what the vulnerability is, and what is the address of related kernel code and/or data.

To defend against this kind of attack, operating systems implemented address space layout randomization (ASLR) years ago. Specifically, in the case of the kernel, it is kernel ASLR (KASLR). Protection is based on keeping the addresses of kernel structures secret from user processes. There are many attacks against KASLR, some of which can be found here. Many of these attacks are based on measuring statistical differences between accessing memory when TLB is hit or missed; these statistics are based on the hardware implementation of the MMU and cannot be easily hidden.

In October 2017, researchers from Graz TU published a proposal to split the process PT into two. One is active when the process is executing kernel code (this is essentially the original PT). The second is a partial copy used for mapping user pages and some additional kernel pages needed for context switching, syscalls (OS system calls), and interrupt handling. The rest of kernel space is invisible; this PT is active when executing in user space. The kernel switches between these PTs when it is entering and

exiting the syscall. This technique was called KAISER (Kernel Address Isolation to have Side-channels Efficiently Removed) and was able to defend against all known KASLR attacks with an added overhead of less than 1%.

Soon enough, work began to implement KAISER. In the Linux kernel, urgent (but silent) work began on a kernel page-table isolation (KPTI) implementation in November of 2017, which was based on KAISER. This part of the kernel is sporadically changed, with discussions taking place long before any code is written. Needless to say, the work to implement KPTI caught the attention of some observers because of its unusual nature.

Changes in this part of the kernel can have a significant impact on performance. Early tests showed a 5% impact in most cases, with worst-case tests indicating a 50% performance hit. On November 16, it became clear that something like KAISER was being implemented in Windows as well. The Linux KPTI was finally committed to the kernel on December 29.

3 Meltdown and Spectre

It soon became clear why KAISER was suddenly being implemented: it was an effective defense against Meltdown.

Both Meltdown and Spectre rely on security flaws in the speculative execution of CPU instructions. Modern processors are so fast that executing instructions in order one-by-one would lead to the CPU waiting for memory access, which takes several hundred clock cycles. Modern CPUs try to execute instructions that are ready for execution while waiting for memory read/write operations. It can be checked later if the instructions that were executed speculatively were correct; if the results are not needed, the effect on the CPU's internal state and memory should be removed. This is called speculative execution and out of order execution.

Unfortunately, some side effects of speculative execution remain in the CPU's state at low levels. For example, if there is an unsuccessful speculative move of a word from memory to a register at the end, the register contains the original value, but the cache is modified by the read cycle. The branch prediction logic buffers contain information about recently taken code branches. The researchers who discovered Meltdown and Spectre used low-level CPU states to gain access to protected memory regions

using so called side channel information transfer.

Meltdown (CVE-2017-5754) allows an unprivileged user to access the complete kernel (and physical) memory of a computer. This attack is relatively simple to execute; to carry it out, attackers need to run their own program on the target system. This attack is particularly damaging to shared systems (such as cloud services), as an attacker with access to one virtual machine can use Meltdown to access other VMs on the same physical system. Meltdown is specific to Intel systems; AMD and ARM processors are not affected.

Spectre (CVE-2017-5753, CVE-2017-5715) is a broader vulnerability. Spectre relies on issues with speculative execution itself to be carried out. In its current form, the attack is more complicated as more prerequisites must be fulfilled. One of them is a code gadget, which must be found in a code shared by both victim and attacker. For some variants of this attack, a branch prediction CPU subsystem must be trained to redirect execution of a code to the selected gadget.

This makes exploitation highly dependent on the CPU version because prediction algorithms and deepness of the prediction buffers differs not only between vendors but across CPU generations. Two Spectre variants are currently known. One variant limits the attack to a single process space; the second requires superuser access. What makes Spectre particularly dangerous is the combination of kernel interpreters and JIT compilers like Linux eBPF which allows an attacker to create and run speculative executed code directly within the kernel context. This could have an impact similar to the Meltdown attack.

The real challenge with Spectre is its mitigation. Unlike Meltdown (which could be mitigated via patches to the operating system), Spectre requires changes to the hardware itself. As a workaround, some vulnerable code can be mitigated by inserting synchronization primitives (like the **LFENCE** instruction on Intel platforms) which effectively stops speculative execution. Another one is using return trampoline approach (Retpoline). This approach requires modification of compilers and careful selection of critical locations, which is non-trivial and cannot be easily done without human interaction; doing otherwise would impose a significant performance penalty.

All modern x86 processors from Intel and AMD, as well as some RISC chips based on ARM, Sparc, PowerPC architecture, are believed to be vulnerable. It makes mitigation of this vulnerability extremely costly, making it a long-term problem for the technology industry as a whole. Ball

is now in the CPU vendors court. They will need to find a way how to keep high performance for their chips but at same time don't ease on security. For sure a speculation algorithm must be changed. It is still an open question what can be done by microcode update and what needs a chip HW redesign.

4 Solutions and best practices

Mitigation must be done across different levels of the computer environment. CPU vendors are releasing microarchitecture updates which will be parts of system firmware. Operating system vendors are releasing kernel and compiler updates. Browsers are going to be patched as well to protect against Spectre. Users and system administrators should react promptly and install available updates to reduce the risk of a successful attack.

More software and firmware updates are about to come in the near future. Unfortunately, detection of an attack is extremely difficult. There are no other signs than on microarchitecture level or unusual system performance statistics like unusually high page fault rate or low cache hit ratio. There are no known malicious code samples for Meltdown and Spectre in the wild but it can change quickly as PoCs are coming out rapidly.

Microsoft has released documents that cover both server and client versions of Windows:

- Windows Server guidance to protect against speculative execution side-channel vulnerabilities
- Windows Client Guidance for IT Pros to protect against speculative execution side-channel vulnerabilities

Note that in order to receive automatic updates from Microsoft, a registry key must be in place on the affected system.

Apple's December updates for macOS (released last December 2017) already resolved the Meltdown vulnerability as well. As noted earlier, patches for Meltdown have been merged into the Linux kernel. It is up to individual vendors to release this update for their distribution; some vendors such as Debian, Red Hat, and SUSE have released bulletins and patches as appropriate.

While no attacks using these vulnerabilities are known to exist in the wild, several proofs of concept have been made publicly available.

5 Which CPU families are affected?

CISC architecture-based central processing units like Intel Core and AMD Ryzen have speculation and out of order execution optimizations logic implemented with hardware and microcode on a chip. On the other hand, processors with RISC architecture like Intel Itanium rely more on the compiler code generator for these types of optimizations. Some CPU families like ARM or IBM Power were formerly strictly RISC but used more CISC-like architecture elements over time and added complex execution pipeline with on-chip supported speculation logic. It is clear that mitigation of vulnerabilities in instruction optimization is much harder for HW-implemented solution. Generally, every CPU that features execution speculation is affected at least by Spectre. The difference is in the complexity of mitigation.

6 How to detect if your system is under attack

Each modern CPU supports the collection of many different performance counters of microarchitectural state changes. Observing these statistics can reveal Meldown and Spectre attacks. The theory for Intel platform using Processor Counter Monitor is described here. For example, deviations in cache miss, instruction retirement aborts or branch mispredictions can be detected during an attack. This is currently the most promising way to detect and abort attack code.

7 Intel releases CPU microcode update for Linux

Intel released a microcode update for all CPUs. Updates are usually part of computer firmware, but Intel released this update for Linux OS directly to speed up the microcode update distribution. It updates CPU microcode from the file stored in the root file system when the kernel starts.

8 Virtualized environment

There has been a lot of questions on how to protect against speculation related vulnerabilities in a virtualized environment. On standalone operating systems, there is isolation between different processes as well as between the user and system space, ensured by the concept of virtual memory management as described above. Virtual machines managed by hypervisor are introducing another level of separation between different guests and between the host and guests.

CPUs today contain HW assisted support for virtualization. Intel, for example, has a special set of VMX instructions for easing hypervisors handling with VMs (and for enabling multi-level virtualization). It also features VT-x and VT-d technology to extend MMU virtual to physical mapping with support for virtualization. Intel did it by combining the guest PT with the host PT to increase the number of PT levels. This way, walking through the PT and TLB logic can remain the same, with just a small memory consumption overhead and slightly more time needed to walk through PT in case of a TLB miss.

Meltdown and Spectre speculation execution attacks can to cross over virtual space boundaries and avoid privilege level limitations to access kernel (and indirectly whole physical) memory. Unfortunately, it also applies to boundaries and privileges in a virtualized environment. Different guest or host memory can be accessed the same way different virtual memory or kernel space can be accessed.

In addition to updating microcode and installing all patches for guests and host operating systems, also hypervisor should be patched on virtualized environments.

Advisories for Vmware are VMSA-2018-0002.1 and VMSA-2018-0004.1. XEN advisory XSA-254 is here. RedHat related information for KVM based virtualization can be found here.

9 Lastest events

Intel has released stable firmware updates that patch the microcode of Skylake, Kaby Lake, and Coffee Lake processors against the Spectre vulnerability. Updates for other Intel processors used in data center environments are also available. Details about these new updates are available from Intel. These fixes will be made available to end users in the form of

BIOS updates, which will be provided by various PC and motherboard vendors as necessary.

Many security teams (and bad guys indeed) are discovering new ways of exploiting CPU speculation combined with side channel extraction. Combined team of Priceton University with Nvidia researcher released a whitepaper in February about automated techniques used for hardware related security vulnerabilities verification. As a result, they have built a PoC using Prime+Probe cache side channel data excavation. Original Spectre and Meltdown used another methodology called Flush+Reload. Prime+Probe has even higher data read accuracy in tests than the former method (99.95% against 97.9%). Good news is that already described mitigation methods are effective also in this variant.

Another team from Ohio State University proved that Spectre attack can be used also to break into Intel SGX enclave. This is really inconvenient as SGX is highly promoted security extension of latest Intel CPUs and already adopted by several ISVs in their products (such like Azure confidential computing) to implement secure functions like private key generation and storage. In short SGX allows to move critical code and data into isolated and AES encrypted memory pages which are unreadable even by CPL0 code. The code can be entered and exited only in specific developer defined entry and exit points with newly implemented CPU instructions. As a consequence, the SGX code needs to be recompiled with Spectre mitigations in place.

CPU vendors are working hard in between on a solution protecting against CPU speculation attacks. Intel announced on March 15 that next generation of processors called Cascade Lake and 8^{th} generation of Core processors shipped in 2^{nd} half of 2018 will include new level of speculation protection called partitioning which should protect against Meltdown and Spectre 2^{nd} variant (1^{st} variant is mitigated in software). As a result, these vulnerabilities should be mitigated without significant performance impact.

JavaScript & spol. nejen pro Java vývojáře

Jaroslav Tulach

Abstrakt

V současnosti běžně používané virtuální stroje umí rychle vykonávat programy napsané v jednom jazyce, či malé množině sobě podobných jazyků. Překladače, správu paměti a vývojové či ladící nástroje je tedy třeba psát znovu a znovu téměř od začátku. To komplikuje život nejen tvůrcům virtuální strojů, ale i vývojářům, jež je používají. Nekonzistetní rychlostní charakteristiky, úplně odlišná vývojová prostředí, nové způsoby konfigurace. To vše musí člověk zvládnout při přechodu z jednoho vývojového ekosystému na jiný. Problémy se ještě znásobí, povšimneme-li si obtížnosti a složitosti komunikace mezi programy napsanými v různých jazycích. To obvykle vyžaduje nákladnou serializaci a opětnou deserializaci datových struktur při volání z jednoho systému do jiného, což je operace, jež bezpečně zabije rychlost jakéhokoli výpočtu. Navíc jsou v současnosti nejvýkonnější virtuální stroje složití bumbrlíčci více připomínající operační systém než něco, co byste si chtěli vložit do svého programu jako knihovnu.

1 OracleLabs

Je tomu již několik let, co se OracleLabs rozhodly založit novou výzkumnou skupinu zaměřenou na překonání výše uvedených nedostatků. Naší vizí je vytvořit univerzální VM, která by poskytla tu nejvyšší výkonnost pro libovolné programovací jazyky a tudíž překonala současná omezení v komunikaci mezi jednotlivými programy a jejich knihovnami. Kromě toho by takováto univerzální architektura poskytla sjednocenou, jazykově nezávislou platformu pro tvorbu nástrojů, zjednodušila jejich použití a údržbu. Při našem důrazu na snadnost začlenění této platformy do existujících a nově vznikajících systémů, by pak mělo stačit přidat tuto univerzální knihovnu a okamžitě by byl k dispozici jakýkoli z téměř neomezeného spektra jazyků.

Abychom tohoto cíle dosáhli, rozhodli jsme se vydat zcela novým směrem výzkumu a po letech experimentování jsme schopni nabídnout systém, který z těchto revolučních myšlenek těží. GraalVM je nejen univerzální virtuální stroj, ale především celý ekosystém navržený pro polyglotní svět!

GraalVM dodá každému jazyku rychlost, umožní jeho propojení s ostatními a to bez jakéhokoli zpomalení. Místo obalování a zapouzdřování datových struktur při přechodu mezi jazyky, objekty používané v **GraalVM** mohou plynule a bez omezení plout z jednoho jazyka do druhého. Tyto jazyky pak s nimi mohou přirozeně pracovat. Díky odstranění bariér mezi jazyky se otvírají nové možnosti psaní polyglotních knihoven a aplikací: Máte vlastní knihovnu v Javě a chtěli byste ji použít v node.js? Napsal váš kolega algoritmus v *Pythonu* a vy byste jej chtěli volat z *Java* aplikace? Provádíte statistické výpočty a zobrazení v jazyce *R* a máte data vypočtena jiným jazykem? Není problém! **GraalVM** dává vývojářům svobodu použít programovací jazyk nejvhodnější pro daný úkol.

2 GraalVM

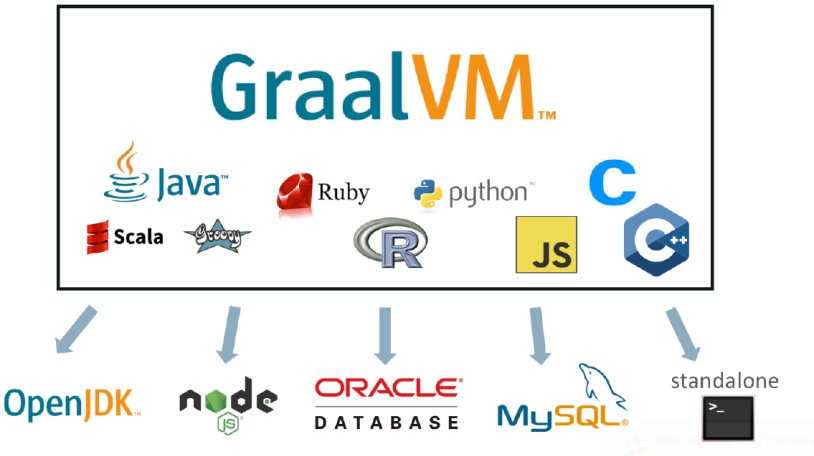
Univerzální virtuální stroj **GraalVM** vám umožní spouštět:

- jazyky postavené nad JVM jako je *Java*, *Scala*, *Groovy* nebo *Kotlin*
- *JavaScript* – samostatně či v rámci *node.js*
- jazyky přeložitelné do LLVM bitkódu jako je *C*, *C++* či *Rust*
- experimentální verze jazyků *Ruby*, *R* a *Python*

GraalVM lze použít buď samostatně a nebo ji začlenit do existujících systémů jako je *OpenJDK*, *node.js* či dokonce integrovat do databází jako je *MySQL* a nebo Oracle RDBMS. Zvláštní pozornost byla věnována vytvoření efektivního mechanismu pro zpřístupnění dat těchto systémů pro jazyky běžících uvnitř **GraalVM** a to bez zbytečných konverzí či alokací proxy objektů. Jazyky mohou tudíž operovat nad daty ve formátech, které se již používají a při tom neztratit nic ze své rychlosti.

Pro jazyky založené nad JVM nabízí **GraalVM** nástroj umožňující vytvoření spustitelných, samostatně distribuovatelných EXE souborů (případně *.so* či *.dylib* knihoven) s okamžitou inicializací a menšími paměťovými nároky. Tento nástroj provádí statickou analýzu kódu s cílem nalezení částí, které se skutečně mohou v průběhu běhu programu použít.

Po té provede úplný ahead-of-time překlad a vytvoří samostatně použitelný soubor pro daný operační systém. Takto vytvořenou knihovnu lze spojit s ostatními přeložitelnými programy a případně i přibalit **GraalVM** kompilátor pro just-in-time překlad dynamických jazyků.



Neopomenutelnou součástí **GraalVM** ekosystému je i podpora pro tvorbu jazykově nezávislých nástrojů. Standardní distribuce **GraalVM** nabízí debugger, profiler a nástroj na analýzu paměti. Tyto nástroje byly vytvořeny nad stabilním instrumentačním API. Výzkumné týmy a nezávislí vývojáři mají tudíž možnost postavit své existující či nové nástroje nad tímto API a získat tak přístup k uživatelům jakéhokoli jazyku implementovaného pro **GraalVM**. Věříme, že **GraalVM** může být univerzálním virtuálním strojem propojujícím nástroje a jazyky v synergii dosud nevídané.

3 Je to vážně tak rychlé?

Jsou opravdu **GraalVM** jazyky tak rychlé, jak o sobě tvrdí? A jak vůbec měřit rychlost jazyka?

Objektivně měřit něco takového je přetěžký úkol. Téměř každý jazyk poskytuje dodatečné vestavěné operace, které již nejsou implementovány v daném jazyce, ale volají do vysoce optimalizované knihovny napsané v Céčku či assembleru. Její autoři si pak dávají dost záležet, aby zrovna ten jejich výpočet byl co nejrychlejší. Tudíž porovnávat rychlost jazyků

na základě takovýchto specializovaných vychytávek nedává moc smysl. Na druhou stranu každý řádný programovací jazyk musí být turingovský úplný (musí podporovat obecné programovací konstrukce jako **if**, **for** a **while**), a tak navrhuji, abychom *turingovskou rychlost* jazyka měřili na základě obecného výpočtu, který pokud možno bude co nejméně využívat zabudovaných zkratek. To je samozřejmě těžké. Pokud se takový obecný výpočet stane známým, tak na něj začnou tvůrci jazyka cílit, speciálně optimalizovat a bude opět po nezávislosti. Tomu lze asi nejlépe zabránit tím, že si člověk napíše svůj vlastní algoritmus a provede si měření sám. Tak jako jsem to já udělal s Eratosthénovým sítem na výpočet prvočísel.

3.1 Ruby

Když jsem se k *OracleLabs* před pár lety přidal, moji kolegové hrdě tvrdili, že jejich implementace **Ruby** je desetkrát rychlejší než jakákoli jiná. *„To určitě!“* pomyslel jsem si. *„Možná je rychlá na nějakém speciálním příkládku, ale jinak to přeci není možné“* a řekl jsem si, že to vyzkouším. Chvíli jsem přemýšlel o nějakém vhodném příkladu a Eratosthénovo síto mi přišlo ideální. Je to výpočet, který může běžet libovolně dlouho (dokud nedojdou prvočísla), který provádí aritmetické operace a který (v mé verzi) alokuje objekty do spojového seznamu. Vcelku vhodný kandidát na měření *turingovské rychlosti*, řekl jsem si a začal psát svůj první program v Ruby:

```
class Natural
  def initialize
    @x = 1
  end

  def next
    @x += 1
  end
end

class Filter
  attr_reader :number
  attr_accessor :next

  def initialize(number)
    @number = number
    @next = nil
    @last = self
  end

  def acceptAndAdd(n)
    filter = self
    upto = Math.sqrt(n)
```

```
    while filter
      if n % filter.number == 0
        return false
      end
      if filter.number > upto
        break
      end
      filter = filter.next
    end
    filter = Filter.new(n)
    @last.next = filter
    @last = filter
    true
  end
end

class Primes
  def initialize(natural)
    @natural = natural
    @filter = nil
  end

  def next
    while true
      n = @natural.next
      if @filter == nil
        @filter = Filter.new(n)
        return n
      end
      if @filter.acceptAndAdd(n)
        return n
      end
    end
  end
end

def ms(start)
  ((Time.now - start) * 1000).floor
end

def fewthousands
  natural = Natural.new
  primes = Primes.new(natural)

  start = Time.now
  cnt = 0
  prntCnt = 97
  begin
    res = primes.next
    cnt += 1
    if cnt % prntCnt == 0
      puts "Computed #{cnt} primes in #{ms(start)} ms. Last one is #{res}."
      prntCnt = prntCnt * 2
    end
  end while cnt < 100000
  ms(start)
end
```

```
puts "Ready!"

count = -1
if ARGV.length == 1
then
  count = ARGV[0].to_i
end

while count != 0
  puts "Hundred thousand prime numbers in #{fewthousands} ms"
  count = count - 1
end
```

Jeho první komponentou je generátor, zvaný **Natural**, přirozených čísel od dvojky do nekonečna. Další částí je **Filter** – to je prvek spojového seznamu do něhož se ukládají již nalezená prvočísla. Ten má i operaci *acceptAndAdd*, jež zkouší dělitelnost nového čísla již známými prvočísly a případně nové prvočíslo přidá do seznamu. Třída **Primes** si pak udržuje hlavu spojového seznamu a umí vydat další prvočíslo. Nakonec je v programu smyčka, která počítá sto tisíc prvočísel a měří čas, jaký to zabralo.

Tato smyčka se opakuje stále dokola. Proč? Protože, a to jsem zatím nezmínil, je hlavním cílem **GraalVM** zrychlit dlouhotrvající výpočty na serverech. Takové výpočty stále do kola opakují to samé a tudíž nevadí, když prvních pár (tisíců) výpočtů je pomalejších. To, co je důležité, je rychlost, které program dosáhne po zahřátí na provozní teplotu. Což je právě případ **GraalVM**: s dalším výpočtem se zrychluje a zrychluje. Proto je třeba provést několik výpočtů na zahřátí a teprve pak změřit dosaženou rychlost.

A jak to měření dopadlo? Ruby, které mám na svém Ubuntu, dokáže jednu smyčku provést asi tak za dvě sekundy:

```
$ ruby -v
ruby 2.3.1p112 (2016-04-26) [x86_64-linux-gnu]
$ ruby ruby/sieve.rb
Hundred thousand prime numbers in 2055 ms
```

Ruby, které je součástí enterprise **GraalVM**, to dokáže za méně než 150 ms:

```
$ /graalvm-0.33/bin/ruby ruby/sieve.rb
Hundred thousand prime numbers in 119 ms
```

Tudíž, světe div se, moji kolegové měli pravdu a já je prosím o odpuštění. Pochyboval jsem, ale již jsem prohlédl: Opravdu je **GraalVM Ruby** alespoň desetkrát rychlejší než standartní implementace.

3.2 JavaScript

Rychlé **Ruby** je sice hezká věc, ale kdo dnes píše v **Ruby**? Ten jazyk už je hezkých pár let za zenitem. Dnes přeci všichni píší v **JavaScript**u, že!?

Ano, **JavaScript** je v současnosti vcelku populární a tudíž jej **GraalVM** také podporuje. A to buď jako čistý jazyk bez knihoven a nebo v kombinaci s *node.js*, který poskytuje rozhraní pro přístup k operačnímu systému. Přepsat moje síto do JavaScriptu nebylo těžké. Struktura programu zůstala zachována: stále je tam generátor přirozených čísel, spojový seznam s prvočísly i smyčka, jež měří čas potřebný pro výpočet prvních sto tisíc prvočísel. Kód tedy máme a můžeme měřit. Obecně je za nejrychlejší virtuální stroj pro běh **JavaScript**u považována **V8** od Googlu, která je také součástí standardní distribuce Ubuntu v podobě systému *node.js*. Můžeme se tedy zkusit porovnat s ní:

```
$ nodejs -v
v6.14.1
$ nodejs js/sieve.js
Hundred thousand prime numbers in 108 ms
$ /graalvm-0.33/bin/js js/sieve.js
Hundred thousand prime numbers in 106 ms
```

Nechci tvrdit, že by **GraalVM** byla vždy rychlejší než **V8**, ale když se ten program správně napíše, tak může být. Určitě však není o moc pomalejší. To by nás šéf hnal – v mládí si na prázdniny odskočil do Googlu napsat *CrankShaft* kompilátor – a teď nedá pokoj, dokud nejsme alespoň stejně rychlí. Zvláště po tom, co mu *CrankShaft* v nových verzích **V8** nahradili *TurboFan*em, se z toho stala otázka cti.

GraalVM JavaScript je tedy rychlejší než **GraalVM Ruby**, ale jen o kousíček. Je tak vhodné položit si otázku: *Má cenu přepisovat celé programy z Ruby do JavaScriptu?* Ne, nemá. Stačí přepsat jen část!

3.3 Ruby a JavaScript

Tak a teď se dostáváme k tomu, v čem je **GraalVM** opravdu jedinečná. Zkusíme napsat kus programu v **Ruby**, zbytek v **JavaScript**u a propojit obě části pomocí **GraalVM** polyglotního rozhraní. Nejprve vyhodnotíme Ruby kód, který vyexportuje symbol **Natural** pro použití z ostatních jazyků:

```
class Natural
def initialize
@x = 1
end
```

```
def next
@x += 1
end
end
def create
Natural.new
end
Polyglot.export("Natural", method(:create));
```

no a nyní provedeme část napsanou v JavaScriptu. V té chybí definice generátoru přirozených čísel, kterou si proto vyzvedneme z **Ruby**:

```
var natural = Polyglot.import('Natural');
```

do **JavaScript**ové proměnné *natural* se tedy přiřadí objekt z **Ruby** a my jej nyní můžeme přirozeně používat, tak jako by to byl normální **JavaScript**ový objekt:

```
var n = this.natural.next();
```

Tak a můžeme se podívat na rychlost. **GraalVM** nabízí binárku **polyglot**, která dokáže spouštět všechny podporované jazyky:

```
$ /graalvm-0.33/bin/polyglot --file ruby+js/sieve.rb --file ruby+js/sieve.js
Hundred thousand prime numbers in 113 ms
```

A hle! Je to rychlejší než samotné **Ruby** a jen maličko pomalejší než čistý **JavaScript**. To je asi tak jediné, s čím se můžeme porovnávat, protože neexistuje žádný jiný systém, který by mohl míchat jazyky takovýmto způsobem a při tom zachovat tak vysokou rychlost běhu.

3.4 Java

GraalVM tedy zřejmě běhá dynamické jazyky jako je **JavaScript** či **Ruby** vcelku rychle (mimochodem k tomu také nabízíme experimentální verze jazyka R, Python a mnoha dalších) a navíc je lze mezi sebou bezostyšně míchat. Má vůbec cenu používat ještě nějaké jiné jazyky?

Samotný překladač **GraalVM** je napsán v **Javě**. Je tedy na místě se zeptat: Jak rychle běhá na **GraalVM Java**? Přepsat Eratosthénovo síto do souborů Natural.java, Filter.java a Primes.java je snadné. Spustit tento projekt také:

```
$ mvn -f java/pom.xml install
$ java -version
java version "1.8.0_171"
$ mvn -f java/algorithm/pom.xml exec:java
Hundred thousand primes computed in 86 ms
```

Na mém počítači se v pohodě dostaneme pod 90 ms. To znamená, že **Java** je stále ještě rychlejší než **JavaScript** & spol. A to jsme ještě neběželi na **GraalVM**! Zkusme:

```
$ JAVA_HOME=/graalvm-1.0.0-rc1/ mvn -f java/algorithm/pom.xml exec:java
Hundred thousand primes computed in 79 ms
```

Ano, je to tak. **GraalVM** dokáže být ještě rychlejší než klasická JVM se serverovým C2 překladačem. To je samo o sobě dost těžké, neboť C2 překladač je velmi výkonný, ale **GraalVM** překladač dokáže ještě více oddálit alokaci objektů a občas je nealokovat vůbec. Stačí místo klasického JDK8 použít enterprise **GraalVM** a váš **Java** program může okamžitě běžet rychleji.

3.5 Céčko

To je všechno hezké, ale není lepší ten program stejně napsat v **C**éčku? Také mne to zajímalo, a tak jsem to tedy zkusil: main.c je stejný jako ostatní, akorát po sobě musí uklízet, neboť **C** nemá garbage kolektor. Jinak je struktura toho programu stejná a výsledek také není špatný . . .

```
$ make -C c
$ ./c/sieve
Hundred thousand prime numbers in 101 ms
```

. . . 100 ms, to je dobré. To je tak na úrovni **JavaScript**u. No čekali byste to? Většina lidí si myslí, že dynamické jazyky jsou pomalé, ale v posledních deseti letech udělal vývoj virtuálních strojů pro dynamické jazyky obrovský skok kupředu a rychlost již není problém.

S **GraalVM** lze vcelku snadno vytvořit velmi rychlý interpret jakéhokoli jazyka. Jde to tak snadno, že bychom třeba mohli napsat interpret **C**éčka. *Interpret* jazyka **C**? To už zní úplně šíleně! Proč by to někdo dělal? Tak za prvé asi dokázat si, že to jde. Takže jsme to skutečně zkusili. V rámci podprojektu *Sulong* umíme interpretovat jakýkoli jazyk, který se dá přeložit pomocí *LLVM* - tedy **C**, **C++**, **Rust**, **Julia**, atd., atd. Nejprve se vygeneruje LLVM bitkód a ten se pak může interpretovat. Zkuste si:

```
$ clang -c -emit-llvm -o c/sieve.bc c/main.c
$ /graalvm-0.33/bin/lli c/sieve.bc
Hundred thousand prime numbers in 115 ms
```

Je to jen o 10–20 % pomalejší. To je vcelku slušné na interpret, že? Ale hlavní výhodou je, že tento interpret je pro **GraalVM** stejně *průhledný* jako ostatní interpretery a tudíž lze optimalizovat skrz naskrz **JavaScript**, **Ruby** a jejich **C**éčkové knihovny. Dohromady to pak celé může běžet rychleji, neboť se ušetří na přepínání z jednoho systému do druhého. Raději než úplně vyoptimalizovat **C** (což dokáže běh jednoho kola srazit na 75 ms), tak je lepší vše převést na jeden dynamický systém a optimalizovat to celé dohromady. **GraalVM** si s tím již poradí.

4 Produkční nasazení

Jednou z nejznámějších společností jež těží z revolučních vlastností **GraalVM** je *Twitter*, který takto spouští své tvítovací *Scala* mikroslužby. Díky agresivním optimalizacím je totiž **GraalVM** překladač schopen zredukovat alokace objektů a urychlit tak celkovou rychlost výpočtů. Díky méně alokovaným objektům je pak třeba třeba trávit méně času při úklidu pomocí garbage kolektoru, méně přerušovat běh uživatelského výpočtu a celkově snížit energetické náklady provozování globální komunikační služby jakou *Twitter* je. Více o tom, jak *Twitter* šetří náklady díky **GraalVM** lze nalézt v prezentaci Twitter JVM inženýra. V současnosti můžeme s čistým svědomím doporučit produkční nasazení **GraalVM** pro běh jazyků založených nad *JVM*, pro *JavaScript* (samostatně či společně s *node.js*). Jazyky jako *R*, *Ruby* či *Python* jsou zatím vhodné spíše pro experimentování.

5 Jak teď dále?

GraalVM Community Edition, jež je sestavena čistě z otevřených zdrojových kódů, lze stáhnout z GitHubu. Kromě toho nabízíme také přístup k *GraalVM Enterprise Edition*, jež poskytuje větší bezpečnost, škálovatelnost a rychlost v produkčních prostředích. Tato verze je dostupná například v prostředí *Oracle Cloudu*, případně může být stažena k vyzkoušení z Oracle Technology Network stránek. Kontaktujte prosím graalvm-enterprise_grp_ww@oracle.com pro možné komerční nasazení.

Nejnovější informace včetně vydaní nových verzí a dokumentace lze nalézt na http://www.graalvm.org. Každodenní vývoj lze sledovat v naší GitHub repozitoři, kde je také možné se k vývojářské komunitě připojit. O všeobecných novinkách tvítujeme jako @graalvm.

GraalVM je připravena na použití. To však neznamená, že není co vylepšovat. Neustále pracujeme na zúplnění implementace (rychlé již jsou dost) jazyků *Python*, *R* a *Ruby*. Není však v našich silách naimplementovat všechny jazyky a nástroje. Naštěstí je **GraalVM** otevřený ekosystém, který motivuje každého k vytváření vlastních jazyků a nástrojů. Navštivte http://www.graalvm.org a:

- postavte svůj vlastní jazyk nad **GraalVM**
- vytvořte svůj vlastní nástroj, jež bude pracovat se všemi jazyky
- začleňte **GraalVM** do své aplikace

Pojďme budovat tuto revoluční technologii pro polyglotní svět společně!

Blockchain & Lightning Network

Michal Švamberg

E-mail: svamberg@civ.zcu.cz

Abstrakt

Bitcoin, Litecoin, Ethereum a mnoho dalších jsou dnes již známy široké veřejnosti, která si je představuje jako nějaké počítačové peníze. Ale jen málo jich ví, jak vlastně fungují uvnitř. Zato existuje mnoho článků na téma jak moc elektřiny se „propálí", kam až vystřelila nebo se propadla jejich hodnota, případně který gauner použil digitální měnu pro zahlazení svých stop. Zkusme si tedy jednoduše vysvětlit, co je to ta zázračná technologie Blockchain, která se skrývá za všemi digitálními měnami současnosti.

1 Základ blockchainu

Napřed si musíme vysvětlit některé pojmy. Ten první je *složitý matematický výpočet* – tak jej popisují autoři některých článků o Bitcoinu. Vlastně složitý vůbec není, jde jen o jeho pochopení. Jedná se o *hašovací funkci*[1] jejíž základem je, že na velmi malou změnu ve vstupních datech reaguje naprosto rozdílným výstupem. Tyto funkce se používají například při přenosu dat po síti, kdy jejich výstup slouží jako kontrolní součet, aby se odhalila chyba. Drobná změna v přenášených datech se projeví jiným hashem. Pro nás důležitou vlastností hašovací funkce je, že se jedná o tzv. jednosměrnou funkci, **nelze** tedy z výsledného hashe rekonstruovat původní data.

Bitcoin používá SHA256 funkci, ale různé měny mohou použít jiný hašovací algoritmus. Právě výběr algoritmu výrazně ovlivňuje vhodnost

[1] Hašovací funkce je algoritmus, pro převod vstupních dat do (relativně) malého čísla.

těžby nových mincí na procesorech, grafických kartách nebo specializovaném hardwaru. My zůstaneme u Bitcoinu. Trik je v tom, že bitcoinová síť uzná výsledný hash jen v určitém tvaru – na začátku bude chtít spoustu nul, například: 00000000000000000048fa48b81ba9813057b6f7d90baa7b0d f5f6ec6b345b43[2].

Jediná cesta, jak vytvořit hash v požadovaném tvaru (tzv. obtížnosti) je trpělivě zkoušet měnit vstupní data v části zvané *nonce*. Jak moc je to složité si můžete vyzkoušet na následujícím zjednodušeném příkladu. Mějme vstupní data o velikosti 100 tisíc binárních nul (to jsou budoucí transakce, které budeme zajišťovat). K nim ale můžeme připojit libovolná data a zkoušet hledat hash začínající čtyřmi nulami.[3] Nejjednodušší je postupně hodnotu *nonce* zvětšovat.

```
$ time \
  for nonce in ‘seq 0 99999‘ ; do \
    ( head -c 1000 /dev/zero ; echo -n "$nonce" ) | \
    sha256sum | \
    tr ’\n’ ’ ’ ; \
    echo  " $nonce"; \
  done | \
  grep ^0000
0000c74029601f753ca39e5f9e37e687de3704b0a93d2df0bd3e8ee91a6e1d7f  -  19271
0000c43225da5b155d79956567be15fb35dd624d6816525a48277cb210b9b03a  -  71981

real    7m9,299s
user    1m50,980s
sys     1m0,820s
```

První nalezený hash je při nonce hodnotě 19271. Pokud námi upravený blok pošleme do bitcoinové sítě a budeme první, pak dostaneme na náš účet odměnu a transakční poplatky. Bitcoin ve skutečnosti provádí zdvojený SHA256, což lze zapsat jako vzorec

$$\$hash = SHA256(SHA256(\$data + \$nonce))$$

U těžebních zařízení se uvádí počet hashů za sekundu (H/s nebo jen Hs), které jsou schopni provést. Záleží pro jakou měnu je těžební zařízení určeno.[4]

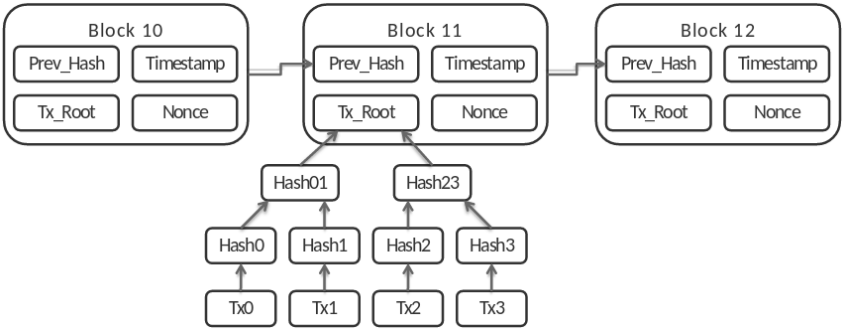
[2] Haš bloku číslo 519555, detaily na `https://blockchain.info/blocks`

[3] V reálu se hash porovnává na úrovni bitů a nikoliv znakové jako je uvedeno v příkladu.

[4] Pro Bitcoin jsou k dispozici specializovaná zařízení s ASIC čipy s hodnotami v THs (biliony hashů za sekundu).

2 Tvoříme seznam bloků

Nonce je tedy takové malé pískoviště, které můžeme libovolně upravovat, abychom jako první našli výstupní hash v požadovaném formátu. V části, kterou jsme měli prozatím označenou jako *data* se nachází **hash předchozího bloku**, čímž vzniká spojový seznam (chain), **adresa pro odměnu a poplatky**, **vybrané transakce** ze zásobníku transakcí organizované ve stromě hashů, **časová značka** a **nonce** část tak jak je to vidět na obrázku číslo 1.

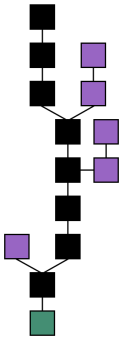


Obrázek 1 Schéma propojení jednotlivých bloků blockchainu
zdroj: Wikipedia

Tím, že součástí dat je uveden hash předchozího bloku vzniká seznam bloků – blockchain. A to je celé. Spočítáme dvojitý hash vstupních dat, zkontrolujeme tvar a pokud je platný, vyhráli jsme v loterii. Ale proč blockchain potřebuje hledat jehlu v kupce sena?

3 Bezpečnost stvrzená prací

Blockchain se často přirovnává k účetní knize. Jsou v ní zaznamenány všechny pohyby měny z účtu na účet a nesmí se stát, že si kdokoliv upraví již jednou provedenou transakci. Těžař – ten kdo hledá správnou nonce – je lákán k této práci odměnou včetně poplatků z transakcí, které zahrne do bloku. Náročnost nespočívá ve složitosti matematických operací, ale v tom, že je velmi malá šance na nalezení správné nonce, tak abychom získali požadovaný hash.



A na tom je celá bezpečnost blockchainu založena – pro podvod musíte teoreticky vlastnit alespoň 51% výpočetního výkonu, jinak nemáte šanci upravit potvrzenou transakci ve svůj prospěch. I malá změna ve vstupních datech způsobí velkou změnu na výstupu hašovací funkce a ta pak nebude odpovídat požadovanému formátu. Než se stihne spočítat nový hash, a tak zamaskovat podvod, zbytek sítě už bude pracovat na novém bloku. Nestačí přijít jen s novým podvrženým blokem, musíte přesvědčit více než polovinu uzlů sítě, že právě váš blok je ten správný a odpovídá transakcím, které byly do sítě zaslány. To je jeden z důvodů, proč za opravdu potvrzenou transakci se považuje transakce, která je alespoň 6 bloků v historii. Může se stát (a v histori bitcoinu se to už stalo), že dva těžaři vytěží ve stejný čas platný blok, každý se svou adresou pro doručení odměny. V takovém případě rozhodne většina uzlů, který z bloků budou akceptovat. Po určitou dobu, než se dohodne na akceptovaném bloku, se může síť rozdělit podobně jako na obrázku vpravo. Neakceptovaný blok, případně celá větev bloků, se zahodí. Potvrzené transakce (zahrnuté do blockchainu) jsou buď již potvrzené v hlavní větvi nebo čekají na potvrzení.

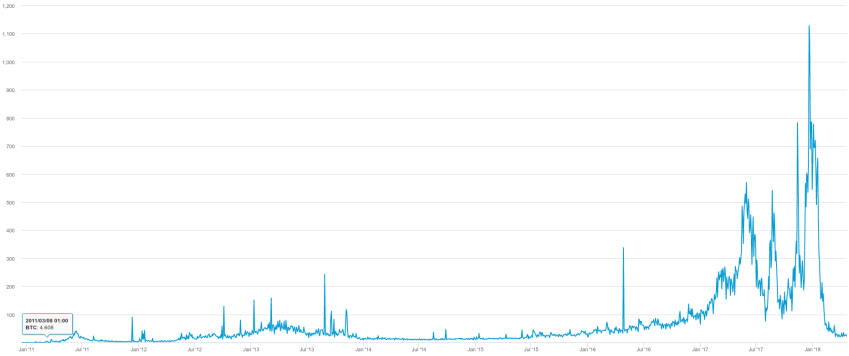
Ale co kdyby nám někdo posílal peníze a řekli jsme si, že do nového bloku zařadíme transakci, jenom si na konec hodnoty přihodíme takovou „bezvýznamnou“ nulu. I na to, že těžař může podvádět se myslelo. Vytvořený blok totiž musí akceptovat celá síť a její uzly si zkontrolují nejen správnost hashe, ale že byly zahrnuty jen takové transakce, které čekají v zásobníku (mempool). Pokud je všechno správně a většina uzlů sítě se na tom shodne, bude blok zařazen do blockchainu a jeho hash se ocitne v hlavičce následujícího bloku.

4 Poplatky za transakce

Transakce mohou obsahovat poplatek, který dostane těžař na stejnou adresu, která je uvedená pro odměnu. Výše poplatku určuje, kdy vám bude transakce schválena. Bitcoin má nastavenou obtížnost těžby tak, aby byl průměrně každých 10 minut vytěžen jeden blok, tj. 6 bloků za hodinu.[5] Bitcoinová síť obtížnost upravuje jednou za 14 dnů. Toto jsou další pa-

[5] Ve skutečnosti je rychlost těžby o cca 9% rychlejší v důsledku narůstajícího výkonu těžby.

rametry, které se prakticky u každé měny mohou lišit. Další významný parametr je velikost bloku, který je v bitcoinové síti 1MB.



Obrázek 2 Denní vývoj transakčních poplatků v bitcoinech bez odměn těžařům

zdroj: `https://blockchain.info/`

Těžař investuje prostředky do provozu zařízení, která poskytují zabezpečení proti změnám v blockchainu. Na oplátku chce ale co největší odměnu. Proto si vybírá transakce, které mají nejlepší poměr velikosti transakce vůči poplatku. Tak dokáže navýšit svoji celkovou odměnu za vytěžený blok. Uživatel sítě ovlivňuje velikostí poplatku čas, kdy bude jeho transakce zařazena do bloku a tím potvrzena. Pokud je transakce dlouho nezařazena do bloku, pak bude zrušena.[6] A na to si dejte pozor, pokud přijímáte bitcoinové platby, nepovažujte ji za provedenou do doby než bude zařazena do blockchainu a dostatečně kryta novými bloky.

Odměna těžařům s časem klesá[7] a předpokládá se, že těžaři postupně budou živi pouze z poplatků za transakce. Ty ale rozhodně nedosahují výše odměn. Další snížení odměny bude v roce 2020 při bloku číslo 630000 a to z 12,5 na 6,125 BTC. Bude velmi zajímavé jak se těžaři zachovají, protože některým se přestane těžba vyplácet a je možné, že poprvé v histori se sníží nároky na tvar výsledného hashe.

Bitcoinová síť dokáže průměrně zpracovat 7 transakcí za sekundu, to

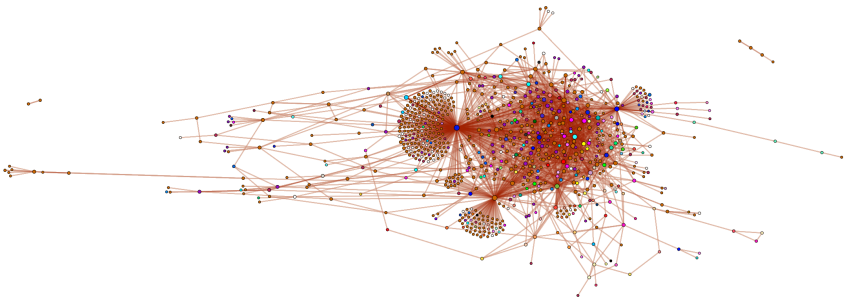
[6] Je možnost transakci nastavit příznak, že poplatek lze navýšit.

[7] Používá se tzv. halving, vždy když je vytěžena polovina zbývajících bitcoinů klesne odměna také o polovinu. Začínalo se na 50 BTC, nyní je odměna na 12,5 BTC. Poslední bitcoin (ve skutečnosti 144 satoshi) bude vytěžen okolo roku 2140.

je na dnešní poměry relativně málo. Nejsnadnějším řešením by bylo zvýšit velikost bloku, ale jen na pohled. Jenže pak je problém s narůstající velikostí celého blockchainu. Na řadu přichází možnost zrychlení, stačí snížit nároky na výsledný hash. Ale to opět způsobí nárust velikosti blockchainu. Ač to může pro mnohé znít překvapivě, blockchain není určen pro mikrotransakce. Je to příliš velký kanón na vrabce. Když budete kupovat auto, blockchain je to správné, ale na zmrzlinu zapomeňte. Nejlepší by bylo dostat malé transakce mimo blockchain, ale zároveň neztratit jeho výhody. Taková technologie již existuje, jmenuje se *Lightning Network*.

5 Chytré dohody

Lightning Network je postavena nad tzv. chytrými kontrakty (smart contracts), kterými se zakládá a spravuje platební kanál. Jsou to dohody mezi dvěma partnery, které se *vypořádají* později. Vypořádání znamená, že se zapíší zpátky do blockchainu a tím se *platební kanál* uzavře. Z těchto peer-to-peer dohod lze postavit celou síť – *Lightning Network*. Chytré dohody jsou to proto, že se umí samy vykonat podle okolností. Digitální měny jsou silně založené na privátních a veřejných klíčích. Pokud se dva uzly sítě Lightning Network dohodnou na nějaké transakci vznikne nové spojení, tzv. platební kanál, které mohou využít i ostatní uzly. Dohoda obsahuje pojistky, které zabraňují páchání podvodů druhé straně.



Obrázek 3 Znázornění testovací Lightning Network – 884 nodů, 2384 platebních kanálů.

zdroj: `https://explorer.acinq.co/`

Alespoň jedna ze stran skládá depozit, který se zapíše do blockchainu.

Tato transakce je odlišná od statních tím, že je multisignature – musí být podepsána více stranami, aby byl převod uskutečněn. Popis transakce v blockchainu není jen převod částky z účtu na účet, ale je to jednoduchý programovací jazyk. Tento depozit tak bude přístupný pouze pokud jej obě strany podepíší. Obě strany si vymění jednostranně podepsané návrhy transakcí. Nyní si mohou mezi sebou posílat peníze až do výše depozitu. Například pokud si pravidelně chci kupovat ve stánku ráno kávu, stačí do depozitu vložit hodnotu na celý měsíc. Při každém nákupu se zneplatní staré jednostranně podepsané dohody a nahradí se novými. Staré dohody nejsou zneplatněné (podpis nelze odvolat), ale v nové dohodě je napsáno, že pokud druhá strana uplatní starou dohodu, pak celý depozit náleží podváděné straně. To velmi snadno odradí od takových pokusů.[8]

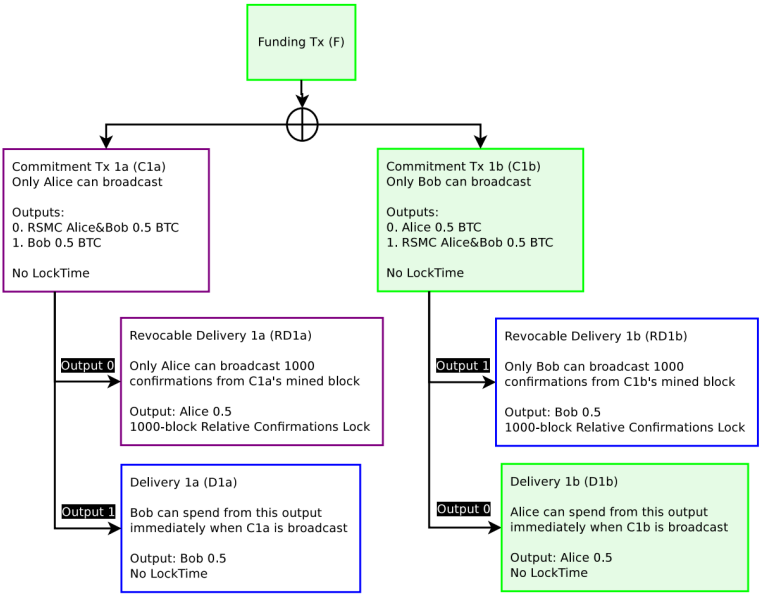
Pokud jste v pekařství kolegovi koupili housku, může vám zaplatit přes kavárnu. Kolega má také navázané spojení s kavárnou. Vy mu jen dodáte číslo svého účtu a kolega vám peníze nechá převést. Ve skutečnosti proběhnou 2 transakce, jedna mezi kolegou a kavárnou, druhá mezi kavárnou a vámi. Obě transakce budou spočívat v aktualizaci peer-to-peer dohod. Peníze tak putují po jednotlivých navázaných spojeních v blockchainu.

Mezi řádky už asi tušíte, jak kolega věděl, že má peníze poslat přes kavárnu. Podíval se do blockchainu, tam je zapsané, kdo má s kým jaké spojení navázané. Stačí si vytvořit graf a zjistit jestli existuje cesta k cílové adrese[9] a tímto směrem poslat nový návrh dohody – kavárně, že chcete provést převod na cílový učet. Z vašeho depozitu se částka přičte kavárně a ta se pokusí upravit dohodu s vámi. V dohodách jsou opět pojistky, pokud by se převody nepodařilo uskutečnit až na cílový účet.

Na obrázku číslo 4 je znázorněna struktura dohody mezi Alicí a Bobem. Alice a Bob vložily každý do společného depozitu 0,5 BTC. Bob se ale rozhodl, že chce platební kanál uzavřít, a tím získat své prostředky zpět. Bob tedy zveřejňuje dohodu C1b a D1b: Alici uvolňuje 0,5 BTC, ovšem na společném účtu ponechává zbytek. Alice může okamžitě získané prostředky využít, Bob ale musí vyčkat. Alice nyní může (ale nemusí) zveřejnit svoji část C1a a D1a, v které uvolní z depozitu Bobovu částku 0,5 BTC. Pokud Alice nebude spolupracovat, může Bob nejdříve za 1000 bloků (cca 1 týden) od vytěžení bloku s transakcí C1b+D1b zveřejnit

[8] Bitcoin funguje na základě proof-of-work (důkaz prací), kdy z práce plynou odměny. Oproti tomu Lightning Network je založen na proof-of-stake (důkaz vlastnictvím), kdy nechcete přijít o vloženou hodnotu a proto raději budete spolupracovat.

[9] Nesmíme zapomenout, že pro naši platbu můžeme využít jen kanály, které mají dostatečný depozit pro naši transakci, který nesmí být překročen.



Obrázek 4 Ukázka jednoduché chytré dohody.
zdroj: The Bitcoin Lightning Network:
Paper (PDF) DRAFT Version 0.5.9.1

dohodu RD1b, a tím se mu prostředky ze společného účtu vrátí. Jednoduché řešení s časovým zámkem chrání Boba před nečinností Alice a zároveň dává čas Alici na reakci pokud by zveřejnil Bob starou dohodu.

6 Zkusím podvádět

Výsledná dohoda se musí zapsat do blockchainu, aby se zaúčtovala a platební kanál se tak uzavřel. Nejlepší je, když se na ukončení kontraktu dohodnou obě strany. Ale může se stát, že jedna ze stran bude úmyslně či neúmyslně protestovat. V takovém případě je možné vzít poslední platnou dohodu podepsanou protistranou, tuto dohodu dopodepsat a poslat do blockchainu. V tomto případě je zde omezení, že vyrovnání nebude provedeno hned, ale až po ochranné lhůtě, aby se mohla druhá strana

bránit proti případnému podvrhu starší dohody. Protistrana tedy musí monitorovat blockchain a hlídat, že neposíláme do sítě ukončení spojení starou dohodou, kde ještě není zaúčtován poslední nákup. Druhá strana je na tom podobně, jen s tím rozdílem, že je vše opačně.

Arbitrem se tak stává blockchain, který pak transakci zpracuje podle toho, kdo ji do sítě posílá a případně trestá podvodníka, pokud na něj někdo ukáže.

7 Omezení

Lightning Network je zatím (duben 2018) vhodné používat spíše v testovací síti nebo v ostré síti, ale vždy s obnosem, který vás nebude bolet, pokud by došlo ke ztrátě nebo chybě. Projekt je to mladý a teprve nyní se dostává do fáze „klinické studie“. Avšak předpokládá se, že bude brzy součástí produkčního prostředí.

Nevýhodou by mohlo být, že musíte sledovat blockchain, abyste se mohli aktivně bránit případnému podvodu protistrany. To ale na druhou stranu posiluje celou síť, protože se tak stává distribuovanější a odolnější.

8 Závěr

Blockchain je velmi zajímavá technologie a oprávněně se o ni zajímá mnoho firem. Ačkoliv může být ekologům trnem v oku a ekonomům působit nespavost, nelze mu upřít inovativní využití již známých technologií. Lightning Network navíc přináší velké zkapacitnění blockchainu napříč digitálními měnami. Digitální měny nejsou evolucí, ale opravdovou revolucí.