VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

http://fit.vut.cz

Projektová praxe 2021/2022

Akcelerace aplikace pro potlačení DDoS útoků

Andrei Shchapaniak

Abstrakt

Jednou z hrozeb internetu byly a zůstávají **DDoS** útoky. Nástroje proti nim se vyvíjí i dnes kvůli rychlému rozvoji technologií. Jeden z takových nástrojů je vyvíjen společností CESNET. DDoS čistička je aplikace pro potlačení **DDoS** útoků a také monitorování síťového provozu. Nástroj je implementován v jazyce C s využitím open source frameworku **DPDK** (Data Plane Development Kit) pro rychlé zpracování paketů. Cílem této práce bylo rozšíření funkčnosti čističky a prostředí pro její vývoj. Abychom cíle dosáhli, musel jsem prvním krokem prozkoumat podporu časových značek síťovými kartami a možnosti uživatelských funkcí z **DPDK** pro práci s nimi. Případně bylo nutné promyšlení integrace časových značek do čističky. Následujícím krokem byla podpora spouštění čističky bez root oprávnění. Pro toto bylo nutné prozkoumat systémové soubory, ke kterým čistička přistupuje a přidání oprávnění pro ně. Posledním krokem bylo přidání dynamického pokrytí kódu. V rámci tohoto úkolu musela být přidána možnost zobrazení pokrytí kódu po každé doběhnuté pipelině na GitLabu.

Klíčová slova: DDoS útoky — code coverge — timestamp

Přiložené materiály: *N/A*

xshcha00@fit.vut.cz, *Fakulta informačních technologií, Vysoké učení technické v Brně*

1. Úvod

DoS[1] (Denial of Services) je typ útoku, jehož cílem je znefunkčnit nebo znepřistupnit napadenou službu ostatním uživatelům. **DDoS** (Distributed Denial of Service) je podtypem DoS, při kterém je pro zahlcení cílové služby využito velké množství počítačů z různých geografických lokalit. Podrobněji si o těchto typech útoků můžete přečíst na tomto webu [7].

V současném světě jsou DoS útoky populární, protože:

1. Jejich vytváření není moc složité. Vytvořit elementární útok může zkusit i běžný uživatel, protože tutoriály a nástroje jsou dostupné na internetu. Příklad můžete naleznout zde[2]. Mnohem obtížnějším úkolem je anonimizace útočníka, ale to je už jiné téma.
2. Až DoS útok začne, je velmi těžké se proti němu bránit. To znamená, že nejde o rychlé řešení problému a tím pádem firmu, na kterou byl proveden útok, čekají velké ekonomické ztráty.

Kvůli rychlému zvětšení počtu DoS útoků bylo potřeba vyvíjet moderní a spolehlivé technologie proti nim. Nejobecnější metody jsou:

- Navrhování plánu odezvy při DoS útoku.
- Zajištění vysoké úrovně zabezpečení sítě.
- Mít redundanci serveru.
- Konstantní monitorování síťového provozu.
- Omezení síťového vysílání.
- Využívat cloud k prevenci DDoS útoků.

Podrobněji o každé metodě si můžete přečíst zde [2].

Před přechodem k následující sekci bych chtěl tady nechat několik odkazů pro zájemce na téma DoS útoků:

1. Typy DDoS útoků [3].

[1] https://en.wikipedia.org/wiki/Denial-of-service_attack

[2] https://www.softwaretestinghelp.com/ddos-attack-tools/

2. Nejpopulárnější DDoS útoky [1].
3. Jeden z nejsílnějších DDoS útoků v České republice [8].

2. DDoS čistička

Jinou možností, jak se bránit proti DDoS útokům, je použití firewallu. Firewall[3] je zařízení pro zabezpečení sítě, které monitoruje síťový provoz a na základě pravidel může buď blokovat nebo povolovat datové pakety. Takový nástroj se vyvíjí v rámci jednoho z projektů výzkumné skupiny akcelerovaných síťových technologií ANT na FIT VUT. Cílem je vyvinout firewall (s názvem DDoS čistička), který bude schopen mitigovat DoS (příp. DDoS) útoky na vysokorychlostních sítích. V krátkosti čistička funguje tak, že podle mitigačních pravidel je síťový provoz zpracován a buď zahazován nebo vrácen zpět do sítě. Provoz je rozdělen do dvou skupin. Ověřený provoz je odeslán dál do sítě. Podezřelý provoz je odeslán do čističky, která tento provoz buď blokuje, nebo přesměrovává zpět do sítě 1. Osobně mě zaujala a překvapila schopnost čističky tvářit se jako zařízení v chraněné sítě. Když přijdou požadavky od neznámých klientů, je schopná vygenerovat falešnou odpověď. Tímto způsobem čistička detekuje, zda daný klient skutečně existuje a nejedná se o útok. Projekt se vyvíjí v jazyce C s využitím frameworku **DPDK**[4]. Proč právě DPDK? Na to je několik důvodů:

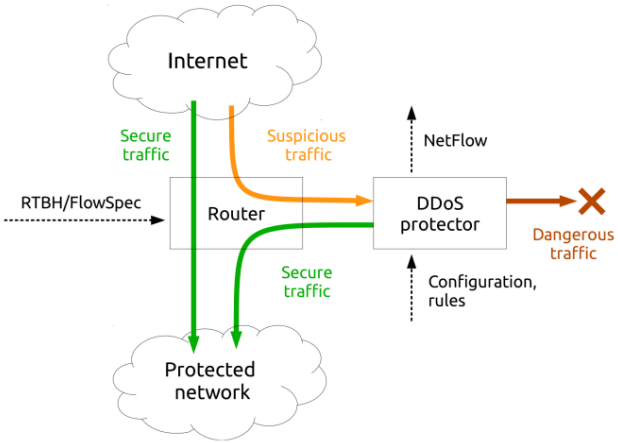
- DPDK poskytuje optimalizované funkce k urychlení zpracování paketů na vícejádrových CPU.
- DPDK poskytuje různé optimalizované komponenty, které využívají vektorové SIMD instrukční sady a umožňují tak efektivnější zpracování některých činností (LPM, ACL, obecné hash tabulky, ...).
- DPDK poskytuje rozhraní, která umožňují přesun některých činností a předzpracování přímo na síťové kartě, čímž dochází ke snížení zátěže CPU a zvýšení výkonu.

V rámci tohoto projektu jsem dostal 3 úlohy, nad kterými jsem pracoval a které byly úspěšně dokončené:

1. Podpora offloadu časových značek na našich kartách.
2. Spouštění služby čističky pod neprivilegovaným uživatelem/skupinou.
3. Přidání měření code coverage pro unit testy.

[3] https://www.forcepoint.com/cyber-edu/firewall

[4] https://www.dpdk.org/



Obrázek 1. Ilustrace čističky [5].

3. Podpora offloadu časových značek na našich kartách

3.1 Motivace

Časové značky jsou generovány pomocí vlastních hardwarových hodin síťové karty. Tuto funkci využívá zejména protokol **PTP** (Precision Time Protocol 2), což je protokol pro synchronizaci času. **SYNC** a **FOLLOW UP** zprávy jsou k určení aktuálního času. Pomocí **DELAY REQUEST/RESPONSE** se určuje zpoždění sítě. Více si můžete přečíst o časových značkách zde[5].

K čemu by se nám hodily? Jsou velmi vhodné pro rozlišování paketů. V případě, že hodnoty značek jsou dostatečně přesné a rostoucí, budeme tak mít definované pořadí paketů. Mohli bychom potom v pipelině provádět například seskupování IPv4 a IPv6 paketů, což dovolí paralelizovat některá vyhledávání. Dřív seskupování dělalo problém, protože způsobuje změny v pořadí. Pomocí časových značek ale můžeme pakety znovu jednoduše seřadit a původní pořadí tak zachovat. Doplnil bych k tomu to, že ji nastaví hardware, což umožňuje dosažení vysoké přesnosti časových značek a současně snížení zátěže CPU.

Cílem úkolu bylo odzkoušet podporu offloadu časových značek na různých typech karet, které máme k dispozici. Časová značka by potom měla být k dispozici skrz tzv. dynflags.

3.2 Řešení

Jak bylo zmíněno výše, nastavení časových značek musí zajistit síťová karta. Proto bylo nutné najít způsob, jak nastavit tuto funkci a ověřit, že časové značky se opravdu budou nastavovat do mbufů paketů. Ověřit nastavení časových značek bylo možné pomocí flagu

[5] https://docs.microsoft.com/en-us/windows/win32/iphlp/packet-timestamping

DEV_RX_OFFLOAD_TIMESTAMP[6].

Začal jsem tím, že jsem vytvořil jednoduchou aplikaci pro testovací účely. Pomocí ní jsem otestoval podporu časových značek na všech kartách, které byly k dispozici. Seznam karet je uveden v tabulce 1. Výsledek je rozepsán v sekci 3.3.1. Dále jsem pak pracoval na modulu, který do čističky integruje tuto možnost zapínání/vypínání nastavení časových značek.

3.3 Dosažený výsledek

3.3.1 Které naše karty offload podporují?

- Na Connect-X6 jsme schopní pomocí mlxconfig[7] tooly nakonfigurovat formát značek přímo na reálný čas, ale nejsme ze strany DPDK schopní ověřit, že je karta takto nakonfigurována.
- Na Connect-X5/X6 lze časovou značku na reálný čas převést voláním funkce mlx5dv_ts_to_ns() z infiniband/verbs.h, ta však potřebuje přístup k infiniband zařízení, které není skrz DPDK jednoduše dostupné.

3.3.2 Jakou přesnost značek karty podporují?

Časové značky mají přesnost v řádu nanosekund. Ještě to ale neznamená, že karta umí přiřazovat značky s takovou přesností. Pro toto budou nutné dodatečné testy, což může být rozšířením tohoto úkolu.

3.4 Budoucnost práce

1. Tento úkol byl pouze prerekvizita pro použití časových značek v modulech čističky, například v mitigátoru. Následně totiž ještě musíme zajistit, aby i v případě, že síťová karta offload časových značek nepodporuje, byly značky v mbufech vyplněny, tak aby na to komponenty pipeline mohly spoléhat. Bude nutné doimplementovat softwarové vkládání časových značek do mbufů (jde o dynamické políčko) i v případě, že karta tuto akci sama nepodporuje.
2. Pro využití offloadovaných časových značek je potom ještě nutné dořešit jejich převod na reálný čas.

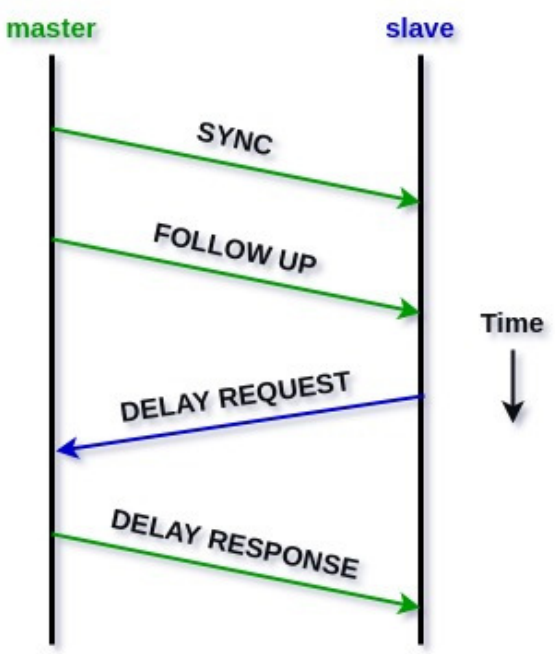
4. Spouštění služby čističky pod neprivilegovaným uživatelem/skupinou

4.1 Motivace

Cílem tohoto úkolu bylo:

[6] https://doc.dpdk.org/api/rte__ethdev_8h_source.html

[7] https://docs.mellanox.com/display/MFTv4110/Using+mlxconfig



Obrázek 2. PTP Protocol.

- Odzkoušet spouštění čističky bez roota a promyslet systém nastavování oprávnění pro různé speciální soubory (hugepages, hpet, uio, vfio, kni, atd.).
- Upravit instalaci RPM balíčků čističky, aby instalace vytvářela systémového uživatele/skupinu **dcpro**, pod kterou se budou nově spouštět služby čističky.
- Upravit služby čističky, aby přípojný bod velkých stránek a jiných zařízení měl správně nastavená oprávnění.
- Samotná instalace RPM bude muset taky zajistit, že instalovaná binárka má správně nastavené takzvané capabilities.

K čemu je vhodná možnost spouštění čističky bez root? Pro libovolnou aplikaci je vhodné, aby neměla root opravnění pro celý systém. Ovládnutí takové aplikace útočníkem by mohlo mít velmi špatné důsledky. Čistička potřebuje oprávnění pouze k některým souborům, takže nedává žádný smysl z hlediska bezpečnosti nechat jí root přístup k celému systému.

4.2 Řešení

Tato úloha byla rozdělena na několik podúkolů:

1. Najít všechny nutné soubory, ke kterým musí mít DPDK aplikace přístup.
2. Non-root spouštění jednoduchých DPDK aplikací.
3. Najít soubory, ke kterým čistička potřebuje přístup.

4. Rozšířit možnosti čističky pro spouštění bez root.

Seznam souborů jsem našel na oficiální stránce DPDK [4], kde také je popsán postup pro spouštění aplikace bez root. Pro zkušební účely jsem použil aplikaci z předchozí úlohy. Až jsem mohl spouštět DPDK aplikace bez root oprávnění, posunul jsem se ke zkoumání čističky a právě ke zdrojům, ke kterým přistupuje. Bylo zjištěno, že navíc potřebuje přístup ke zdrojům: `Netlink`, `TAP` a `KNI`. Pak jsem nalezl vhodné řešení na stránkách dokumentace Mellanox [6] s nastavením opravnění pro řadu vyžadovaných souborů. Bylo potřeba nastavit zejména správné capatibilities. Přečíst o nich si můžete v tabulce 2 nebo jsou podrobněji popsané zde[8]. Nastavení těchto capatibilities zajišťuje RPM balíček. Díky nim čistička dostává přístup k nutným souborům a je možné ji spouštět bez root.

4.3 Dosažený výsledek

1. Rozšíření připojování velkých stránek. Uživateli je poskutnuta možnost zvolit si jméno uživatele a skupiny, pro které budou umožněny manipulace s velkými stránkami čističkou.
2. Byl přidán volitelný parametr pro místo alokace velkých stránek.
3. Zvyšená oprávnění mají pouze speciální soubory, ke kterým přistupuje čistička.
4. RPM balíček zajišťuje instalaci nutných komponent a jejich nastavení.

5. Přidání měření code coverage pro unit testy

5.1 Motivace

Měření pokrytí kódu (Code coverage) je procento kódu, které je pokryto automatickými testy. Tímto můžeme jednoduše určit, které příkazy v těle kódu byly provedené prostřednictvím testovacího běhu a které ne.

Nástroj **gcov**[9] umožňuje provedení měření pokrytí kódu pro projekty v jazycích C/C++. Pro funkčnost vyžaduje instalaci a přidání některých parametrů pro překlad. Více o jeho možnostech a použití si lze přečíst zde[10].

[8] https://man7.org/linux/man-pages/man7/capabilities.7.html

[9] https://man7.org/linux/man-pages/man1/gcov.1.html

[10] https://www.linuxtoday.com/blog/analyzing-code-coverage-with-gcov/

Při překladu čističky je možné zapnout gcov, který umožní změřit pokrytí kódu, např. při spouštění unit testů nebo pytestů.

Pro co by bylo vhodné mít automatické měření kódu? Když programátoři píší testy pro aplikaci, kterou vyvíjí, nemohou si být jistí, že testy pokrývají všechny možné případy. Integrace automatického pokrytí testů do pipeliny na GitLabu by mohla pomoct efektivněji kontrolovat přidávání testů pro čističku. Také bychom díky ní mohli zjistit aktuální stav pokrytí testů.

5.2 Řešení

Začal jsem se zkoušením základních možností nástroje gcov. Vytvořil jsem jednoduchou aplikaci v jazyce C. Při překladu se vytvoří GCNO soubory. Po spuštění aplikace jsou vytvořeny GCDA soubory, které obsahují informace o vykonaných částech kódu. Gcov ale sám neumí vytvořit výstup čitelný pro vývojáře, bylo proto potřeba najít nějaký jiný vhodný nástroj, který by takovou možnost poskytl. Existuje několik způsobů, jak získat výslednou HTML stránku se statistikou. Zvolili jsme nástroj **gcovr**[11]. Jedná se o nástavbu nad základním nástrojem gcov, která poskytuje stejné možnosti a navíc umožňuje generování výstupů v dalších různým formátech, např. HTML, JSON nebo XML.

Nejprve je pomocí gcovr vygenerován jediný JSON soubor, který agreguje výsledky ze všech vygenerovaných GCDA souborů, pro daný běh testů. Jednotlivé JSON soubory jsou následně pomocí volání gcovr s použitím jiných argumentů sloučeny. Výsledkem je HTML stránka, kterou lze zobrazit v libovolném webovém prohlížeči.

5.3 Dosažený výsledek

Výsledky tohoto úkolu lze vidět na obrázcích 4 a 3. HTML stránka s úplným pokrytím kódu je dostupná jako artefakt po ukončení pipeliny na GitLabu.

5.4 Budoucnost práce

1. Rozšíření pokrytí kódu pro pytesty. Úplné pokrytí kódu i v pytestech zajistí, že všechny části kódu byly v nějaké situaci alespoň jednou vykonány a tedy řádně testovány, čímž se významně snižuje pravděpodobnost výskytu chyby.
2. Rozšířená statistika kódu. Jedná se například o přidání labelů, jaký test kterou část kódu právě testoval. Vyhledávání nutné testovací informací podle regulárních výrazů atd..

[11] https://gcovr.com/en/stable/

Types of cards	
Mellanox	Intel
ConnectX-4 100 GbE	X552 10 GbE SFP+
ConnectX-5 100 GbE	82599ES 10 GbE SFI/SFP+
ConnectX-5 10/25 GbE	X710 10 GbE SFP+
ConnectX-6 Dx 100 GbE	E810CQDA2BLK

Tabulka 1. Seznam karet pro DPDK

Capatibilities	
Name of capatibility	Meaning
CAP_SYS_ADMIN	Perform various privileged filesystem Modify allow/deny rules for device control groups Perform administrative operations on many device drivers
CAP_NET_ADMIN	Modify routing tables Set promiscuous mode Enabling multicasting
CAP_NET_RAW	Use RAW and PACKET sockets
CAP_IPC_LOCK	Allocate memory using huge pages

Tabulka 2. Linux capatibilities

6. Závěr

Podle zadání jsem se seznámil s problematikou DoS útoků, s jejich negativními vlivy na služby a metodami, jak se proti nim bránit. Také jsem pracoval nad projektem "DDoS čistička" společnosti CESNET. V dálším kroku bylo mnou splněno několik zajímavých a hlavně vhodných úkolů pro zlepšení čističky a prostředí pro její vývoj. Čistička byla rozsířená o podporu zapnutí/vypnutí časových značek, spouštění bez root oprávnění a měření pokrytí kódu pro unit testy.

7. Poděkování

Rád bych poděkoval svému vedoucímu, Ing. Janu Kučerovi, a Ing. Lukáši Hutákovi za pomoc a konzultace.

Literatura

[1] ADMIN GLOBALDOTS. *15 Most Dangerous DDoS Attacks That Ever Happened*. June 2016. Dostupné z: https://www.globaldots.com/resources/blog/15-most-dangerous-\ddos-attacks-that-ever-happened/.

[2] ANDREJA VELIMIROVIC. *How to Prevent DDoS Attacks: 7 Tried-and-Tested Methods*. December 2021. Dostupné z: https://phoenixnap.com/blog/prevent-ddos-attacks.

[3] IMPERVA. *Common DDoS attacks types*. 2021. Dostupné z: https://www.imperva.com/learn/ddos/ddos-attacks/.

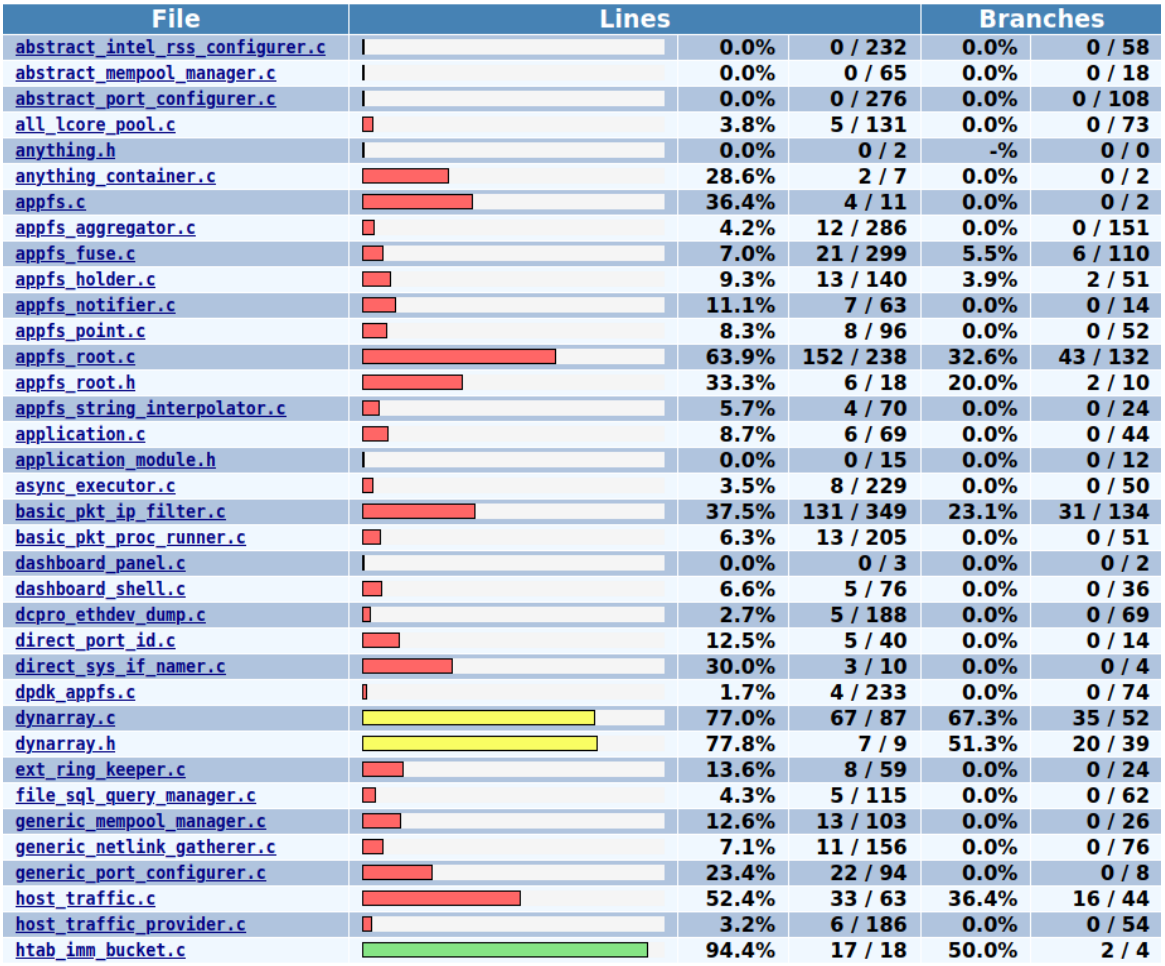
[4] INTEL CORPORATION. *Running DPDK Applications Without Root Privileges*. 2021. Dostupné z: http://doc.dpdk.org/guides/linux_gsg/enable_func.html.

[5] MARTIN ŽÁDNÍK. *DDoS Protector aneb čistička*. 2017. Dostupné z: https://www.cesnet.cz/wp-content/uploads/2017/02/\ddos-protector.pdf.

[6] MELLANOX TECHNOLOGIES. *MLX5 poll mode driver - run as non-root*. 2021. Dostupné z: https://doc.dpdk.org/guides/nics/mlx5.html.

[7] TIM KEARY. *Dos vs DDoS Attacks: The Differences and How To Prevent Them*. July 2020. Dostupné z: https://www.comparitech.com/net-admin/dos-vs-ddos-\attacks-differences-prevention/.

[8] WEDOS. *Jak probíhal zřejmě nejsilnější DDoS útok v Česku*. April 2021. Dostupné z: https://blog.wedos.cz/jak-probihal-zrejme-nejsilnejsi-\ddos-utok-v-cesku.



File	Lines		Branches	
abstract_intel_rss_configurer.c	0.0%	0 / 232	0.0%	0 / 58
abstract_mempool_manager.c	0.0%	0 / 65	0.0%	0 / 18
abstract_port_configurer.c	0.0%	0 / 276	0.0%	0 / 108
all_lcore_pool.c	3.8%	5 / 131	0.0%	0 / 73
anything.h	0.0%	0 / 2	-%	0 / 0
anything_container.c	28.6%	2 / 7	0.0%	0 / 2
appfs.c	36.4%	4 / 11	0.0%	0 / 2
appfs_aggregator.c	4.2%	12 / 286	0.0%	0 / 151
appfs_fuse.c	7.0%	21 / 299	5.5%	6 / 110
appfs_holder.c	9.3%	13 / 140	3.9%	2 / 51
appfs_notifier.c	11.1%	7 / 63	0.0%	0 / 14
appfs_point.c	8.3%	8 / 96	0.0%	0 / 52
appfs_root.c	63.9%	152 / 238	32.6%	43 / 132
appfs_root.h	33.3%	6 / 18	20.0%	2 / 10
appfs_string_interpolator.c	5.7%	4 / 70	0.0%	0 / 24
application.c	8.7%	6 / 69	0.0%	0 / 44
application_module.h	0.0%	0 / 15	0.0%	0 / 12
async_executor.c	3.5%	8 / 229	0.0%	0 / 50
basic_pkt_ip_filter.c	37.5%	131 / 349	23.1%	31 / 134
basic_pkt_proc_runner.c	6.3%	13 / 205	0.0%	0 / 51
dashboard_panel.c	0.0%	0 / 3	0.0%	0 / 2
dashboard_shell.c	6.6%	5 / 76	0.0%	0 / 36
dcpro_ethdev_dump.c	2.7%	5 / 188	0.0%	0 / 69
direct_port_id.c	12.5%	5 / 40	0.0%	0 / 14
direct_sys_if_namer.c	30.0%	3 / 10	0.0%	0 / 4
dpdk_appfs.c	1.7%	4 / 233	0.0%	0 / 74
dynarray.c	77.0%	67 / 87	67.3%	35 / 52
dynarray.h	77.8%	7 / 9	51.3%	20 / 39
ext_ring_keeper.c	13.6%	8 / 59	0.0%	0 / 24
file_sql_query_manager.c	4.3%	5 / 115	0.0%	0 / 62
generic_mempool_manager.c	12.6%	13 / 103	0.0%	0 / 26
generic_netlink_gatherer.c	7.1%	11 / 156	0.0%	0 / 76
generic_port_configurer.c	23.4%	22 / 94	0.0%	0 / 8
host_traffic.c	52.4%	33 / 63	36.4%	16 / 44
host_traffic_provider.c	3.2%	6 / 186	0.0%	0 / 54
htab_imm_bucket.c	94.4%	17 / 18	50.0%	2 / 4

Obrázek 3. Pokrytí kódu

V tomto formátu dostáváme pokrytí kódu. Vlevo jsou všechny soubory, které byly otestovány. Následuje pokrytí v procentech a informace o pokrytí jednotlivých větví. Soubory vlevo je možné otevřít a podívat se, co testy pokrývají. Výsledek je vidět na obrázku 4.

Legenda barev:

červená: 0 - 75%, kód byl pokryt špatně

žlutá: 75 - 90%, kód byl pokryt dostatečně

zelená: 90 - 100%, kód byl pokryt dobře

```
43          8  static inline uint16_t rte_pktmbuf_alloc_burst(
44                     struct rte_mempool *pool,
45                     struct rte_mbuf **mbufs,
46                     uint16_t count)
47             {
48          8          uint16_t idx = 0;
49          8          int rc;
50
51   ▸ 1/2  8          rc = rte_mempool_get_bulk(pool, (void **) mbufs, count);
52   ▸ 1/2  8          if (unlikely(rc)) {
53                             struct rte_mbuf *m;
54
55          ✗                  for(idx = 0; idx < count; idx++) {
56          ✗                          rc = rte_mempool_get(pool, (void **) &m);
57          ✗                          if (unlikely(rc))
58                                             break;
59
60          ✗                          rte_pktmbuf_reset(m);
61          ✗                          mbufs[idx] = m;
62                             }
63
64          ✗                  return idx;
65                 }
```

Obrázek 4. Rozklinutý soubor

Čísla v levém sloupci ukazují, kolikrát řádek byl spouštěn. Například na řádku `52` je pěkně vidět, že podmínka `unlikely(rc)` byla otestována 8krát, ale vždy byla nepravdivá, kvůli čemu nedošlo k testování jejího těla.

Legenda barev:

zelená: kód, který byl pokryt alespoň jedním testem

červená: kód, který nebyl pokryt žádným testem