دانشکده مهندسی کامپیوتر

بررسی روش‌های تولید خودکار داده‌های آزمون برای استفاده در فازرهای مبتنی بر قالب فایل

گزارش سمینار کارشناسی ارشد

در رشته مهندسی کامپیوتر – گرایش نرم‌افزار

نام دانشجو:

مرتضی ذاکری نصرآبادی

استاد راهنما:

دکتر سعید پارسا

آبان‌ماه ۱۳۹۶


بسم الله الرحمن الرحيم

چکیده

فنون آزمون فازی نرم‌افزار برای شناسایی خطاها و آسیب‌پذیری‌های نرم‌افزار نیاز به داده‌های خاص آزمون دارند. مشکل اساسی پیچیده بودن ساختار برنامه‌های دنیای واقعی و ورودی‌های آن و در نتیجه تعداد بیش از حد مسیرهای اجرایی در برنامه است. از یکسو اگر همه مسیرهای اجرایی پوشش داده نشوند، آسیب‌پذیری‌هایی که در مسیرهای پوشش داده نشده قرار دارند، قابل شناسایی نخواهند بود؛ از سوی دیگر سنجش کلیه مسیرهای اجرایی برنامه فرایندی زمان‌بر و پرهزینه است. افزون بر این، بررسی‌ها نشان می‌دهد بسیاری از موردهای آزمون تولید شده مسیرهای یکسان و سطحی را می‌پیمایند و در کل آزمون‌های فازی معمول پوشش مسیر ضعیفی دارند. آزمون‌های فازی که پوشش مسیر به‌نسبت بهتری دارند نیز معمولا وابسته به برنامه هدف بوده و مقیاس‌پذیر نیستند.

برای حل مسائل گفته شده نیازمند بررسی روش‌های مختلف تولید داده آزمون به‌ویژه داده‌های با ساختار پیچیده هستیم. بسیاری از برنامه‌های کاربردی، مثل مرورگرهای وب، فایل‌های با قالب مشخص را به عنوان ورودی می‌پذیرند. تولید فایل‌های خوش ساختار امکان عبور از پویشگر اولیه، نفوذ به عمق برنامه و پوشش مسیرهای جدید را میسر می‌سازد. در این صورت، یافتن خطاها و آسیب‌پذیری‌ها محتمل‌تر خواهد شد.

روش‌های تصادفی محض و مبتنی بر جهش برای ساختارهای پیچیده اصلا مناسب نیستند. روش‌های مبتنی بر تولید پوشش کد بالایی فراهم می‌کنند، اما زمان‌بر، پرهزینه و نیازمند داشتن مشخصه‌های قالبِ فایل هستند. روش‌های اکتشافی و آگاه از برنامه که در حاضر رایج‌اند، برخی مشکلات را حل کرده، اما معمولا کُند هستند. در این پژوهش روش‌های فوق معرفی و مقایسه شده‌اند. همچنین روش مبتنی بر یادگیری ژرف و مدل‌های مولد RNN بررسی شده که بسیار جالب بوده و راه حل مناسبی برای یادگیری ساختارهای فایل پیچیده است. در نهایت می‌توان گفت راه‌حل‌های مبتنی بر هوش‌مصنوعی به‌خصوص یادگیری ماشینی لبه پژوهش در قلمرو فازرهای مبتنی بر قالب فایل هستند و افق روشنی در مقابله با چالش‌های موجود و پیش‌رو دارند.

واژه‌های کلیدی: آزمون فازی، داده آزمون، پوشش کد، یادگیری ژرف، شبکه‌های عصبی.

فهرست مطالب

عـــنوان **صـــفحه**

فهرست شکل‌ها

عــنوان **صــفحه**

فهرست جدول‌ها

عــنوان **صــفحه**

جدول واژگان و نمادهای اختصاری

مفهوم واژه (نشانه) اختصاری	واژه (نشانه) اختصاری
Denial of Service	**DoS**
Long-Short Term Memory	**LSTM**
Rectified Linear Unit	**ReLU**
Recurrent Neural Network	**RNN**
Software Under Test	**SUT**
تابع با متغیر مستقل x و پارامتر θ	$f(x,\theta)$
پارامتر اسکالر a	a
بردار a	a
ماتریس A	A

فصل ۱: مقدمه

من می‌گویم، امنیت، بالاترین اولویت ماست. زیرا برای همه چیزهای هیجان‌انگیزی که شما قادر به انجام دادن آن با کامپیوترها هستید – سازمان‌دهی زندگی‌تان، در ارتباط ماندن با دیگران، خلاق بودن – اگر ما مسائل امنیتی را حل نکنیم، مردم از همه این‌ها عقب خواهند ماند.

بیل گیتس

۱-۱- پیش‌زمینه

آزمون نرم‌افزار بخش مهمی از فرایند توسعه و ساخت یک سیستم نرم‌افزاری را تشکیل می‌دهد و فنون مختلفی برای آن وجود دارد. *آزمون فازی*[۱] یا *فازینگ*[۲]، یک فن مؤثر آزمون نرم‌افزار به‌منظور کشف زود هنگام خطا[۳]ها، قبل از تبدیل شدن آنها به آسیب‌پذیری[۴] است. آزمون فازی برای آزمودن و نمایان‌سازی خرابی‌ها در نرم‌افزارهایی که ورودی‌هایی با ساختار(ها)ی پیچیده می‌پذیرند؛ از قبیل مرورگرهای وب، ویرایشگرهای متن، پخش‌کننده‌های چندرسانه‌ای و غیره، بسیار مناسب ظاهر شده است [1] و [2]. در این فن ورودی‌هایی خاص توسط یک برنامه دیگر، یعنی با روش خودکار، تولید شده و به نرم‌افزار تحت آزمون (SUT)[۵] تزریق می‌شود. برنامه در عین حال، به امید یافتن خطا بر اثر پردازش ورودی تزریق شده، پایش می‌شود. ورودی که به برنامه داده می‌شود نقش *داده (مورد) آزمون*[۶] را داشته و عامل اصلی نمایان‌سازی خطا(های) احتمالی موجود در برنامه با بردن آن به یک حالت خرابی[۷] است. به‌همین علت مهم‌ترین قسمت در فرایند آزمون فازی را می‌توان تولید خودکار داده‌های آزمون دانست، به‌نحوی که بیشترین خطاها و ایرادات شناسایی گردند.

۱-۲- شرح مسئله

راهکارهای آزمون فازی نرم‌افزار برای شناسایی خطاها و آسیب‌پذیری‌ها [3]، [4]، [5] و [6] نیاز به داده‌های خاص آزمون دارند [2] و [7]. مشکل اساسی در اینجا پیچیده بودن ساختار ورودی، SUT و در نتیجه تعداد بیش از حد مسیرهای اجرایی در کد برنامه است. از یکسو اگر مسیرهای اجرایی پوشش داده نشوند، نمی‌توان انتظار داشت که برخی آسیب‌پذیری‌ها مشخص گردند؛ از

---

۱ fuzz testing
۲ fuzzing
۳ fault
۴ vulnerability
۵ software under test
۶ test data (case)
۷ failure

سوی دیگر بررسی کلیه مسیرهای اجرایی برنامه می‌تواند زمان‌بر و پرهزینه باشد [8] و [9]. علاوه‌بر این بررسی‌ها نشان می‌دهد بسیاری از موردهای آزمون تولید شده مسیرهای یکسان و سطحی را می‌پیمایند [9] و در کل آزمون‌های فازی معمول پوشش مسیر ضعیفی دارند [10]. بیشتر موردهای آزمون ساخته شده به‌صورت تصادفی، از لحاظ ساختاری کاملا نامعتبر هستند و در همان مراحل اولیه بررسی توسط پویشگر[۱] فایل برنامه هدف رد می‌شوند [7] و [9]، در نتیجه قادر به نفوذ به عمق برنامه و کشف مسیرهای جدید نیستند. در واقع این نوع ورودی‌ها به نوعی هدر رفته محسوب می‌شوند. اگر در جهت افزایش پوشش کد فازینگ نیز تلاش شود معمولا از مقیاس‌پذیری[۲] ابزار توسعه داده شده برای آزمون فازی کم می‌شود [9].

شـــهود اولیه. برای درک بهتر مسئله‌ی ساختار پیچیده کد و ممانعت آن از پوشش مناسب در آزمون فازی قطعه کد شکل (۱-۱) به زبان C را نظر می‌گیریم. این کد یک فایل را از ورودی خوانده و براساس بایت‌های مشخصی در آدرس نسبی[۳] ثابت ورودی مسیرهای معینی را اجرا می‌کند. چندین نکته قابل توجه در قطعه کد شکل (۱-۱) وجود دارد [9]:

۱. **بایت‌های جادویی**[۴]: بایت دوم و بایت اول ابتدا برای اعتبارسنجی ورودی با مقادیر ثابتی مقایسه می‌شوند. اگر نتیجه این مقایسه صحیح نباشد؛ ورودی فوراً رد می‌شود. در این مثال ابتدا آدرسی نسبی ۱ با مقدار 0xEF و سپس آدرس نسبی ۰ با مقدار 0xFD مقایسه می‌گردد. بایت‌های جادویی در قالب‌های فایل بسیاری وجود دارند. از جمله قالب فایل jpeg که در ابزار djpeg با همین روش اعتبارسنجی می‌شود.
۲. **اجرای عمیق‌تر:** به منظور رسیدن به مسیر اجرایی عمیق‌تر در خط ۱۵ قطعه‌کد نیز یک مقایسه برمبنای آدرس‌های نسبی انجام می‌شود. ممکن است آدرس نسبی به‌کار رفته در این مقایسه‌ها نیز از ورودی خوانده شده و بنابراین در هر بار اجرا متفاوت باشند. این امر برخلاف مورد بایت‌های جادویی است که در آدرس نسبی ثابتی دارند.

---

[۱] parser
[۲] scalability
[۳] offset
[۴] magic bytes

```
1 int main(int argc, char **argv){
2     unsigned char buf[1000];
3     int i, fd, size, val;
4     if ((fd = open(argv[1], O_RDONLY)) == -1)
5         exit(0);
6     fstat(fd, &s);
7     size = s.st_size;
8     if (size > 1000)
9         return -1;
10    read(fd, buf, size);
11    if (buf[1] == 0xEF && buf[0] == 0xFD) // notice the order of CMPs
12        printf("Magic bytes matched!\n");
13    else
14        EXIT_ERROR("Invalid file\n");
15    if (buf[10] == '%' && buf[11] == '@') {
16        printf("2nd stop: on the way...\n");
17        if (strncmp(&buf[15], "MAZE", 4) == 0) // nested IF
18            ... some bug here ...
19        else {
20            printf("you just missed me...\n");
21            ... some other task ...
22            close(fd); return 0;
23        }
24    } else {
25        ERROR("Invalid bytes");
26        ... some other task ...
27        close(fd); return 0;
28    }
29    close(fd); return 0;
30 }
```

شکل (۱-۱) یک قطعه‌کد با ساختار تودرتو که چالش‌های پیچیدگی برنامه و پوشش کد در آزمون فازی قالب فایل را نشان می‌دهد [9].

۳. **نشان‌گرها**[۱]: به‌منظور رسیدن به کد دارای خطا در خط ۱۸ باید شرط موجود در خط ۱۷ ارضا شود. این مقایسه با یک توالی از نشانه‌ها انجام می‌شود که آدرس نسبی شروع آن لزوماً ثابت نیست؛ البته در این مثال ثابت نشان داده شده است. در قالب‌های فایلهایی مانند jpeg، png و gif این قبیل نشان‌گرها دیده می‌شود.

۴. **شرط‌های تودرتو**[۲]: در زمینه پوشش کد هر مسیر اجرایی مهم است. هرچند رسیدن به برخی مسیرها ممکن است مشکل‌تر باشد یا حتی امکان‌پذیر نباشد [11]. در این مثال برای رسیدن به خط ۱۸ کد بایستی همه شرط موجود در خط ۱۷ برقرار باشد که به‌نوبه خود نیاز

---

[۱] marker

[۲] nested conditions

هست تا شرط موجود در خط ۱۵ نیز برقرار شد و به همین ترتیب. لذا مورد آزمون تولیدی باید تا حد زیادی معتبر باشد تا بتواند به عمق مدنظر دسترسی پیدا کند.

برای حل مسائل گفته شده نیازمند بررسی روش‌های مختلف تولید داده آزمون به‌ویژه داده‌های با ساختار پیچیده هستیم. روش‌های متعددی برای تولید داده آزمون وجود دارد که هر یک از آنها مزایا و معایب مربوط به خود را دارند. ما در گام اول به دنبال یافتن روش‌هایی برای تولید داده‌های آزمون با امکان پوشش کد بیشتر و نفوذ به مسیرهای عمیق برنامه هستیم. در گام بعد هدف افزایش تعداد داده‌های آزمون برای آشکارسازی اشکال‌های برنامه مورد نظر است [9] و [12].

۱-۳- حوزه سمینار

آزمون فازی قالب فایل، روش‌های تولید داده آزمون در آن، آسیب‌پذیری‌های قابل شناسایی توسط آزمون فازی، ساختار فایل‌های دودویی مورد استفاده برنامه‌ها و نیز مبانی یادگیری ماشینی ژرف و بالاخره کارهای انجام شده در زمینه و استخراج ساختار فایل‌ها و تولید خودکار داده‌های آزمون، از مباحث تحت پوشش این سمینار هستند. در زیر مروری مختصر بر مباحث نام برده خواهیم داشت.

آزمون فازی. روش‌ها و فنون مختلفی برای آزمون امنیت نرم‌افزار و کشف آسیب‌پذیری‌ها وجود دارد. آزمون فازی که نخستین بار در [3] معرفی شد، یک روش پویا، مبتنی بر تولید داده‌های آزمون تصادفی، جعبه‌سیاه[۱] و خودکار برای آزمون نرم‌افزار است [13]. البته این روش به رویکردهای جعبه‌سفید[۲] و جعبه‌خاکستری[۳] تعمیم داده شده است [14]. در این روش ورودی‌های ناخواسته‌ای (معمولا به‌صورت تصادفی) تولید و در تکرارهای متوالی به نرم‌افزار تحت آزمون تزریق می‌گردد و رفتار نرم‌افزار در مواجهه با این ورودی‌ها مشاهده می‌شود. چنان‌چه خطایی رخ دهد،

---

[۱] black-box

[۲] white-box

[۳] gray-box

تصویر حافظه برنامه برای تحلیل محل و علت وقوع خطا روبرداری[۱] می‌شود. همچنین داده ورودی که منجربه ایجاد این خطا شده است نیز به‌منظور تولید مجدد خطا ذخیره خواهد شد. مزیت این روش در نگاه اول سادگی در فهم و راحتی در به‌کارگیری آن حتی برای برنامه‌های پیچیده است. مزیت مهم بعدی عدم نیاز به کد منبع است. امکان خودکارسازی همه یا بخش‌هایی از فرایند ذکر شده نیز از مزایای آزمون فازی به شمار می‌آید [13].

فازرها. برای خودکار سازی فرایند آزمون فازی ابزاری موسوم به ***فازر***[۲] توسعه داده می‌شود. یک فازر در کامل‌ترین حالت، دارای سه پیمانه‌ی تولیدکننده مورد آزمون، تزریق‌کننده مورد آزمون و ابزار پایش وضعیت SUT، است [2] و [13]. البته تفکیک پیمانه‌ها به این قسم همواره برقرار نیست و برخی فازرها ممکن است همه‌ی پیمانه‌ها را نداشته باشند؛ مثلا برای پایش ممکن است از امکانات سیستم‌عاملی که SUT روی آن مقیم است، استفاده شود. معمولا دو روش مرسوم برای تولید موردهای آزمون توسط فازرها وجود دارد: روش‌های مبتنی جهش که به‌صورت تصادفی ورودی‌هایی را جهش داده و روش‌های مبتنی بر تولید که بر اساس مجموعه‌ای از قوانین ورودی را تولید می‌نمایند [15].

فازرهای مبتنی بر قالب فایل. فازرها برای اهداف گوناگون استفاده می‌شوند. یک دسته مهم از آنها با هدف آزمودن برنامه‌هایی که ورودی آنها فایل (با قالب متنی یا دودویی) هستند، طراحی می‌شوند. به‌دلیل اینکه برنامه‌های بسیاری وجود دارند که ورودی آنها فایل‌های دارای ساختار مشخص و پیچیده است، این فازرها اهمیت بسیاری در کشف دسته وسیعی از اشکال‌ها و آسیب‌پذیری‌های ***روزِ صفر***[۳] دارند [9]، [12].

آسیب‌پذیری‌ها. بیشتر آسیب‌پذیری‌هایی که با فازرها قابل تشخیص هستند، نقص‌های حافظه نظیر سرریز میانگیر، سرریز پشته و هرم، دسترسی بعد آزادسازی و غیره بوده که در صورت باقی ماندن در نرم‌افزار به‌راحتی قابل ***بهره‌برداری***[۴] و سوءِ استفاده خواهند بود [1]، [3] و [14].

---

[۱] dump

[۲] fuzzer

[۳] zero-day

[۴] exploit

پیدا کردن آسیب‌پذیری‌ها در درجه اول منوط به پوشش عمیق و پیمایش مسیرهای اجرایی مختلف درون کد است که در سایر فنون آزمون نرم‌افزار نیز مطرح هستند.

معیارهای پوشش. معیارهای پوشش[۱] روش‌هایی برای نحوه آزمون (تولید داده) و همچنین اندازه‌گیری میزان آزمون انجام شده هستند. یکی از مهمترین معیارهای پوشش، پوشش کد است. روش‌های پوشش کد خود به سه دسته پوشش جملات[۲]، پوشش شاخه[۳]ها و بلأخره پوشش مسیرهای اجرایی[۴] برنامه تقسیم می‌گردد [16]. روش جدیدتر و کامل‌تر پوشش دامنه است که در آزمایشگاه مهندسی معکوس دانشکده مهندسی کامپیوتر دانشگاه علم و صنعت در حال توسعه است و در مقاله [17] نیز بحث شده است.

یادگیری ماشین. یادگیری ماشین بدون حل مستقیم و قاعده‌مند مسئله در تلاش است تا با استفاده از داده‌ها و تجربه‌های موجود یک حل تقریبی بهینه از مسئله را ارایه دهد [18]. ساختار فایل‌های پیچیده می‌تواند با استفاده از این روش درک شود. بدین ترتیب نیاز به روش‌های سنتی مبتنی برگرامر و قواعد زبانی نیست و بیشتر هزینه صرف محاسبات ماشینی می‌شود تا تهیه دستی گرامرها و قواعد مورد نیاز.

در این پژوهش به‌دنبال مطالعه روش‌های مختلف تولید داده‌های ورودی برای آزمون فازی هستیم که در عین حفظ مقیاس‌پذیری استفاده فازر، بتوانند مسیرهای اجرایی عمیق و مختلف کد را طی کنند و بیشترین خطای ممکن را نمایان سازند. برای این منظور، برخی رویکردهای جدید در تولید داده‌های آزمون را بررسی می‌کنیم. خواهیم دید که روش‌های فازی اکتشافی [19] و آگاه از برنامه [9] و نیز روش‌هایی که با استفاده از یادگیری ماشین ساختار فایل‌های ورودی را یاد می‌گیرند و سپس ورودی‌ها را از روی یک مدل مولد[۵] تولید می‌کنند [8] چطور توانسته‌اند، تا اندازه خوبی به این مهم دست یابند و چه چالش‌هایی کماکان باقی مانده است.

---

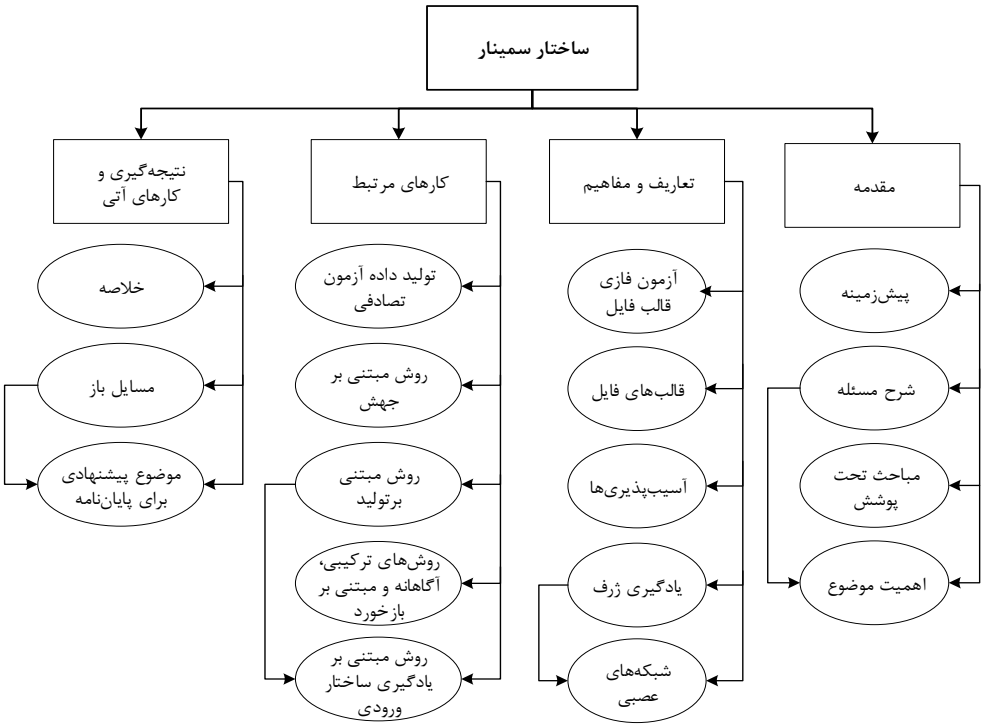
۱ coverage criteria
۲ statement coverage
۳ branch coverage
۴ paths coverage
۵ generative model

۱-۴- اهمیت موضوع

کشف خطاها و آسیب‌پذیری‌ها چه در آن دسته از نرم‌افزارهایی که روانه بازار شده‌اند و چه در آنهایی که در مرحله توسعه هستند، بسیار حائز اهمیت است و همواره از موضوعات مهم حوزه مهندسی نرم‌افزار و امنیت به شمار می‌آید. آزمون فازی به‌نوعی یک آزمون امنیت و نفوذ به برنامه محسوب می‌شود که برای کشف آسیب‌پذیری‌ها هم در جهت اصلاح آنها و هم برای بهره‌برداری از آنها می‌تواند مورد استفاده قرار بگیرد [14]. فازرها پس از استفاده بر روی یک برنامه معمولا اشکال‌هایی را که می‌توانند تشخیص می‌دهند و برای کشف اشکال‌ها و آسیب‌پذیری‌های جدید نیاز به روش‌های جدید تولید داده‌های آزمون می‌باشد [2]. به‌علاوه روش‌هایی که اشکال‌های بیش‌تری را با صرف هزینه و زمان کمتری پیدا می‌کنند نسبت به باقی روش‌ها اولویت دارند. بنابراین جست‌وجو برای روش‌های جدید تولید داده‌های آزمون قابل توجیه و مهم تلقی می‌شود.

۱-۵- ساختار گزارش

گزارش پیشرو مشتمل بر چهار فصل است. **فصل اول** حاوی پیش‌زمینه، شرح مسئله، مباحث تحت پوشش و نیز اهمیت موضوع بود. **فصل دوم** تحت عنوان تعاریف و مفاهیم مبنایی به دو بخش مستقل تقسیم شده است. در بخش نخست آن مفاهیم مربوط به آزمون فازی، داده آزمون، و آسیب‌پذیری‌ها مطرح شده است. بخش دوم به‌طور خاص به مبحث یادگیری ژرف و شبکه‌های عصبی پرداخته است. این بخش مقدمات لازم برای استفاده از این شبکه‌ها در راستای تولید داده‌های آزمون جدید را فراهم می‌سازد. در **فصل سوم** کارهای مرتبط و پیشین صورت گرفته روی روش‌های خودکار تولید داده آزمون برای استفاده در فازرهای مبتنی بر قالب فایل، بحث می‌شود. تمرکز اصلی این فصل بر بیان روش تولید داده آزمون با استفاده از شبکه‌های عصبی است. در نهایت **فصل چهارم** به جمع‌بندی مباحث ذکر شده، بیان چالش‌های موجود و مسایل باز در این حوزه می‌پردازد. در پایان این فصل برخی از کارهای آتی قابل انجام در قالب موضوع مورد نظر برای پایان‌نامه ارشد پیشنهاد می‌گردد. ساختـار گـزارش مطابق آنچه گفته شد در شکل (۱-۲) نشان داده شده است.



شکل (۱-۲) ساختار سلسله مراتبی سمینار.

فصل ۲: تعاریف و مفاهیم مبنایی

ما ممکن است امیدوار باشیم که ماشین‌ها در نهایت در همه زمینه‌های هوشمند با انسان رقابت خواهند کرد. اما بهترین زمینه برای شروع کدام است؟

آلِن تورینگ

۲-۱- مقدمه

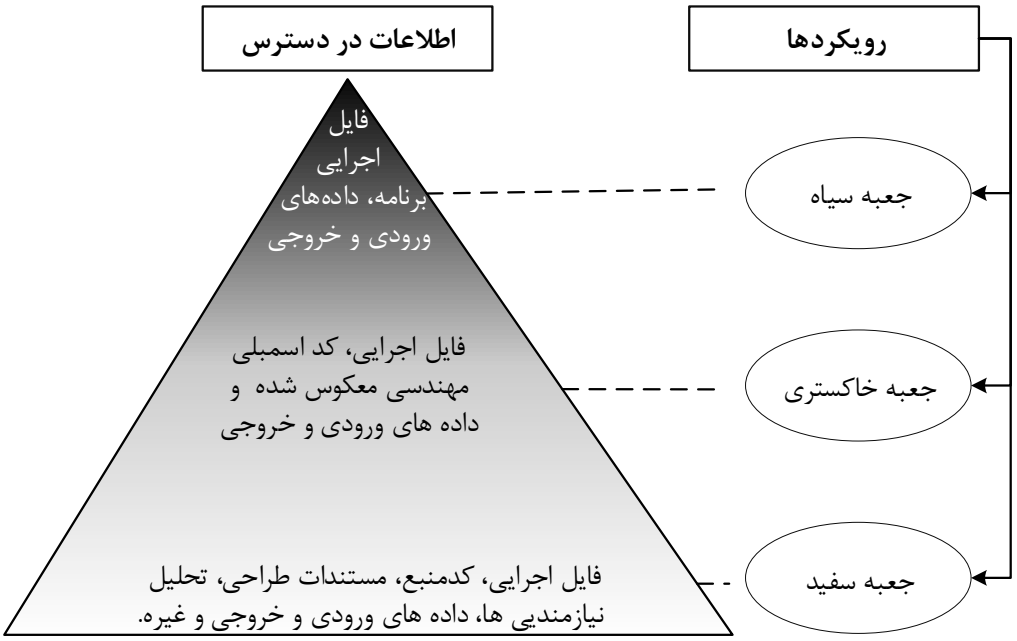
در این فصل ابتدا به‌بیان برخی از اصطلاحات رایج در حوزه آزمون نرم‌افزار به ویژه آزمون امنیت می‌پردازیم. سپس فن آزمون فازی، معماری کلی فازرها و روش‌های تولید خودکار داده‌های آزمون مورد استفاده آنها را معرفی می‌کنیم. در ادامه مختصری راجع‌به قالب‌های فایل مختلف موجود و نیز آسیب‌پذیری‌های مهم شناخته شده صحبت خواهیم کرد. بخش دوم این فصل به مبانی نظری و ریاضیاتی یادگیری ژرف[۱] تخصیص دارد. شبکه‌های عصبی ژرف[۲] نیز در ادامه آن به عنوان ابزاری برای پیاده‌سازی یادگیری ژرف معرفی خواهند شد و در نهایت شبکه‌های عصبی مکرر (RNN)[۳] توضیح داده خواهند شد که برای یادگیری وظیفه[۴]های که در آنها ترتیب اطلاعات ورودی مهم است به‌کار می‌روند. هدف استفاده از این شبکه‌ها در یادگیری ساختار قالب‌های فایل پیچیده و ایجاد یک مدل مولد بر اساس آن جهت تولید داده‌های آزمون جدید است به نحوی که پوشش کد قابل قبولی را ارایه دهند.

۲-۲- آزمون نرم‌افزار

به‌طور کلی مجموعه فنون کشف و آشکارسازی خرابی‌های نرم‌افزار را آزمون نرم‌افزار گویند. منظور از خرابی بروز یک رفتار ناخواسته و خلاف مشخصات[۵] در نرم‌افزار است. خرابی خود حاصل یک خطا (نقص[۶]) ایستا (اشتباه طراحی[۷]) در نرم‌افزار است. حالت داخلی نادرست برنامه را که ناشی از یک خطا است، اشکال[۸] می‌گویند [11]. واژه‌های خطا، اشکال و خرابی از حوزه اتکاپذیری وارد آزمون نرم‌افزار شده‌اند [20] و تعاریف یکسانی برای آنها وجود ندارد [11] و [21].

---

۱ deep learning
۲ deep neural networks
۳ recurrent neural networks
۴ task
۵ specification
۶ defect / bug
۷ design fault
۸ error



شکل (۲-۱) طرح‌واره‌ای از رویکردهای مبتنی بر جعبه بر اساس اطلاعات در دسترس به‌صورت یک مثلث آزمون.

متأسفانه آزمون تنها قادر است وجود خرابی را نشان دهد و نبود آن را تضمین نمی‌کند. به‌بیان رسمی مسئله یافتن تمامی خرابی‌ها در یک برنامه **تصمیم‌ناپذیر**[۱] است [11]. آزمون نرم‌افزار منحصر به کد اجرایی برنامه نیست و شامل آزمون نیازمندی‌ها و مشخصات و طراحی نیز می‌گردد. در آزمون فازی اما منحصرا به کد اجرایی پرداخته می‌شود. لذا در ادامه منظور از آزمون فقط آزمون مربوط به کداجرایی برنامه است. واژگان حاکم بر حوزه آزمون نرم‌افزار بسیار گسترده می‌باشد. در ادامه روی برخی از مفاهیم پایه‌ای تمرکز می‌کنیم.

۲-۲-۲- رویکرد جعبه

در دیدگاه سنتی یک رویکرد ساده نام‌گذاری مبتنی بر جعبه به آزمون نرم‌افزار وجود دارد که گرچه امروزه استفاده از این اصطلاح کمرنگ‌تر شده است؛ اما هر آزمونی را پیش از انجام، بسته

[۱] undecidable

به اطلاعات در دسترس می‌توان در یکی از سه دسته جعبه سیاه، جعبه سفید و جعبه خاکستری طبقه‌بندی کرد [11]. شکل (۲-۱) یک طرح‌واره از رویکردهای مبتنی بر جعبه ارایه می‌دهد.

رویکرد جعبه سیاه. در این رویکرد به نرم‌افزار به عنوان یک جعبه بسته نگاه می‌شود و هیچ دانشی از سازمان داخلی نرم‌افزار، کد منبع و غیره در اختیار آزمون کننده نیست. این روش هنگامی که کد منبع برنامه در دسترس نیست مورد استفاده قرار می‌گیرد. معیار بررسی تنها داده‌های ورودی و خروجی برنامه هستند[13]. از این رویکرد با عناوین آزمون عملیاتی[۱]، آزمون رفتاری[۲] و نیز آزمون جعبه‌بسته نیز یاد می‌شود [22].

رویکرد جعبه سفید. این رویکرد مبنی بر جست‌وجوی دقیق کدمنبع برنامه، منطق و ساختار آن است. آزمون کننده دانش کاملی در مورد کد منبع برنامه در اختیار دارد. معیار بررسی علاوه بر داده‌های ورودی و خروجی، گراف‌های جریان کنترلی و جریان داده‌ای و نیز میزان پوشش کد است. آزمون ساختاری[۳]، جعبه‌شیشه‌ای[۴] و جعبه‌باز نام‌های دیگر این رویکرد هستند [22].

رویکرد جعبه خاکستری. در این رویکرد مانند رویکرد اول، کد منبع در دسترس نیست اما با روش‌های مهندسی معکوس کد اسمبلی فایل اجرایی برنامه استخراج و تحلیل‌های ایستا و پویا روی آن انجام می‌شود. بدین ترتیب اطلاعات بیشتری از ساختار برنامه و ورودی‌های آن به دست می‌آید [13].

۲-۲-۳- رویکرد معیارهای پوشش

مستقل از رویکرد مبتنی بر جعبه که بر اساس میزان اطلاعات در دسترس از SUT بود، می‌توان رویکردی مبتنی بر معیارهای پوشش برای آزمون در نظر گرفت. معیارهای پوشش بیشتر با هدف کمی‌سازی و اندازه‌گیری مقدار آزمون انجام شده مطرح شده‌اند؛ هرچند متقابلا در تولید داده

---

[۱] functional testing
[۲] behavioral testing
[۳] structural testing
[۴] glass-box

آزمون هم کاربرد دارند. از آنجایی که مسئله یافت تمامی خرابی‌ها تصمیم‌پذیر نیست باید معیاری وجود داشته باشد که مشخص کند چه زمانی می‌توانیم آزمون را خاتمه دهیم و آزمون تا چه‌حد خوب انجام شده است. در [11] چهار معیار پوشش گراف[۱]، پوشش عبارت منطقی[۲]، پوشش فضای ورودی[۳] و پوشش ساختار نحوی[۴] مطرح شده است.

پوشش گراف در سطح کد اجرایی که به آن پوشش کد هم گفته می‌شود، شامل پوشش گراف جریان کنترلی[۵] برنامه می‌شود. پوشش عبارت منطقی، مقدار دهی و تعیین ارزش عبارات منطقی ظاهر شده در متن برنامه است. پوشش فضای ورودی یعنی انتخاب از بین حالت‌های مختلف ترکیب ورودی‌ها و در نهایت پوشش ساختار نحوی، استفاده از قوانین گرامر برای اعتبارسنجی[۶] داده‌های ورودی یا تولید داده‌های جدید آزمون است.

معیارهای پوشش به دو شکل مختلف قابل استفاده هستند. یک روش به این صورت است که مستقیما مقادیر داده‌های آزمون برای برآوردن یک معیار داده شده، تولید شود. این روش در برخی موارد بسیار مشکل است. به‌ویژه اگر ابزاری برای تولید خودکار داده آزمون نداشته باشیم یا ساختار ورودی پیچیده باشد. روش دیگر تولید داده‌های آزمون به‌طور کاملا مجزا و سپس اندازه‌گیری میزان پوشش معیار مربوطه است. به دلیل این نوع استفاده به معیارهای پوشش، سنجه هم اطلاق می‌شود. برای روش اول یک برنامه شناسنده[۷] برآورده شدن معیار و برای روش دوم یک برنامه مولد[۸] داده آزمون نیاز است. در صنعت استفاده از روش دوم مرسوم‌تر بوده است [11].

در این پژوهش معیار اول (پوشش گراف) با روش دوم را مبنا قرار می‌دهیم. به این ترتیب یک سازوکار تولید داده و مشاهده میزان پوشش کد در اختیار خواهیم داشت. آزمون فازی معمول بر

---

۱ graph coverage
۲ logic coverage
۳ input space coverage
۴ syntax structure coverage
۵ control flow graph
۶ validation
۷ recognizer program
۸ generator program

اساس این روش است. پوشش کد همانطور که گفته شد در سه سطح پوشش جملات، پوشش شاخه و پوشش مسیر اجرایی مطرح است. در رویکرد جعبه سفید که کد منبع برنامه وجود دارد، مشاهده پوشش کد برنامه با اضافه کردن دستوراتی به متن کد به آسانی امکان پذیر است. این عمل را ***ابزارگذاری***[۱] کد می‌گویند [16]. در محیط GNU/GCC برای این منظور کافی است برنامه را همراه با پرچم‌های `-fprofile-arcs` و `-ftest-coverage` کامپایل کنیم:

```
01.   $  gcc -g -fprofile-arcs -ftest-coverage -o test1 test1.c
02.   $  ./test1 12 3
03.   $  gcov test1.c
04.   $  more test1.c.gcov
```

خط اول دستورات فوق برنامه test1 را با پرچم‌های گفته شد کامپایل می‌کند. خط دوم برنامه را با دو آرگومان خط فرمان ۱۲ و ۳ اجرا می‌کند. خط سوم پوشش کد اجرای اخیر برنامه را محاسبه و فایلی تحت عنوان test1.c.gcov تولید می‌کند و نهایتا در خط چهارم این فایل برای مشاهده باز می‌شود. پرچم‌هایی برای اخذ انواع مختلف پوشش کد وجود دارد [16].

اگر کد منبع برنامه در اختیار نباشد اما کار اندکی دشوار می‌شود. برای این منظور باید از ابزارهای مهندسی معکوس کد باینری استفاده کرد. یک ابزار موفق در این زمینه چارچوب مهندسی معکوس PaiMei[۲] است [16].

۲-۳- آزمون فازی

آزمون فازی و فازر به‌طور خلاصه در بخش ۱-۳- معرفی شدند. ایده اولیه این آزمون بسیار ساده است. ورودی‌های ناخواسته‌ای (مخرب)[۳] تولید و به برنامه می‌دهیم. اجرای برنامه با این ورودی‌ها ممکن است آن را دچار خرابی کند و خطایی که مسبب این خرابی است با تحلیل برنامه مشخص خواهد شد. شکل (۲-۲) فرایند آزمون فازی را نشان می‌دهد. هدف اصلی، انجام کلیه فرایند

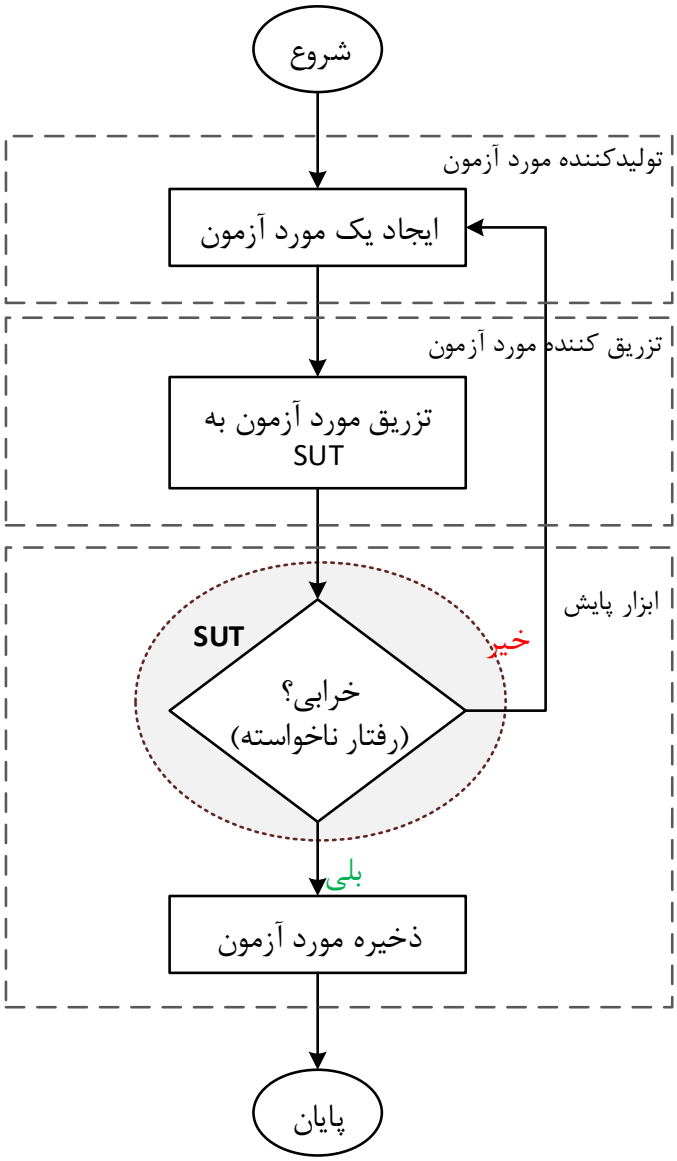
---

[۱] instrumenting

[۲] https://github.com/OpenRCE/paimei

[۳] unexpected input

به‌صورت خودکار است. SUT می‌تواند هر برنامه‌ای باشد. معمولا برنامه‌های تحت شبکه، مرورگرها و برنامه‌های مبتنی بر فایل مثل واژه‌پردازها و pdfخوان‌ها با این فن مورد آزمون قرار می‌گیرند.



شکل (۲-۲) فرایند آزمون فازی به‌صورت فلوچارت. پیمانه‌های مورد نیاز برای خودکارسازی با مستطیل خط‌چین نشان داده‌شده‌اند. فرایند به‌طور معمول دارای پایان نیست، اما در اینجا فرض شده است با یافتن خطا فرایند تمام می‌شود.

۲-۳-۲- معماری فازرها

یک معماری مرجع برای فازرها وجود ندارد ولی می‌توان از روی فرایند ترسیم شده برای آزمون فازی پیمانه[۱]های اصلی لازم برای یک فازر را پیشنهاد کرد. پیمانه اول تولید کننده مورد آزمون است که داده‌های ورودی لازم برای آزمون را تولید می‌کند. روش‌های مختلفی برای تولید داده آزمون وجود دارد که بررسی خواهند شد. پیمانه دوم تزریق کننده مورد آزمون است که وظیفه آن تحویل داده‌های تولید شده توسط تولیدکننده به SUT است. هر برنامه واسط مختص به خود را دارد. واسط می‌تواند مبتنی بر خط فرمان (CLI)، مبتنی بر گرافیک (GUI)، پروتکل شبکه یا فایل باشد. هدف این پژوهش برنامه‌هایی هستند که ورودی آنها به شکل یک فایل است. معمولا داده‌های ورودی به شکل فایل (مثل PDF) ساختار پیچیده‌تری نسبت به داده‌های خط فرمان یا پروتکل‌های شبکه دارند و تولید آنها سخت‌تر است. فازری که برای آزمون این برنامه‌ها توسعه داده می‌شود، فازر مبتنی بر قالب فایل هم نامیده می‌شود [1].

آخرین پیمانه مورد نیاز ابزار پایش است که SUT را پایش می‌کند تا در صورت بروز خرابی ضمن ذخیره حالت برنامه محل وقوع خرابی را با تحلیل حافظه برنامه مشخص کند. خوشبختانه ابزارهای زیادی برای این منظور وجود دارند. ابزار windbg برای سیستم عامل ویندوز را می‌توان برای این مقصود استفاده نمود [2].

۲-۳-۳- تولید داده آزمون

داده یا مورد آزمون مقداری است که به عنوان ورودی به SUT داده می‌شود. در آزمون پویا - آزمونی که برنامه در آن واقعاً اجرا می‌شود – نیاز به تعداد زیادی مورد آزمون داریم. آزمون فازی

[۱] module

آزمونی پویا است. روش‌هایی که برای تولید داده آزمون در آزمون فازی استفاده می‌شوند معمولا به دو دسته اصلی تقسیم می‌شوند: روش‌های مبتنی بر جهش[۱] و روش‌های مبتنی بر تولید[۲] [15].

در روش مبتنی بر جهش، تعداد یک یا بیشتر نمونه ورودی معتبر[۳] به عنوان پایه برای تولید مورد آزمون به‌کار می‌رود. این ورودی‌های نمونه جهش (تغییر ناگهانی) می‌یابند تا مورد آزمون دیگری تولید شود. جهش می‌تواند ساده باشد؛ شبیه معکوس کردن یک بیت، جایگزین کردن یک کاراکتر، یا پیچیده‌تر باشد؛ شبیه شناسایی و تکرار ساختارهای مشخص در داده، جایگزینی اعداد صحیح و حقیقی با مقادیر مرزی (بسیار بزرگ یا کوچک) و غیره. ساختن یک فازر همه‌منظوره مبتنی بر جهش و تولید ورودی بد شکل و مخرب با آن، آسان است و نیاز به شناخت قبلی ساختار داده ورودی مورد جهش ندارد. عیب روش مبتنی بر جهش اینجاست که این روش وابسته به تنوع ورودی‌های نمونه است. بدون وجود ورودی‌های مختلف با پیچیدگی کافی، فازر مبتنی بر جهش به پوشش کد بالایی دست نمی‌یابد [2]. FileFuzz [1] و radamsa [2] مثالی از فازرهای مبتنی بر جهش هستند.

روشی که در آن مورد آزمون از ابتدا ساخته شود، مبتنی بر تولید است؛ حتی اگر کاملا تصادفی باشد. اما در روش مبتنی بر تولید معمولا مشخصات داده ورودی برای ساخت یک مدل مولد داده استفاده می‌شود. این مدل سپس موردهای آزمون را تولید می‌کند. این روش بیشتر روی قالب‌های داده‌ای که یک توصیف صوری از مشخصات آنها در درسترس است به‌کار می‌رود، با این حال زمان و هزینه زیادی باید صرف شود تا مشخصات قالب داده کاملا فهمیده و مدل خوبی از آن تهیه شود [15]. SPIKEfile [1] و jsfunfuzz [2] فازرهای مبتنی بر تولید هستند.

روش‌های ترکیبی نیز وجود دارد که از ویژگی‌های هر دو روش بیان شده در تولید مورد آزمون کمک می‌گیرد. یک مثال از فازرهای ترکیبی LangFuzz [23] است. LangFuzz برای آزمون جعبه‌سیاه موتورهای مبتنی بر زبان‌های مستقل از متن طراحی شده است. در LangFuzz به این علت از روش ترکیبی استفاده شده است که روش مبتنی بر تولید محض نیاز به معرفی قواعد

---

۱ mutation-based
۲ generation-based
۳ valid

تولیدی دارد که LangFuzz را به زبان‌های خاصی مقید می‌کند. همچنین یک روش مبتنی بر جهش محض نیز، LangFuzz را محدود به استفاده از اطلاعات گرامری که در نمونه‌ها است، می‌کند.

۲-۳-۴- قالب‌های فایل

برای ساخت فازر مبتنی بر قالب فایل اولین گام شناخت مشخصات فایل ورودی برای SUT است. فازر مبتنی بر قالب فایل از این حیث حائز اهمیت است که بسیاری از برنامه‌ها را می‌توان مبتنی بر فایل دانست. برای مثال مرورگرها طیف وسیعی از قالب‌های فایل را به‌عنوان ورودی می‌پذیرند. فایل‌ها را می‌توان به دو دسته متنی و دودویی تقسیم بندی کرد. بر خلاف قالب‌های متنی، قالب‌های دودویی ظاهر کاربر پسندی ندارند و معمولا جاهایی استفاده می‌شوند که ساختارهای پیچیده با وابستگی‌های داخلی زیاد نیاز به ذخیره به‌صورت کارایی (از بعد فضا و زمان) دارند؛ برای مثال فایل‌های اجرایی و چندرسانه‌ای. یک فایل پیکربندی ساده به‌ندرت نیاز به ذخیره سازی به‌صورت دودویی دارد [16].

برنامه به‌طور معمول دو گام مجزا را برای پردازش یک فایل طی می‌کند: گام اول پویش[۱] فایل و گام دوم پرداخت[۲] آن. در مرحله پویش فایل در حافظه بارگذاری، مقادیر فیلدهای آن خوانده شده و تبدیل به داده‌ساختارهای داخل حافظه اصلی (مثل ساختمان، آرایه، بافر و غیره) می‌شود. در این مرحله چنانچه اشکال نحوی در ساختار فایل باشد (فایل از مشخصه‌های قالب خود پیروی نکند) باید توسط پویشگر تشخیص داده شود وگرنه منجربه خرابی می‌شود. در مرحله پرداخت، برنامه روی اطلاعات خوانده شده از فایل پردازش لازم را انجام می‌دهد و خروجی تولید می‌کند (مثلا نمایش یک تصویر روی صفحه نمایش یا اجرای یک ویدئو و غیره) [16]. خطاهای این مرحله معمولا جدی‌تر هستند و تشخیص آن نیز مشکل‌تر است.

---

[۱] parse

[۲] render

در مشخصات قالب‌های فایل دودویی، فیلدها نام‌گذاری می‌شوند و یک نوع داده‌ای، طول و گاهی اوقات یک مقدار ثابت به‌عنوان صفت دارند. فیلدها آدرس‌های نسبی نسبت به ابتدای فایل هستند. خیلی از قالب‌های فایل دودویی به‌صورت عبارات شبه منظم یا نمادهای شبه EBNF مشخص می‌شوند. EBNF یک نماد گذاری برای بیان گرامرهای مستقل از متن به‌صورت فشرده و با خوانایی بالاست. شبه EBNF به نوعی مشابه شبه کد است که توضیحی سطح بالا از یک الگوریتم کامپیوتری، مناسب فهم انسان، ارایه می‌دهد، ولی جزئیات اجرایی مورد نیاز برای اجرا توسط کامپیوتر را حذف کرده نموده است. مشخصات شبه EBNF (و عبارت منظم) اغلب با توضیحات اضافه‌ای به زبان طبیعی که قابل بیان به‌صورت رسمی نیست، همراه می‌شوند. این توضیحات معمولا شامل روابط حساس به متن نظیر ارتباط اندازه آرایه با طول واقعی آن یا ارتباط یک اشاره‌گر آدرس با مکان واقعی داده اشاره شده در فایل هستند. اینگونه روابط حساس به متن قابل بیان به‌صورت گرامر مستقل از متن نیستند [24].

خانواده قالب فایل‌های دودویی. از قالب‌های فایل دودویی با یک ساختمان مشابه، به عنوان خانواده قالب فایل دودویی یاد می‌شود. دو خانواده از فایل دودویی وجود دارند که به آسانی قابل تفکیک هستند: ۱- قالب‌های فایل مبتنی بر تکه[۱] و ۲- قالب‌های فایل مبتنی بر دایرکتوری[۲]. البته این دو خانواده تمامی ساختارهای قالب فایل را شامل نمی‌شوند. برای نمونه قالب‌های فایل اجرایی، ساختار متفاوتی دارند [24].

قالب‌های فایل مبتنی بر تکه توسط شرکت‌های Electronic Art و Commodore-Amiga به عنوان IFF[۳] ایجاد شدند. یک فایل IFF خودش یک تکه IFF هست. یک تکه متشکل از یک برچسب شناسه id، یک اندازه size و تعداد size بایت داده است. خود داده ممکن است شامل تکه باشه که زیر-تکه خوانده می‌شود. زیر-تکه‌ها همان ساختار تکه‌ها را دارند. زیر-تکه‌ها نیز خود می‌تواند شامل زیر-تکه‌های دیگر باشند (ساختار تو در تو). قالب‌های فایل مبتنی بر تکه به‌صورت

[۱] chunk-based
[۲] directory-based
[۳] interchange file format

عمده به عنوان نگهدارنده چندرسانه‌ای استفاده می‌شوند؛ اما، همچنین برای فایل‌های واژه پردازها شامل تصاویر و دیگر موارد به‌کارگرفته شده‌اند.

یک قالب فایل مبتنی بر دایرکتوری از تعداد یک یا بیشتر جدولِ فرهنگ لغت تشکیل شده است که شامل یک یا چند مدخل دایرکتوری هستند. یک مدخل دایرکتوری معین می‌کند که داده واقعی برای یک نوع از اطلاعات در کجا قرار دارد. این مدل در فایل‌های TIFF و OLE[1] استفاده شده است.

۲-۴- آسیب‌پذیری‌های نرم‌افزار

تعاریف گوناگونی برای اصطلاح *آسیب‌پذیری نرم‌افزار* وجود دارد. می‌توان آسیب‌پذیری را اجتماع سه عنصر دانست: وجود یک خطا در نرم‌افزار، دسترسی مهاجم به خطا و قابلیت مهاجم برای بهره‌برداری از خطا [25]. چنانچه قبلا اشاره شد، خطا، نرم‌افزار را در یک حالت اشکال قرار می‌دهد که منجربه خرابی می‌شود.

همه خطاهای برنامه‌نویسی یکسان نیستند. برخی از آنها به فرد مهاجم امکان به دست آوردن اطلاعات یا قابلیت‌هایی را که پیش از این نداشته می‌دهند. آنها ممکن است فرد مهاجم را قادر کنند تا با خراب کردن برنامه دسترسی سایر کاربران به آن را غیر ممکن سازد یا به اطلاعاتی دسترسی پیدا کند که نبایستی قابل دسترسی برای همه باشد. در مواردی نیز باعث می‌شود تا فرد مهاجم بتواند هر دستوری را به برنامه بگوید و برنامه آن را اجرا کند. بعضی خطاها ممکن است تنها منجربه خرابی نرم‌افزار و بروز رفتار ناخواسته (مغایر با مشخصات آن) شوند. بنابراین می‌توان گفت صرف وجود یک خطا در نرم‌افزار آسیب‌پذیری محسوب نمی‌گردد [14] و [16].

اگرچه آزمون فازی می‌تواند برای اهداف مختلفی استفاده شود، اما مهمترین هدف آن تحلیل نرم‌افزار به منظور کشف آسیب‌پذیری‌ها است. کشف آسیب‌پذیری‌ها در ارتباط مستقیم با امنیت

[1] Microsoft object link and embedding

است. امنیت اطلاعات با سه خصلت محرمانگی[۱]، جامعیت[۲] و دسترسی‌پذیری[۳] (به اختصار CIA) تعریف می‌شود. نقض محرمانگی می‌تواند در صورت دسترسی به هرگونه اطلاعات محرمانه رخ دهد. نقض جامعیت در سوی دیگر یعنی اصلاح و دستکاری هرگونه داده حتی بدون افشای آن. بلأخره نقض دسترسی یعنی هر گونه خرابی یا تنزل در سرویس ارایه شده توسط سیستم نرم‌افزاری. مسائل دسترسی‌پذیری معمولا آسان‌ترین برای تشخیص هستند. آسیب‌پذیری‌های مرتبط به مسائل دسترسی‌پذیری معمولا در قالب حمله ممانعت از سرویس (DoS)[۴] بهره‌برداری می‌شوند. هنگامی آسیب‌پذیری سرریز حافظه که اجرای کد در سیستم هدف را میسر می‌کند، وجود داشته باشد؛ نتیجه اغلب خطر پذیری کامل سیستم و نقض هر سه خصلت یاد شده است [14].

پس از آنکه برنامه هدف آزمون مشخص شد، مهم است که بدانیم چه نوع خطاهایی برای یافتن وجود دارد. انواع مختلفی از خطاها وجود دارند که می‌توانند توسط فازرها کشف شوند؛ به‌ویژه آنهایی که از جنس نقض دسترسی به حافظه هستند. پیش از بیان هر نوع خطا و یا آسیب‌پذیری بهتر است ساختار حافظه برنامه به دقت تشریح شود تا عملکرد برنامه در حال اجرا مشخص شده و مفهوم خطاهای مختلف حافظه به روشنی قابل بیان باشد. در ادامه مرور کوتاهی بر ساختار حافظه در اختیار برنامه خواهیم داشت و سپس به‌بیان خطاها می‌پردازیم.

۲-۴-۱- ساختار حافظه

حافظه یک برنامه کامپیوتری در ساده‌ترین حالت به دو قسمت فقط خواندنی و فقط نوشتنی قابل تقسیم است. این تفکیک از سیستم‌های اولیه که برنامه در حافظه فقط خواندنی ذخیره می‌شد، نشأت گرفت. با ظهور حافظه‌های خواندنی نوشتی این ایده که بعضی بخش‌های حافظه برنامه، مثلا کد اجرایی، نبایستی در حین اجرا تغییر کنند، کماکان حفظ شد. این بخش‌ها تحت

[۱] confidentiality
[۲] integrity
[۳] availability
[۴] denial of service

عنوان بخش text. و بخش rodata. از برنامه هستند. مابقی حافظه نیز به بخش‌های مجزا تقسیم می‌شوند که مختص وظایف مختلفی‌اند.

text. بخش کد که همچنین به نام بخش text نیز شناخته می‌شود، جایی است که یک تکه از فایل مقصد یا آن بخش از فضای آدرس مجازی برنامه که حاوی کد دستورات اجرایی برنامه است، ذخیره می‌شود و معمولا فقط خواندنی و در اندازه ثابت است.

data. بخش داده حاوی متغیرهای سراسری یا ایستای برنامه است، که یک مقدار از پیش تعریف شده دارند و می‌توانند تغییر هم یابند. یعنی، هر متغیری که داخل بدنه یک تابع تعریف نشده باشد (و بنابراین از هر جایی قابل دسترسی است). یا داخل بدنه تابع تعریف شده ولی به‌صورت ایستا، پس مقدار خود را همواره حفظ می‌کند (با این تفاوت که فقط از داخل تابع قابل دسترسی است)، در این بخش برای آن حافظه اخذ می‌شود. برای مثال در کد زیر به زبان C متغیرهای val و str در حافظه بخش data. مقیم می‌شوند.

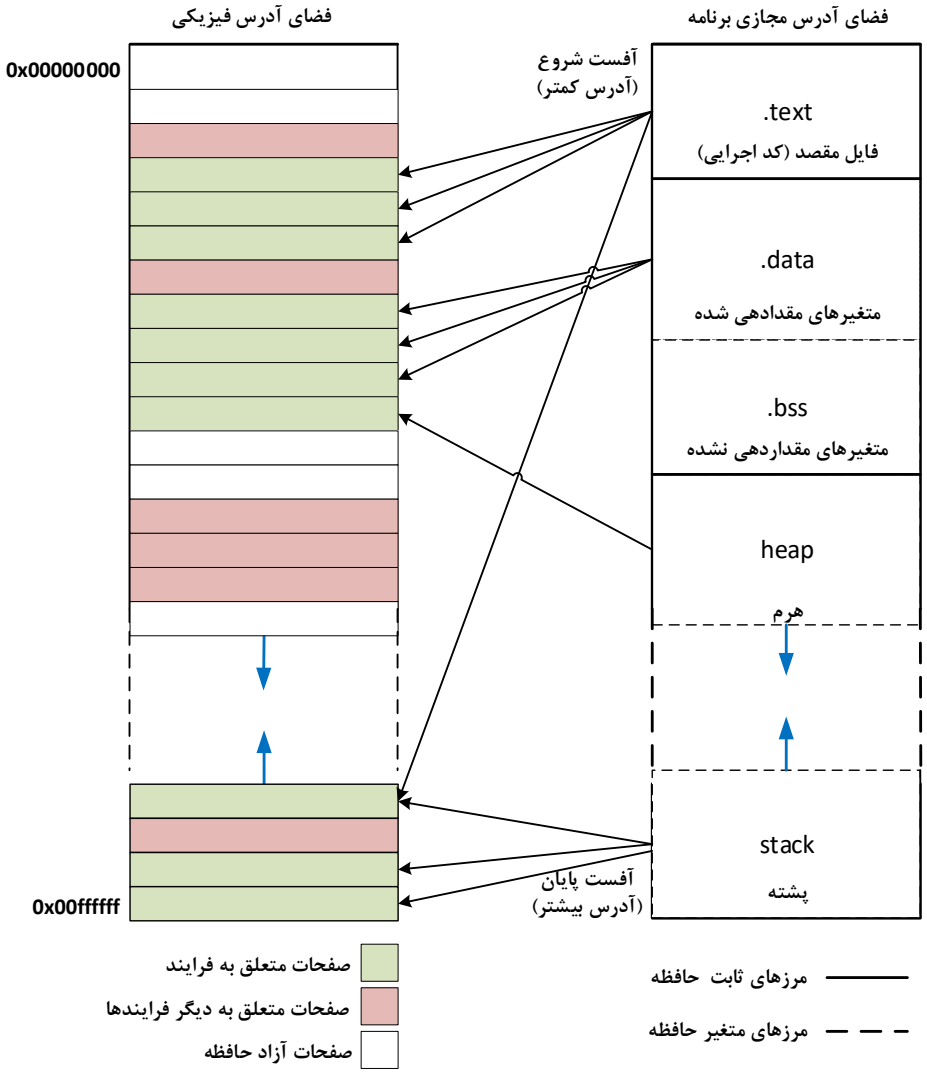
```
05.   int val = 3;
06.   char str[] = “hello world!”;
07.   int main () {}
```

مقدارهای این متغیرها در آغاز در بخش فقط خواندنی حافظه برنامه (بخش text.) ذخیره می‌شوند و در طول روال شروع برنامه در بخش data. کپی می‌شوند.

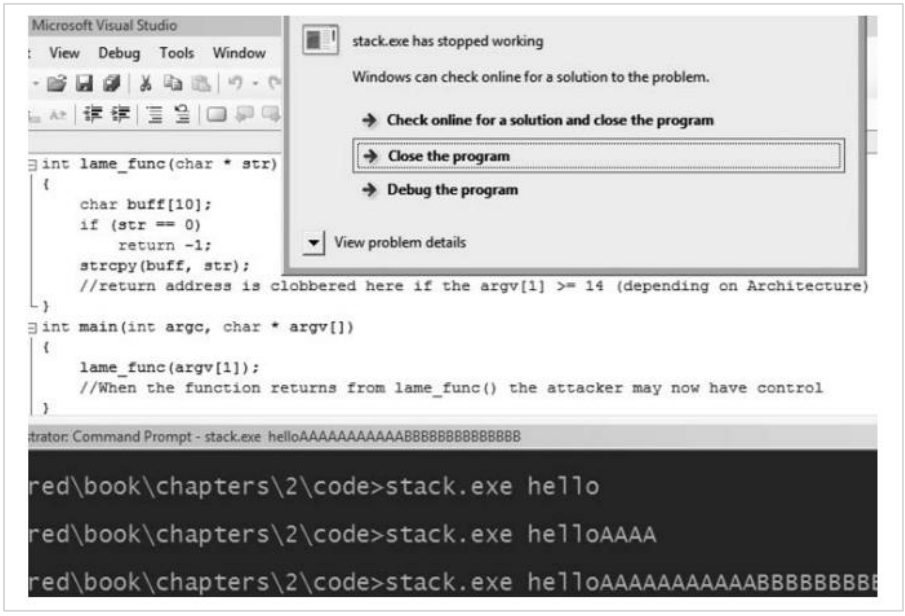
bss. این بخش که به نام داده‌های مقداردهی نشده هم شناخته می‌شود، همجوار بخش داده است. bss. حاوی همه آن متغیرهای سراسری یا ایستایی می‌شود که به‌صورت صریح در متن کد برنامه مقداردهی نشده و یا به مقدار صفر مقداردهی شده‌اند. برای مثال متغیر i که به‌صورت `static int i;` در برنامه تعریف شده است در بخش bss. حافظه نگهداری می‌شود.

heap. محدوده حافظه هرم در پایان بخش bss. شروع شده و از آنجا به سمت آدرس‌های بزرگتر رشد می‌یابد. حافظه هرم با دستورات malloc، calloc، realloc و free مدیریت می‌شود که ممکن است از فراخوان‌های سیستمی brk و sbrk برای تنظیم اندازه استفاده کنند. محدوده آن بین همه ریسمان‌ها، کتابخانه‌های مشترک و پیمانه‌هایی که به‌صورت پویا بارگذاری شده‌اند،

مشترک است. فضای تخصیص یافته به هرم در برنامه با یک آدرس شروع و یک طول (اندازه) مشخص می‌شود. آدرس پایه یا شروع هرم در طول اجرای برنامه ثابت است.



شکل (۲-۳) قسمت‌های مختلف فضای تخصیص داده شده به برنامه در حال اجرا. **راست:** نحوه قرار گرفتن بخش‌های مختلف در کنار یکـدیگر. **چپ:** صفحه‌بندی و تخصیص بخش‌ها در حافظه مجازی.



شکل (۲-۴) مثالی از خطای سرریز پشته در یک قطعه کد C نوشته شده در محیط visual studio [14].

stack. محدوده پشته حاوی پشته برنامه است که به‌صورت یک داده‌ساختار LIFO[1] معمولا در بالاترین جای حافظه تخصیص یافته قرار دارد. اشاره‌گر پشته، بالای پشته (آدرس کوچکتر) را دنبال می‌کند. این اشاره‌گر هرگاه محتوایی push می‌شود تنظیم می‌گردد. مجموعه مقادیری که برای یک فراخوانی تابع داخل پشته push می‌گردند، قاب پشته[2] نامیده می‌شود. قاب پشته در کمترین حالت حاوی آدرس برگشت است. متغیرهای خودکار نیز روی پشته ذخیره می‌شوند.

محدوده پشته و محدوده هرم به سمت یکدیگر رشد می‌کنند؛ آدرس هرم به سمت پایین افزایش می‌یابد و آدرس پشته به سمت بالا کاهش می‌یابد (شکل (۲-۳)). وقتی که اشاره‌گر پشته به اشاره‌گر هرم می‌رسد، حافظه آزاد تمام می‌شود. البته در فن حافظه مجازی روند متفاوت است. در معماری x86 (معماری کامپیوتر استاندارد) پشته به سمت آدرس صفر رشد می‌کند، بدان معنی که داده‌های اخیر و عمیق‌تر در زنجیره فراخوانی، در آدرس عددی کمتری هستند و به بخش هرم نزدیک ترند. در برخی معماری‌ها نیز پشته به سمت مخالف رشد می‌کند. کلیه مطالب ذکر شده در شکل (۲-۳) به‌صورت یک طرح‌واره نشان داده شده‌اند.

[1] last in first out
[2] stack frame

۲-۴-۲- خطاهای حافظه

خطاهای فساد حافظه بدون شک رایج‌ترین و مؤثرترین روش بهره‌برداری خرابکارانه از یک سیستم کامپیوتری محلی یا راه دور هستند. اگر حافظه بتواند به روشی خراب شود (یک آدرس بازگشت، اشاره‌گر پشته، اشاره‌گر تابع و غیره) اغلب اجرا می‌تواند به کد تهیه شده توسط مهاجم هدایت شود. بنابراین اغلبِ موارد خرابی حافظه، آسیب‌پذیری به حساب می‌آیند.

بیشترین آسیب‌پذیری‌هایی که معمولا توسط محققین حوزه امنیت کشف می‌شوند، آسیب‌پذیری‌های مرتبط با میانگیر[۱] هستند. یک میانگیر یک ناحیه ثابت از حافظه اصلی در اختیار برنامه است، برای منظور خاصی، کنار گذاشته شده و معمولا در همان بخش عمومی از حافظه است که دیگر داده‌های مورد استفاده برنامه هم، حضور دارند.

عبارت *سرریز*[۲] *میانگیر* در زمینه امنیت رایج است و عموما به معنای این است که چیزهایی بدی در حال اتفاق افتادن هستند! در عین درست بودن این حرف اما دقیق نیست. برای مثال یک خطای ایستا در پشته می‌تواند منجربه نقض دسترسی شود، یا یک تخصیص میانگیر در هرم هم می‌تواند سبب نقض دسترسی شود. هر دو خطای سرریز (سرریز میانگیر) هستند و هر دو به‌صورت سنتی قابل بهره‌برداری هستند. در حالی که یکی سرریز پشته است و دیگر سرریز هرم. در اینجا هریک را به‌طور جداگانه تعریف می‌کنیم تا بتوانیم به‌نحو دقیقی هر خطای فساد حافظه‌ای را مشخص کنیم. یک درک ابتدایی از چگونگی بهره‌برداری هر نوع آسیب‌پذیری نیز بیان خواهد شد [14] و [16].

سرریز پشته. سرریز پشته باعث می‌شود که بخش پشته از حافظه، به دلیل بررسی نامناسب حد و مرزها وقتی دستورات نوشتن حافظه اعمال می‌شوند، خراب شود. شکل (۲-۴) یک خطای سرریز پشته را نشان می‌دهد. این مثال روی ویندوز **۷** نشان داده شده است و به دلیل بهبودهای امنیتی صورت گرفته در این نسخه از سیستم عامل، این خطا قابل بهره‌برداری برای حمله DoS

---

[۱] buffer

[۲] overflow

یا DoS توزیع‌شده (DDoS)[۱] نیست. اما دیگر سناریوهای سرریز پشته یا همین کد بر روی سیستم‌های عامل نسخه پایین‌تر ویندوز می‌تواند جهت اجرای کد مخرب فراهم شده توسط حمله کننده، مورد بهره‌برداری قرار گیرد.

رشته قالب. نقص رشته قالب در ابتدای شناسایی آن در دهه ۱۹۹۰م نقص بزرگ و خطرناکی بود. در ادامه با توجه به اینکه توسط فنون تحلیل ایستا (حسابرسی کد[۲]) به راحتی قابل تشخیص بود، به‌حاشیه رانده شد. توابعی که در آن نقص رشته قالب می‌تواند رخ دهد توابعی هستند که کاراکترهای قالب مختلفی را برای درست کردن خروجی می‌پذیرند. نظیر تابع printf() در زبان برنامه‌نویسی C. برای مثال تکه کد صحیح `printf("%s", user_supplied_buff);` ممکن است به شکل نادرست زیر استفاده گردد:

```
01.   #include <stdio.h>
02.   int my_format_func(char * buff)
03.   {
04.      printf(buff);
05.      return 0;
06.   }
07.   int main(int argc, char * argv[])
08.   {
09.      my_format_func(argv[1]);
10.   }
```

چنین خطایی در ابتدا بی‌ضرر تلقی می‌گردد؛ زیرا، برنامه به درستی اجرا می‌شود. اگر این برنامه با ورودی یا آرگومان به عنوان مثال %x اجرا شود، داده خروجی بر خلاف انتظار کاربر همان %x نخواهد بود. بلکه چیزی شبیه 19f908 باز می‌گردد؛ چرا که تابع printf() کاراکتر %x را به عنوان یک کاراکتر مخصوص قالب استفاده می‌کند. آرگومان‌های روی پشته برای فراخوانی تابع `printf("%x", number)`، اول کاراکتر %x است که مخصوص چاپ عدد است و دوم عددی است که برنامه‌نویس می‌خواهد در خروجی چاپ کند. در مثال فوق تابع printf() مقدار بعدی روی پشته (بعد از رشته قالب) را چاپ می‌کند. از آنجایی که داده معتبری برای چاپ پاس داده نشده است، تابع داده بعدی روی پشته را که ممکن است یک متغیر محلی، اشاره‌گر قاب پشته، آدرس برگشت یا غیره باشد، چاپ می‌کند. بنابراین این تکنیک می‌تواند برای بررسی پشته

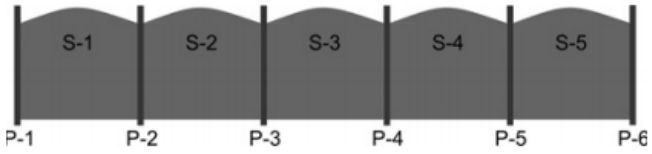
[۱] distributed denial of service
[۲] code auditing

جهت استخراج مقادیر جالب مثل آدرس بازگشت به‌کار رود. به‌صورت مشابه %d و %s هم می‌توانند مقادیری از داخل پشته را چاپ کنند. این در حالی است که کاراکتر مخصوص قالب %n می‌تواند برای نوشتن یک مقدار در حافظه (پشته) به‌کار رود. %n تعدادی بایت را در یک آدرس پشته می‌نویسد. یک بهره‌برداری عادی می‌تواند ترکیب این فنون را برای بازنویسی کردن یک اشاره‌گر تابع کتابخانه‌ای و یا آدرس بازگشت، جهت در اختیار گرفتن کنترل سیستم و اجرای کد بدخواه، به‌کار بندد.

سرریز عدد صحیح. یک نوع داده‌ای مهم در هر زبان برنامه‌نویسی عدد صحیح است. عدد صحیح می‌تواند علامت‌دار (مثبت، صفر، منفی) و بدون علامت (مثبت، صفر) باشد. در یک سیستم ۳۲بیتی، عدد صحیح بدون علامت بین ۰ (۳۲بیت صفر) تا ۴۲۹۴۶۹۷۲۹۵ (۳۲بیت یک) متغیر است. نمایش دودویی این مقادیر به‌صورت ۳۲ صفر یا یک متوالی است. اضافه کردن یک واحد به مقدار ۴۲۹۴۶۹۷۲۹۵، عدد ۴۲۹۴۶۹۷۲۹۶ را تولید نمی‌کند، به‌جای آن عدد صفر تولید می‌شود. برای ذخیره کردن عدد بزرگتر از ۴۲۹۴۶۹۷۲۹۵ به حداقل ۳۳ بیت نیاز است (مثلا عدد ۴۲۹۴۶۹۷۲۹۶ به‌صورت یک رقم ۱ و ۳۲ صفر جلوی آن است). CPU به هنگام ذخیره عدد بزرگتر به سادگی ۳۲بیت کم ارزش را ذخیره می‌کند و مشخص می‌کند که عدد خیلی بزرگ است (بیت پرچم سرریز). ضرب اعداد صحیح نیز ممکن است سبب سرریز عدد صحیح شود. اگر برنامه‌نویسی احتمال دهد که جمع یا ضرب انجام شده روی اعداد صحیحی ممکن است عدد بزرگتری ایجاد نماید و سرریز عدد صحیح را بررسی و کنترل نکند، ممکن است یک آسیب‌پذیری را به وجود آورد. برای مثال برای نگهداری n قلم داده که هریک m بایت طول دارند، برنامه‌نویس ممکن است به برنامه بگوید که یک حافظه m*n بایت اخذ کن. اگر حاصل ضرب m*n از بزرگترین عدد قابل نمایش بزرگتر باشد، سرریز رخ می‌دهد، حافظه کمتری از مقدار درخواستی تخصیص داده می‌شود و ممکن است منجربه سرریز میانگیر شود.

+2,147,483,647	0111 1111 1111 1111 1111 1111 1111 1111
–2,147,483,648	1000 0000 0000 0000 0000 0000 0000 0000

شکل (۲-۵) نمایش دودویی اعداد صحیح علامت‌دار ۳۲بیتی [16].



شکل (۲-۶) خطای off-by-one در شمارش [16].

در مورد اعداد صحیح علامت‌دار برای ۳۲بیت محدوده آنها از ۲۱۴۷۴۸۳۶۴۸- تا ۲۱۴۷۴۸۳۶۴۷+ خواهد بود. این نمایش کمی پیچیده‌تر است؛ زیرا، پرارزش‌ترین بیت (بیت سمت چپ) بیت علامت است. یک نماد منفی و صفر نماد عدد مثبت است و اعداد در شیوه مکمل دو نمایش داده می‌شوند. نمایش دودویی این مقادیر به‌صورت شکل (۲-۵) است. نمایش مکمل دو رایج‌ترین نمایش اعداد صحیح علامت‌دار است و مزایای زیادی را فراهم می‌کند. ازجمله امکان جمع اعداد مثبت و منفی بدون نگرانی در مورد وضعیت بیت علامت. اگر برنامه‌نویس فرض کند که مقدار یک متغیر عدد صحیح فقط مثبت است، اما از عدد علامت‌دار استفاده کند، عملیات حسابی ممکن است بیت علامت عدد را بازنویسی کرده و سبب تولید عدد منفی شود که می‌تواند منجربه رفتار آسیب‌پذیر قابل بهره‌برداری گردد. توجه شود که برای مثال معادل عدد صحیح بدون علامت ۲۱۴۷۴۸۳۶۴۸ در حالت علامت‌دار عدد ۲۱۴۷۴۸۳۶۴۸- است!

برخی خطاهای مشابه دیگر شبیه malloc(0) می‌تواند منجربه ایجاد یک میانگیر تُهی شود. در نتیجه وقتی در ادامه در یک دستور دیگر استفاده می‌شود، خطای اشاره‌گر تهی اتفاق می‌افتد. همچنین اگر یک عدد به‌صورت علامت‌دار یا غیر علامت‌دار خوانده شود و در ادامه خلاف نوعش استفاده شود، مسائل مشابهی رخ می‌دهد.

خطای Off-by-One. خطای Off-by-One به دو حالت مختلف اطلاق می‌شود. یک حالت در اشتباه شمارشی انجام شده توسط برنامه‌نویس و دیگری ناشی از ورودی ناخواسته‌ای که کنترل نشده است. در حالت اول به نظر می‌رسد که هرکسی نمی‌تواند شمارش کند نباید برنامه‌نویسی کند (!) ولی این خطا در واقع ناشی از عدم توانایی برنامه‌نویسی در شمارش نیست بلکه ناشی از روشی است که او می‌شمارد. یک مثال معروف در این زمینه شمارش تیرهای‌های یک حصار است به این صورت که چه تعداد تیر برای ایجاد ۲۵ متر حصار نیاز است به نحوی که فاصله تیرها ۵

متر به باشد؟ می‌توان پاسخ را با تقسیم ۲۵ به ۵ محاسبه کرد و جواب ۵ را به‌دست آورد. درحالی که این عدد برای تعداد بخش‌های حصار درست است (S-1 تا S-5)، جواب صحیحی برای تعداد تیرها نیست. پاسخ درست برای تعداد تیرها ۶ است (p-1 تا p-6)، که در شکل (۲-۶) نشان داده است. در عین سادگی این اشتباه اما بسیار رایج است.

مثال دیگر استفاده ناصحیح از عملگرهای مقایسه است. مثلا استفاده از < به جای => و بالعکس. یک خطا در کد زیر که برای چاپ اعداد صحیح از ۱ تا ۱۰ است، در زمان اجرا خود را نشان می‌دهد. عدد ۱۰ چاپ نمی‌شود و برای این منظور باید شرط حلقه از < به => اصلاح شود.

```
01.   #include <stdio.h>
02.   int main(void)
03.   {
04.      int Count = 1;
05.      while (Count <10)
06.      {
07.          printf(“%d”,Count);
08.          ++Count ;
09.      }
10.   return 0;
11.   }
```

حالت دوم از خطای off-by-one از نظر فنی بیشتر حائز اهمیت است. این حالت زمانی اتفاق می‌افتد که یک بایت اضافی در یک میانگیر خاص نوشته شود. در بعضی از سیستم‌ها (به‌ویژه معماری‌های little-endian مثل معماری اینتل x86) این امر چنانچه بافر مستقیما همجوار اشاره‌گر آغاز پشته (ebp) یا اشاره‌گرهای دیگر توابع باشد[۱]، می‌تواند منجربه بهره‌برداری شود. یک تکه‌کد بد می‌تواند شبیه زیر باشد:

```
01.   int off_by_one(char *s)
02.   {
03.      char buf[32];
04.      memset(buf, 0, sizeof(buf));
05.      strncat(buf, s, sizeof(buf));
06.   }
```

[۱] این همان مشکل سرریز پشته است با این تفاوت که بهره‌برداری از آن متفاوت خواهد بود.

تابع strncat یک بایت اضافی را در میانگیر کپی می‌کند. زیرا این تابع اندازه داده شده در آرگومان را کپی کرده و یک بایت تهی یا 0x00 را، که نشان‌دهنده خاتمه رشته است، هم می‌نویسد. اگر این میانگیر همجوار اشاره‌گر قاب پشته (ebp) باشد، چیزی شبیه به آدرس 0x08041200 می‌شود که کم ارزش‌ترین بایت (LSB) یعنی دو رقم سمت راست آن، صفر (تهی) است. در پشته x86 به‌ترتیب اعمال زیر انجام می‌شود:

۱. درج
   i. آرگومان‌های تابع روی پشته *push* می‌شوند.
۲. فراخوانی
   i. آدرس دستور اجرایی بعد از فراخوانی به عنوان آدرس بازگشت روی پشته *push* می‌شود.
۳. ورود
   i. epb روی پشته *push* می‌شود.
   ii. ebp برابر esp می‌شود.
   iii. یک فضا برای متغیرهای محلی با تفریق آن مقدار از اشاره‌گر پشته ایجاد می‌شود.
۴. خروج
   i. esp برابر ebp قرار داده می‌شود.
   ii. dword (۴بایت) به داخل ebp، *pop* می‌شود.
۵. بازگشت
   i. آدرس بازگشت به eip انتقال داده می‌شود.

اگر ebp خراب شده باشد، اشاره‌گر پشته قبل از بازگشت در جای درست خود نیست. چون LSB تهی است، این امکان هست که وقتی آدرس بازگشت اجرا می‌شود، esp به یک میانگیر تهیه شده توسط کاربر اشاره کند که منجربه خطای off-by-one می‌شود. این امر می‌تواند سبب اجرای کد دلخواه نیز گردد. برخی کامپایلرها برای مقابله با این مشکل تمهیداتی مثل بررسی محدوده آرایه در زمان اجرا را لحاظ می‌کنند.

خطای سرریز هرم. سرریز هرم وقتی است که داده بیرون از محدوده یک تکه اخذ شده از حافظه هرم نوشته شود. حافظه هرم در زبان‌های C و C++ در زمان اجرا و توسط خانواده توابع malloc() اخذ می‌شود. همانند سرریز پشته، اطلاعات کنترلی که در مرز حافظه ذخیره شده‌اند

وقتی با داده‌های تهیه شده توسط مهاجم بازنویسی می‌شوند، منجربه تغییر مسیر اجرا و پرش به کد دلخواه می‌گردند. در اینجا نیز راه‌ها و شرایط گوناگونی وجود دارد که تحت آنها خطا می‌تواند قابل بهره‌برداری باشد و یا نه. همچنین محافظت‌های مختلفی مثل بررسی جامعیت هرم می‌تواند برای جلوگیری از حملات نام‌برده انجام شود.

بهره‌برداری از اطلاعات موجود در محدوده هرم کمی پیچیده‌تر از بازنویسی آدرس بازگشت از یک پشته یا اشاره‌گر SEH[1] است. بهره‌برداری همچنین وابستگی زیادی به نحوه پیاده‌سازی تابع کتابخانه‌ای malloc و مشتقات آن دارد. این مسئله تعجب برانگیز نیست چرا که حتی بهره‌برداری از سرریز پشته در محیط ویندوز با لینوکس متفاوت است.

به‌طور کلی در بهره‌برداری از آسیب‌پذیری‌ها معمولا آدرس بازگشت پشته با آدرسی که به یک shellcode اشاره می‌کند، بازنویسی می‌شود و در نتیجه آن کنترل اجرای برنامه کاربر به برنامه فراهم شده توسط حمله کننده منتقل می‌شود و کد بدخواه حمله کننده به اجرا در خواهد آمد. این امر هنگام بازگشت از توابع برنامه اصلی یا در فراخوانی بعدی آن تابع اتفاق می‌افتد.

بازنویسی متغیرهای (مقداردهی نشده) پشته و هرم. این دسته جدیدتری از آسیب‌پذیری‌ها است و معمولا بهره‌برداری موفق از آن مشکل است. تکه‌کد زیر این مدل را توضیح می‌دهد:

```
01.   // uninitialized.c program
02.   int un_init(char *s)
03.   {
04.      char buf[32];
05.      int logged_in;
06.      if ( strlen(s) > 36)
07.      {
08.         printf("String too long!\r\n");
09.         logged_in =0;
10.      }
11.      else
12.         strncpy(buf, s, strlen(s) );
13.      if (logged_in == 0x41414141)
14.         printf("hi -- you should never see this, because
   logged_in is never set by program code.\r\n");
15.   }
16.   //main function
17.   int main(int argc, char * argv[])
```

[1] structured exception handling

```
18.   {
19.      un_init(argv[1]);
20.   }
```

در زیر برنامه با سه ورودی مختلف اجرا شده است:

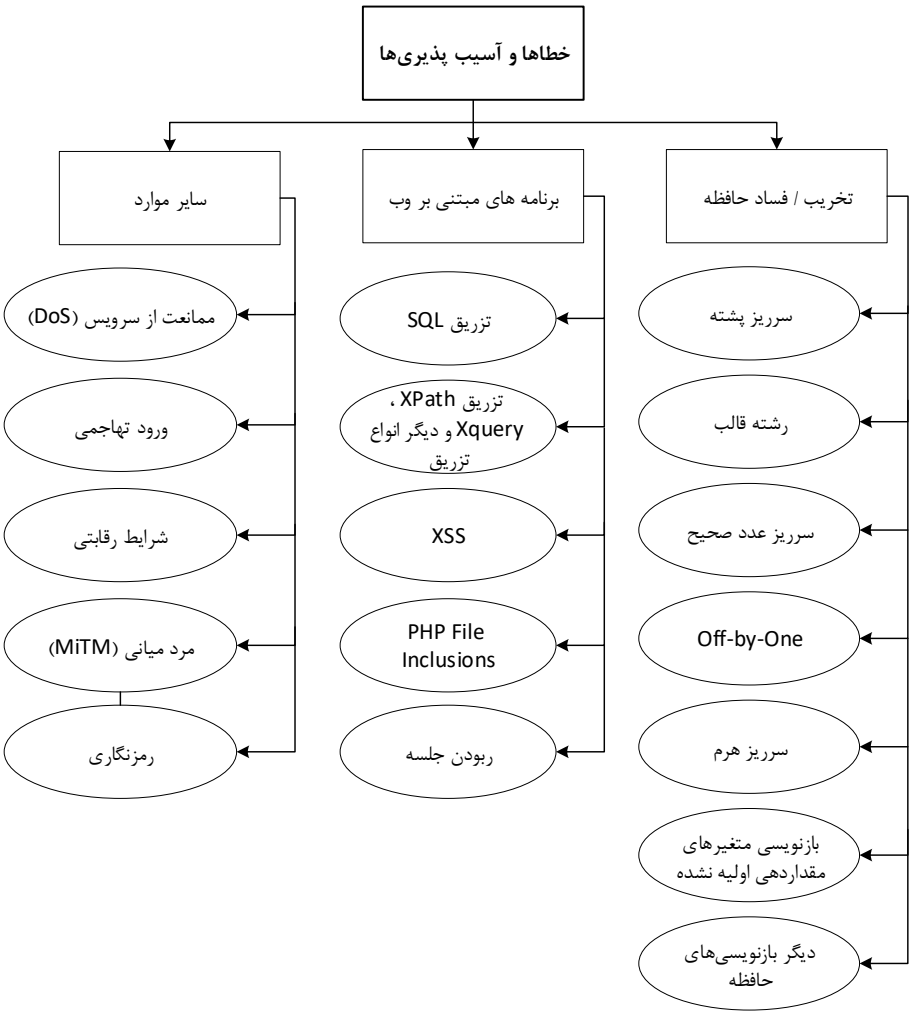
```
01.   $ ./uninitialized aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbCCCCDDDD
02.   String too long!
03.   $ ./uninitialized aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbCCCC
04.   $ ./uninitialized aaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbAAAA
05.   hi -- you should never see this, because logged_in is never
   set by program code.
```

توجه شود که کدی در برنامه فوق وجود دارد که هرگز مقداردهی اولیه نشده است (متغیر logged_in). البته در این مثال از آنجایی که بازنویسی مستقیمی اتفاق می‌افتد، این مقداردهی نشدن تقریبا بی‌ربط است ولی به هر حال در برخی موارد مقدار اولیه نداشتن سبب وجود آسیب‌پذیری می‌شود. در این مثال ساده، اگر رشته خاصی به عنوان ورودی به برنامه داده شود، عملیات داخلی برنامه به مسیری که نباید، تغییر می‌یابد. اینکه آیا بازنویسی یک متغیر همیشه قابل بهره‌برداری است یا خیر به برنامه وابسته است و نیازمند مقدار زیادی مهندسی معکوس و تحلیل جریان کد (جریان داده و جریان کنترل) است.

دیگر بازنویسی‌های حافظه. همان طور که در مورد سرریز پشته، هرم و عدد صحیح دیده شد، هر زمان فرد مهاجم بتواند حافظه داخلی زمان اجرای برنامه را به روشی ناخواسته دستکاری کند، اتفاقات بدی رخ می‌دهد؛ از جمله در دست گرفتن کنترل سیستم توسط فرد مهاجم. بنابراین تعجب انگیز نیست که اگر داده در بخش‌هایی از برنامه بتواند دستکاری شود یک حمله نیز احتمالا موفقیت آمیز است. بازنویسی اشاره‌گر تابع نیز گزینه مناسبی برای حمله به برنامه است. همچنین خواندن و نوشتن اطلاعات معتبر در بخش bss. و یا بخش data. می‌تواند جذاب باشد. نکته پایانی قابل توجه اینجاست که اگر نوشتن و خواندن دلخواه حافظه ممکن باشد، عواقب ناخوشایندی هم به تبع آن ممکن است پیش آید.

آسیب‌پذیری‌های برنامه‌های وب. برنامه‌های وب بسیار گسترده هستند و معماری خاص خود را دارند، بنابراین بدیهی است که آسیب‌پذیری‌هایی مختص خود داشته باشند. پرداختن به

آسیب‌پذیری‌های برنامه‌های وب خارج از محدوده این سمینار است. در شکل (۲-۷) آسیب‌پذیرهای برنامه‌های وب در کنار دیگر خطاها / آسیب‌پذیری‌های شرح داده شده در این بخش طبقه‌بندی شده‌اند.



شکل (۲-۷) طبقه‌بندی خطاهای شناخته شده با امکان بهره‌برداری از آنها.

۲-۵- یادگیری ژرف

یادگیری ژرف اصطلاحی است که اخیرا مرسوم شده است، گرچه از مدت‌ها قبل تحت عنوان‌های *سایبرنتیک*[۱] و *ارتباط‌گرایی*[۲] نزد محققین شناخته شده بود. یادگیری ژرف شاخه‌ای از یادگیری ماشین[۳] است. در فنون یادگیری ماشینی سیستم‌های هوش مصنوعی دانش خود را از طریق استخراج الگو[۴]ها از *داده‌های خام*[۵] کسب می‌کنند. یادگیری ماشینی جایگزین مناسبی برای رویکرد سنتی پایگاه دانش[۶] در حل مسائلی است که بیان آنها در قالب زبان و قواعد صوری (ریاضی و منطقی) مشکل یا غیر ممکن بوده[۷] و لذا منجربه پیدایش حوزه‌ای کاملا متفاوت در هوش مصنوعی شده است [18].

در این رهیافت به کامپیوتر اجازه داده می‌شود تا از تجربه‌های موجود آموزش ببیند و فضای مسئله را در قالب سلسله مراتبی از مفاهیم[۸] درک کند. هر مفهوم به‌وسیله ارتباطش با مفاهیم ساده‌تر تعریف می‌شود. با گردآوری دانش از طریق تجربه، این رویکرد نیاز به یک عامل انسانی را جهت بیان صوری تمامی دانش مورد نیاز برای حل مسئله، از میان بر می‌دارد. سلسله مراتب مفاهیم کامپیوتر را قادر می‌سازد تا مفهوم‌های پیچیده را با ساختن آنها از مفهوم‌های ساده‌تر یاد بگیرد؛ مانند ساختن یک تابع پیچیده از طریق ترکیب چندین تابع ساده. اگر چگونگی ترکیب این مفاهیم بر روی یکدیگر را در قالب یک گراف نشان دهیم، گرافی عمیق با لایه‌های زیاد حاصل می‌شود، به همین دلیل این رویکرد به هوش مصنوعی را یادگیری ژرف می‌نامند [18].

ابزار اصلی بیان و پیاده‌سازی یادگیری ژرف، شبکه‌های عصبی ژرف هستند. شبکه عصبی ژرف در واقع یک شبکه عصبی چندلایه است. به‌همین دلیل در ادامه با تعاریف و مفاهیم شبکه‌های

---

[۱] cybernetics
[۲] connectionism
[۳] machine learning
[۴] pattern
[۵] raw data
[۶] knowledge base
[۷] نظیر تبدیل گفتار به متن، تشخیص چهره و ...
[۸] concepts

عصبی ساده و نیز چندلایه آشنا خواهیم شد. پیش از آن اما لازم است تا انواع روش‌های یادگیری توسط ماشین را مرور کنیم. راهبردهای مختلفی برای یادگیری وجود دارد. سه راهبرد اساسی عبارتند از یادگیری بانظارت[۱]، یادگیری بدون نظارت[۲] و یادگیری تقویتی[۳] [26].

یادگیری با نظارت. این راهبرد یک ناظر[۴] یا راهبر را به خدمت می‌گیرد که از خود شبکه هوشمندتر است. برای مثال در تشخیص چهره، ناظر یک دسته تصویر را به شبکه نشان می‌دهد و نام هر چهره را می‌داند. شبکه پیش‌بینی خود را انجام می‌دهد و حدس خود را با پاسخ صحیح مقایسه می‌کند، سپس بر اساس میزان خطا، پارامترهای خود را تنظیم می‌کند تا به پاسخ درست نزدیک شود. در یادگیری با نظارت هر نمونه در مجموعه داده‌ها علاوه بر بردار ویژگی[۵]ها با یک برچسب[۶] یا هدف[۷] همراه است که نقش ناظر را ایفا می‌کند. طبقه‌بندی[۸] نمونه‌ای از این نوع است.

یادگیری بدون نظارت. این روش هنگامی که مجموعه داده نمونه با پاسخ معلوم در دسترس نباشد، نیاز است. به‌عبارت دیگر نمونه‌ها برچسب گذاری نشده باشند. جست‌وجو برای یافتن الگوی ناشناخته در بین یک مجموعه داده، کاربردی متداول از این روش است؛ برای مثال خوشه‌بندی[۹]، یعنی تقسیم یک مجموعه از عناصر به گروه‌هایی با توجه به الگوهای از قبل نامعلوم، به‌صورت بدون نظارت انجام می‌شود.

یادگیری تقویتی. این راهبرد بر اساس مشاهده و نظام پاداش و جزا است. برای مثال یک موش درون مارپیچ را در نظر بگیرد، چنانچه موش به سمت چپ بپیچد، یک تکه پنیر می‌گیرد. اگر به سمت راست بپیچد، یک شوک الکتریکی دریافت می‌کند (نگران نباشید این مثال واقعی نیست!) احتمالا موش در طول زمان می‌آموزد که در پیچ‌ها به سمت چپ بپیچد. شبکه عصبی موش یک

---

۱ supervised learning
۲ unsupervised learning
۳ reinforcement learning
۴ supervisor (teacher)
۵ feature
۶ label
۷ target
۸ classification
۹ clustering

تصمیم با یک خروجی (پیچیدن به چپ یا راست) می‌گیرد و محیطش را مشاهده می‌کند. اگر مشاهده منفی بود، شبکه خودش را برای گرفتن تصمیم متفاوت در زمان بعدی تنظیم می‌کند. یادگیری تقویتی در رباتیکز رایج است. در زمان t رُبـات وظیفه T را انجام می‌دهد و نتیجه آن روی محیط را مشاهده می‌کند. آیا به‌جایی برخورد کرد یا صدمه‌ای ندید؟

علاوه بر موارد فوق راهبردهای دیگری را نیز می‌توان نام برد از جمله راهبرد ***نیمه نظارتی***[۱] که در آن مجموعه داده که برای آموزش استفاده می‌شود، تلفیقی از نمونه‌های دارای برچسب و بدون برچسب است. راهبردهای یادگیری به‌صورت آنچه در بالا گفته شد اصطلاحاتی با تعریف رسمی نیستند و بسیاری از فنون یادگیری ماشینی می‌توانند برای همه انواع وظایف به‌کار روند. اهمیت یادگیری با نظارت در این است که می‌توان آن را مبنای ایجاد روش‌های یادگیری شبکه‌های عصبی و در نتیجه یادگیری ژرف قرار داد و سپس روش‌های حاصله را به حالت بدون نظارت نیز تعمیم داد. به همین دلیل این روش را با جزئیات بیشتری دنبال می‌کنیم.

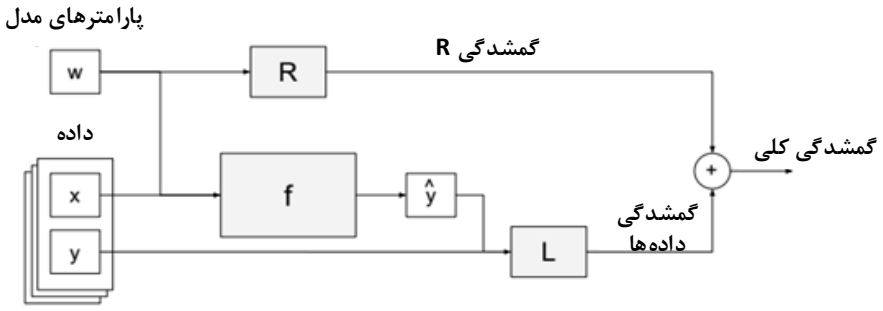
۲-۵-۱- یادگیری با نظارت

تقریباً هر مسئله محاسباتی با هر پیچیدگی را می‌توان در قالب یک تابع از ورودی‌ها به خروجی‌ها تحت یک ضابطه صورت‌بندی کرد که در آن یک کامپیوتر نیاز است تا نگاشت $f: X \to Y$ را که X فضای ورودی مسئله و Y فضای خروجی آن است انجام دهد. برای مثال در پردازش تصویر X می‌تواند فضای تصاویر و Y بازه پیوسته $[۰,۱]$ باشد که احتمال وجود گربه در تصویر ورودی را مشخص می‌کند. اگر قانون حاکم بر حل مسئله، یعنی ضابطه تابع، مشخص باشد نوشتن یک برنامه برای آن بسیار ساده است. متأسفانه، در خیلی از موارد تشخیص یک تابع معلوم و متداول آسان نیست. اما می‌توان یک مسئله را با بیشمار تابع تقریب زد. یادگیری با نظارت روش جایگزینی را پیشنهاد می‌کند. همان‌طور که پیش از این اشاره شد، در یادگیری با نظارت یک مجموعه داده شامل n داده نمونه $\{(x_1, y_1), \dots, (x_n, y_n)\}$ در دسترس است که $(x_i, y_i) \in X \times Y$. سه کمیت برای صورت‌بندی مدل یادگیری نیاز است [27]:

[۱] semi-supervised

۱. یک فضای جست‌وجو از توابع $\mathcal{F}$ که هر $f \in \mathcal{F}$ نگاشت $X \to Y$ را انجام می‌دهد.
۲. تابع عددی-مقدار گمشدگی[۱] $L(\hat{y}, y)$ که میزان اختلاف بین برچسب صحیح y و خروجی پیش بینی شده $\hat{y} = f(x)$ را مشخص می‌کند.
۳. تابع عددی-مقدار منظم‌سازی[۲] $R(f)$ که میزان پیچیدگی نگاشت را مشخص می‌کند.

شکل (۲-۸) جریان معمول داده در یادگیری با نظارت را نشان می‌دهد. f، L و R توابعی هستند که باید انتخاب شوند. w پارامترهای تابع f (مدل) است. هدف یافتن wای است که گمشدگی نهایی را کمینه کند. در یادگیری ژرف فضای توابع $\mathcal{F}$ یک شبکه عصبی با تعدادی پارامتر خواهد بود. تابع گمشدگی L معمولا گمشدگی اقلیدسی یا آنتروپی متقاطع است و تابع منظم‌سازی R در اکثر مواقع نورمِ L2 (جمع مجذورات همه وزن‌ها) است که در ادامه فصل توضیح داده خواهند شد. هنگامی که این توابع انتخاب شدند، مسئله یادگیری مدل به یک مسئله بهینه‌سازی کاهش می‌یابد. الگوریتم بهینه‌سازی و آموزش در یادگیری با نظارت مختص شبکه‌های عصبی نیست، در این سمینار اما مفاهیم یاد شده را صرفاً در حوزه شبکه عصبی بررسی می‌کنیم.



شکل (۲-۸) جریان داده در یادگیری با نظارت و چگونگی ارتباط بین سه تابع f، R و L [27].

[۱] loss function، همچنین به آن تابع هزینه (cost)، تابع خطا (error) و تابع عینی یا هدف (objective) هم می‌گویند.
[۲] regularization function

۲-۶- شبکه‌های عصبی

در بخش قبلی مفهوم یادگیری با نظارت مطرح شد و گفتیم که هدف پیدا کردن بهترین تابع است که نگاشت $X \to Y$ را برای هر نمونه از فضای ورودی انجام دهد. همچنین مفهوم این تابع را یک شبکه عصبی که پارامترهای آن در طول فرایند یادگیری تنظیم می‌شوند، در نظر گرفتیم. در این قسمت الگوریتم یادگیری و سایر موارد مربوط به شبکه‌های عصبی را تشریح می‌کنیم.

۲-۶-۱- معرفی

شبکه‌های عصبی مصنوعی یا به زبان ساده شبکه‌های عصبی، نوعی سیستم محاسباتی هستند که مفاهیم مطرح در آن، از شبکه عصبی زیستی موجود در مغز حیوانات از جمله انسان نشأت گرفته‌اند. ایده اصلی موجود پشت شبکه‌های عصبی در واقع به‌کار گرفتن مدل یادگیری سیستم عصبی انسان است. مغز انسان در تشخیص و پاسخ به الگوهای رفتاری مختلف مثل تشخیص تصاویر از قوانینی به شکل اگر-آنگاه پیروی نمی‌کند، بلکه از یک مدل تطبیقی[۱] که به‌تدریج از محیط پیرامون دریافت کرده است، در انجام این وظایف بهره می‌برد. برای مثال یک نوزاد در بدو تولد تفاوتی بین پدر و مادر (یا هر فرد یا شیء دیگری) قائل نیست ولی سیستم عصبی مغز وی به‌تدریج با دریافت این برچسب‌ها از سوی اطرافیان، به تمییز دادن آنها از یکدیگر سوق داده می‌شود. محققین هوش مصنوعی با الهام از این شیوه یادگیری، آن را برای حل مسائلی که برای آنها روش‌های مبتنی بر قانون اگر-آنگاه (که تا قبل از این در هوش مصنوعی بیشترین استفاده را داشت) امکان پذیر نبود به‌کار بستند [26].

امروزه رایج‌ترین کاربرد شبکه‌های عصبی در محاسبات، انجام وظایف «ساده برای انسان، سخت برای ماشین» است. اغلب از این وظایف به نام تشخیص الگو یاد می‌شود. کاربردهایی نظیر

[۱] adaptive

تبدیل عکس به نوشته (OCR)[۱]، ترجمه ماشینی(MT)[۲]، تبدیل گفتار به متن، تشخیص چهره، طبقه‌بندی و غیره [26].

۲-۶-۲- معماری و سازمان شبکه عصبی

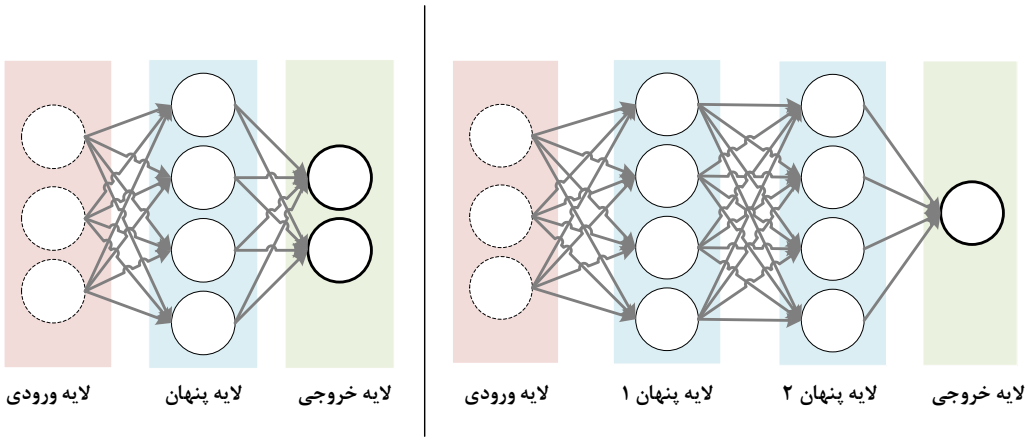
این امکان وجود دارد که شبکه‌های عصبی را بدون توجه به مفاهیم مغز، مطالعه کرد. شبکه عصبی همان‌طور که اشاره شد پیاده‌سازی یک تابع است. بدون هیچ پیش فرضی درباره ساختار ورودی x شبکه عصبی با تکرار ضرب ماتریسی و اعمال یک تابع غیر-خطی به‌صورت عنصر-محور قابل ساختن است. شبکه در این عنوان بدین معنا است که یک تابع (مفهوم) پیچیده در اینجا از ترکیب چند تابع (مفهوم) ساده‌تر ساخته می‌شود. هر تابع ساده‌تر در این شبکه یک لایه است. یک شبکه با دو لایه تابع $f(x) = W_2\sigma(W_1x)$ را پیاده می‌کند که W_1 و W_2 ماتریس و σ یک تابع غیر خطی (مثل tanh) است که روی تک تک عناصر ماتریس‌ها عمل می‌کند. یک شبکه سه لایه به فرم $f(x) = W_3\sigma(W_2\sigma(W_1x))$ خواهد بود[27] و به همین ترتیب (جزئیات بیشتر در ارتباط با نمایش ماتریسی شبکه عصبی در بخش ۲-۶-۴- بحث شده است). توجه شود که آخرین لایه شبکه (بیرونی‌ترین ضرب در شکل تابعی) شامل تابع غیر خطی نمی شود؛ زیرا، لایه آخر معمولا خروجی واقعی شبکه یا احتمال وقوع یک مجموعه خروجی را پیش‌بینی می‌کند و نیازی به غیر خطی سازی ندارد. همچنین با تعاریف گفته شده یک شبکه تک لایه یک انتقال خطی را پیاده‌سازی می‌کند. تابع غیر-خطی موسوم به تابع انگیزش[۳] نیز همانطور که در بخش ۲-۶-۶- بحث خواهد شد، مانع از خطی شدن فضای توابعی خواهد شد که شبکه عصبی نماینده آنها است.

می‌توان شبکه عصبی را به‌صورت یک گراف محاسباتی نمایش داد. از این منظر شبکه عصبی یک گراف جهت دار بدون دور (DAG)[۴] است[۵]. هر گره یک نورون و هر یال ارتباط بین دو نورون را مشخص می‌کند که جهت آن جهت انتقال اطلاعات و وزن آن میزان تأثیر اطلاعات را مشخص

[۱] optical character recognition
[۲] machine translation
[۳] activation function
[۴] directed acyclic graph
[۵] این شبکه را شبکه رو به جلو (feedforward network) یا وانیلی (vanilla network) نیز می‌نامند.

می‌کند. به‌عبارت دیگر خروجی برخی از نورون‌ها، ورودی برخی دیگر از آنها را تشکیل می‌دهند. دور در شبکه قابل قبول نیست، زیرا سبب یک حلقه بی‌نهایت در هنگام تولید خروجی شبکه می‌شود. البته همان‌طور که در بخش ۲-۷- خواهیم دید انواع مختلفی از شبکه‌های عصبی وجود دارند. در این بخش ساده‌ترین حالت مَد نظر است.

ســازمان لایه محور. به جای رسم حباب‌های درهم و بی‌نظم به‌عنوان نورون‌های شبکه عصبی، گراف شبکه عصبی معمولا در لایه‌های جداگانه از نورون‌ها سامان دهی می‌شوند. در اینجا هم هر لایه یک تابع را پیاده‌سازی می‌کند. در یک شبکه عصبی معمولی رایج‌ترین نوع اتصال لایه‌ها، لایه‌های کاملا متصل است که در آن نورون‌های دو لایه مجاور به‌صورت جفتی کاملا به یکدیگر وصل هستند، اما نورون‌های داخل یک لایه هیچ اتصالی به یکدیگر ندارند. به‌بیان ریاضی هر دو لایه مجاور یک گراف دوبخشی کامل[۱] را تشکیل می‌دهند. در ساده ترین حالت شبکه یک نورون تنها است (یک لایه خروجی متشکل از یک نورون). لایه‌های مابین لایه ورودی و خروجی را لایه پنهان می‌گویند. در صورتی که بیش از یک لایه پنهان در شبکه وجود داشته باشد، شبکه ژرف (عمیق) نامیده می‌شود. در شکل (۲-۹) دو شبکه عصبی اتصال کامل نشان داده شده است.



شکل (۲-۹) **چپ:** یک شبکه عصبی ۲-لایه (یک لایه پنهان متشکل از ۴ نورون یا واحد و یک لایه خروجی متشکل از ۲ نورون) به همراه سه عدد ورودی. **راست:** یک شبکه عصبی ۳-لایه (دو لایه پنهان هر کدام متشکل از ۴ نورون و یک لایه خروجی متشکل از یک نورون) و سه عدد ورودی.

[۱] complete bipartite graph

نام‌گذاری. توجه شود هنگامی که گفته می‌شود شبکه عصبی n لایه، لایه ورودی در نظر گرفته نمی‌شود؛ زیرا اساسا لایه ورودی نورون نیست. بنابراین شبکه عصبی تک لایه شبکه عصبی بدون لایه پنهان خواهد بود (ورودی(ها) مستقیما به خروجی(ها) نگاشت می‌شوند). می‌توان گفت رگرسیون لَجستیک[۱] یا ماشین بردار پشتیبان (SVM)[۲] نیز حالت خاصی از شبکه عصبی تک‌لایه هستند. همچنین به شبکه عصبی، پرسپترون چندلایه (MLP)[۳] و به نورون‌ها واحد[۴] هم می‌گویند.

اندازه شبکه عصبی. دو سنجه‌ای که معمولا برای بیان اندازه شبکه عصبی استفاده می‌شود عبارت است از تعداد نورون‌ها و تعداد پارامترها. برای شبکه‌های نشان داده در شکل (۲-۹) داریم:

- شبکه سمت چپ ۶ = ۴+۲ نورون (ورودی‌های نورون نیستند) و ۲۰ = [۴×۲]+[۴×۳] وزن و ۶ = ۴+۲ بایاس (بخش شکل (۲-۱۰)) دارد که در مجموع ۲۶ پارامتر برای یادگیری (تنظیم) در این شبکه وجود خواهد داشت.
- شبکه سمت راست ۹ = ۱+۴+۴ نورون، ۳۲ = [۴×۱]+[۴×۴]+[۴×۳] وزن، ۹ بایاس و در مجموع ۴۱ پارامتر قابل یادگیری دارد. بنابراین اندازه شبکه سمت راست بزرگ‌تر است.

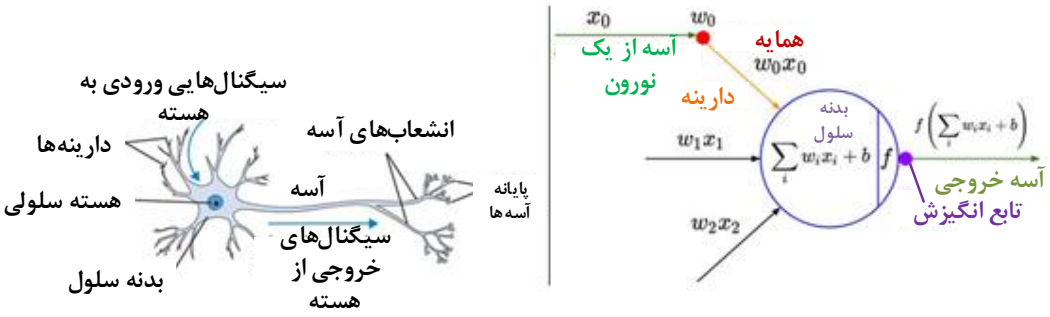
در عمل در وظایف حوزه یادگیری ژرف اندازه شبکه تا ۱۰۰ میلیون نورون و ۱۰ تا ۲۰ لایه پنهان (عمق بسیار زیاد) هم می‌رسد که بدین ترتیب قادر به پیاده‌سازی توابعی بسیار پیچیده و با پارامترهای بسیار زیاد خواهیم بود.

تفسیر بیولوژیکی. از نقطه نظر تاریخی، شبکه‌های عصبی از ساختمان نورون‌های عصبی الهام گرفته شدند. تقریبا ۸۶ میلیون نورون در سیستم عصبی انسان وجود دارد و این نورون‌ها توسط 10^{14} الی 10^{15} همایه[۵] به یکدیگر متصل هستند. شکل (۲-۱۰) یک طرح‌واره از نورون‌های زیستی (سمت چپ) را در کنار یک مدل ریاضی از یک نورون (سمت راست) نشان می‌دهد. هر نورون

---

۱ logistic regression
۲ support vector machine
۳ multi-layer perceptron
۴ unit
۵ synapse

سیگنال‌های (الکتریکی) ورودی را از خوشه دارینه[۱]های خود دریافت می‌کند و سیگنال خروجی را روی (تنها) آسه[۲] خود هدایت می‌کند. آسه در نهایت منشعب شده و توسط همایه‌ها، به دارینه سایر نورون‌ها متصل می‌شود و پیام عصبی منتقل می‌شود. در مدل محاسباتی مبتنی بر نورون سیگنال‌هایی که در طول آسه‌ها حرکت می‌کنند (برای مثال x_0 در شکل) به‌صورت ضربدر گونه (برای مثال w_0x_0) با دارینه‌های دیگر نورون‌ها بر مبنای قدرت سیناپتیک در محل آن آسه (برای مثال w_0) تعامل می‌کنند. ایده این است که قدرت سیناپتیک (وزن w) قابل یادگیری بود و قدرت و نیز جهت (وزن مثبت یا تحریک کنندگی و وزن منفی یا بازدارندگی) تأثیر یک نورون بر نورون دیگر را کنترل می‌کند. در یک مدل ابتدایی، دارینه سیگنال دریافتی را به داخل ماده سلولی حمل می‌کند، جایی که همه سیگنال‌ها با هم جمع می‌شوند. اگر مقدار جمع نهایی بالاتر از آستانه مشخصی باشد، نورون می‌تواند شلیک کند و یک سیگنال تحریک را روی آسه خود بفرستد. در مدل محاسباتی، ما فرض می‌نماییم که زمان دقیق تحریک مهم نیست و تنها بسامد (نرخ) تحریک اطلاعات را مبادله می‌کند. بر مبنای این تفسیر بسامد شلیک یک نورون را با یک تابع انگیزش f، مدل می‌کنیم. باید توجه داشت که سیستم عصبی زیستی بسیار پیچیده‌تر از مفاهیم فوق بوده و شبکه عصبی یک الهام از کلیات این سیستم است [26] و [27].



شکل (۲-۱۰) **چپ:** یک طرح‌واره از نورون زیستی. **راست:** مدل ریاضی - محاسباتی معادل آن.

---

[۱] dendrite

[۲] axon

۲-۶-۳- پرسپترون

تاکنون درباره معماری شبکه عصبی و معماری آن صحبت کردیم، اما عملکرد داخلی هر نورون و نیز عملکرد کلی شبکه بحث نشد. برای ورود به این مقوله از مفهوم ساده‌ای به نام پرسپترون[۱] آغاز می‌کنیم که توسط روزنبلات[۲] و با الهام از کار کُلاچ[۳] و پیتز[۴] توسعه داده شد [28].

یک پرسپترون دارای ساختمان ساده‌ای است. پرسپترون معمولا به‌صورت یک گره نمایش داده می‌شود که هر تعداد ورودی دودویی (صفر و یک) x_i به آن وارد شده و یک خروجی دودویی $\hat{y}$ را تولید می‌کند ($x_i, \hat{y} \in \{0,1\}$). هر ورودی توسط یک مقدار حقیقی ($w_i \in \mathbb{R}$) وزن می‌گیرد که در واقع میزان تـأثیر آن ورودی در تولید خروجی را بیان می‌کند (بعدا خواهیم دید این وزن(ها) همان مقدار(ها)ی هستند که در فرایند یادگیری شبکه عصبی تنظیم می‌شوند). وزن‌ها در واقع مفهومی شبیه به میانگین وزن دار را ایجاد می‌کنند. شکل (۲-۱۱) یک پرسپترون تنها را نشان می‌دهد. خروجی توسط قانون ساده‌ای تولید می‌شود: اگر جمع وزن دار همه ورودی‌های پرسپترون از یک آستانه مشخص بیشتر شد، خروجی برابر یک (پرسپترون شلیک می‌کند) و در غیر این صورت خروجی برابر صفر خواهد شد. این آستانه بایاس[۵] نامیده می‌شود و به‌صورت یک مقدار منفی تعریف می‌شود. بنابراین خروجی یک پرسپترون توسط رابطه (۲-۱) قابل بیان است.

$$\hat{y} = \begin{cases} 1, & if \sum_{i=0}^{n} w_i x_i + \mathrm{b} > 0 \\ 0, & otherwise \end{cases} \qquad (۱-۲)$$

اگرچه رابطه فوق بسیار ساده به نظر می‌آید، اما آن‌طور که خواهیم دید، هنگامی که تعداد زیادی از این واحدهای پرسپترونی بر روی یکدیگر و در لایه‌های مختلف پشته شوند (MLP)

---

۱ perceptron
۲ Frank Rosenblatt
۳ Warren S.McCullach
۴ Walter Pitts
۵ bias

(به‌صورتی که در شکل (۲-۹) نشان داده شده است) و هریک خروجی خود را از رابطه فوق تولید نمایند، می‌توانند قدرت تصمیم‌گیری قابل توجهی را بازنمایی کنند.

۲-۶-۴- نمایش ماتریسی MLP

وقتی صحبت از یک شبکه پرسپترونی چندلایه می‌شود، برای نمایش ورودی‌ها و وزن آنها، بایاس‌ها و درنهایت خروجی‌های پرسپترون‌ها در هر لایه، می‌توان از بردارها و ماتریس‌ها استفاده کرد. به این ترتیب امکان بیان عملیات به‌صورت ماتریسی فراهم می‌شود که نمایش و کار با آنها بسیار ساده‌تر خواهد بود. به عنوان مثال شبکه پرسپترونی دو لایه شکل (۲-۱۲) را در نظر بگیرید.

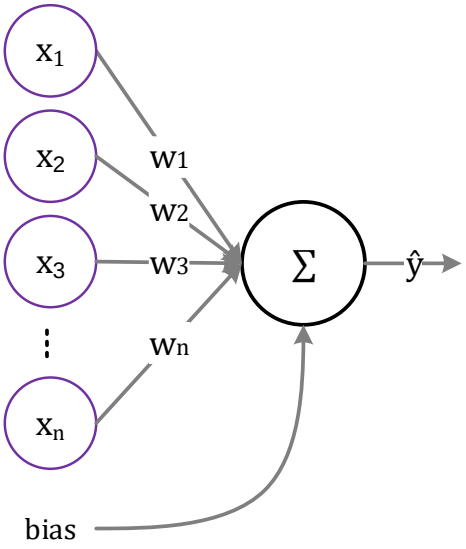
برای لایه اول، ورودی‌ها توسط یک بردار ستونی، وزن‌ها توسط یک ماتریس و مقادیر بایاس نیز توسط یک بردار ستونی قابل نمایش هستند:

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},\ W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix},\ b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

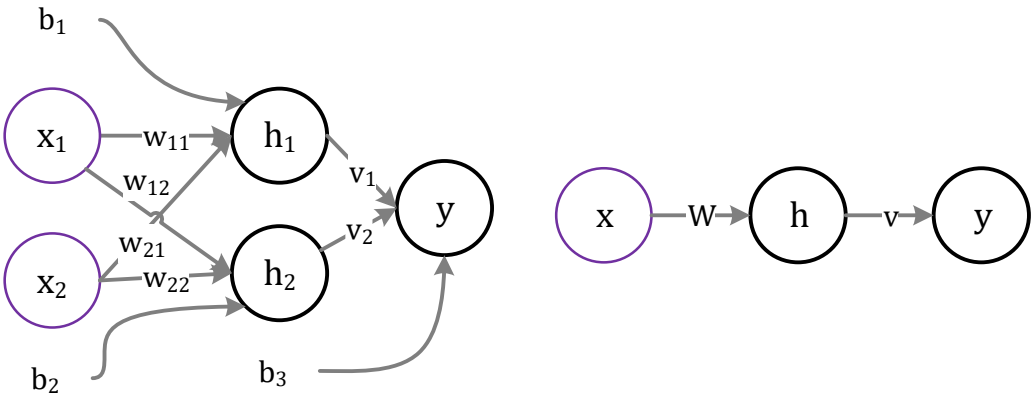
برای محاسبه خروجی پرسپترون‌ها در لایه اول (لایه پنهان) نیز خواهیم داشت:

$$h = W^T x + b \qquad (۲-۲)$$

$$= \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{21}x_2 + b_1 \\ w_{12}x_1 + w_{22}x_2 + b_2 \end{bmatrix} = \begin{bmatrix} h_1 > 0 ? 1 : 0 \\ h_2 > 0 ? 1 : 0 \end{bmatrix}$$



شکل (۲-۱۱) ساختار گرافی یک پرسپترون تنها دارای n ورودی x_i و خروجی $\hat{y}$.



شکل (۲-۱۲) یک شبکه پرسپترونی دولایه، دارای یک لایه پنهان با دو پرسپترون و لایه خروجی با یک پرسپترون،که به دو صورت مختلف نمایش داده شده است. خروجی هر پرسپترون داخل آن نوشته شده است. **چپ:** در این شکل هر پرسپترون با یک گره مجزا در گراف نشان داده می‌شود. این نمایش بسیار واضح و بدون هیچگونه گنگی است، اما برای شبکه با گره‌های بسیار زیاد فضای زیادی اشغال می‌کند. **راست:** در این شکل هر گره از گراف به عنوان یک بردار از مقادیر که همه خروجی‌های یک لایه را شامل می‌شود، در نظر گرفته شده است. همچنین برچسب هر یال مجموعه پارامترهای بین دو لایه را نشان می‌دهد. در اینجا W ماتریس نگاشت از x به h و بردار v نگاشت از h به y را نشان می‌دهد. این مدل نمایش فشرده‌تر بوده و برای نمایش شبکه‌های بزرگ مناسب‌تر است. دقت شود که برای سادگی مقادیر بایاس از شکل سمت راست حذف شده‌اند [18].

توجه شود که خروجی پرسپترون یک مقدار باینری است. به همین ترتیب برای محاسبه خروجی لایه دوم (خروجی نهایی شبکه) داریم:

$$y = v^T h + b \qquad (۳-۲)$$

$$= [v_1 \quad v_2]\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + b_3 = v_1 h_1 + v_2 h_2 + b_3 > 0 ? 1 : 0$$

نحوه نام گذاری‌ها و ستونی یا سطری بودن بردارها می‌تواند متفاوت باشد ولی در هرحال باید به خروجی صحیح منتهی شود. چنان‌چه مشاهده شد عملیات شبکه پرسپترونی با استفاده از جبر خطی به سادگی قابل بیان است.

به ازای هر دسته ورودی x_1 تا x_n و ضرایب متناظر با آن یعنی w_1 تا w_n و نیز مقدار b، پرسپترون یک خروجی را محاسبه می‌کند. این مرحله پیش بینی، استنتاج[۱] نامیده می‌شود [28]. در یک شبکه پرسپترونی جمیع این مقادیر به‌صورت ماتریسی در کنار هم قرار داده می‌شوند. همانطور که قبلا دیدیم ورودی و خروجی ما مشخص است، بنابراین هدف ما این است تا به ازای یک ورودی مشخص خروجی متناظر با آن را دریافت نماییم، برای این منظور باید مقادیر ضرایب و بایاس به‌طور مناسبی تنظیم شوند. فرایند تنظیم این مقادیر را آموزش (یادگیری) شبکه می‌نامیم.

۲-۶-۵- آموزش شبکه

هدف از آموزش شبکه تعیین مقادیر مناسب برای ضرایب و بایاس‌ها است. داده‌هایی که در فرایند آموزش شبکه استفاده می‌شود ***مجموعه آموزش***[۲] نامیده می‌شوند. داده‌هایی که برای ارزیابی عمومیت داشتن شبکه برای هر ورودی استفاده می‌شوند، ***مجموعه آزمون***[۳] نامیده می‌شوند و مجموعه سوم از داده‌ها که در طول فرایند آموزش شبکه برای انتخاب بهترین راهکارهای یادگیری

---

۱ inference

۲ training set

۳ test set

به‌کار گرفته می‌شوند، به نام ***مجموعه تأیید***[۱] شناخته می‌شوند. یک روش متداول برای بیان مجموعه داده استفاده از ***ماتریس طراحی***[۲] است. ماتریس طراحی یک ماتریس است که هر سطر آن شامل یک نمونه متفاوت از داده‌های ورودی است. هر ستون بدین ترتیب یک ویژگی از داده‌ها را شامل می‌شود. در یادگیری با نظارت که نمونه‌ها حاوی برچسب هستند برچسب‌ها را می‌توان به‌صورت یک بردار ستونی در مجاورت ماتریس طراحی در نظر گرفت که i امین سطر آن برچسب مربوط به نمونه i ام در ماتریس طراحی را نشان می‌دهد [18].

در مرحله آموزش، شبکه به ازای هر ورودی از مجموعه آموزش یک خروجی تولید می‌کند. از آنجایی که خروجی درست برای ورودی مربوطه در زمان آموزش دردسترس است (یادگیری با نظارت در بخش ۲-۵-۱- می‌توان به‌صورت کمی خطای مقدار محاسبه شده $\hat{y}$ و مقدار واقعی y را توسط تابع گمشدگی L، مانند تابع خطای مطلق میانگین (MAE)[۳]:

$$L_{MAE}(\hat{y}, y;\ W, b) = \frac{1}{n}\sum_{i=0}^{n}|\hat{y}_i - y_i|, \qquad (۴-۲)$$

خطای میانگین مربعات (MSE)[۴]:

$$L_{MSE}(\hat{y}, y; W, b) = \frac{1}{n}\sum_{i=0}^{n}(\hat{y}_i - y_i)^2, \qquad (۵-۲)$$

یا خطای آنتروپی متقاطع دودویی(BCE)[۵]:

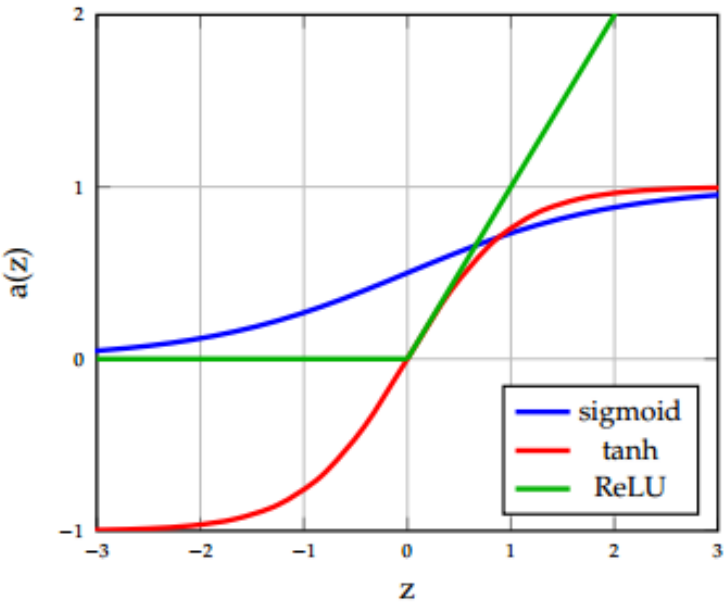
$$L_{BCE}(\hat{y}, y; W, b) = -\frac{1}{n}\sum_{i=0}^{n}(y_i \,.\log(\hat{y}_i) + (1-y).\log(1-\hat{y}_i)), \qquad (۶-۲)$$

[۱] validation set
[۲] design matrix
[۳] mean absolute error
[۴] mean squared error
[۵] binary cross-entropy

محاسبه کرد که در آن n تعداد نمونه‌ها در مجموعه آموزش است. البته توابع گمشدگی به‌صورت‌های دیگری نیز تعریف شده‌اند، اما در بسیاری از موارد سه تابع خطای فوق استفاده می‌شوند. در طول فرایند آموزش شبکه مقادیر W و b به نحوی تنظیم می‌شوند که مقدار تابع گمشدگی کمینه شود، یعنی:

$$f^* = arg \min_{W,b} L(\hat{y}, y; W, b) \qquad (۷-۲)$$

که f^* تابع یادگیری شده است و مقادیر W و b پارامترهای شبکه گفته می‌شود. سایر مقادیر را که در طول این فرایند یادگیری نمی شوند، ***ابَر*** پارامتر می‌نامند [28]. از جمله ابر پارامترها می‌توان به تعداد لایه‌ها و تعداد واحدها در هر لایه اشاره کرد، که از آنها برای تعیین اندازه شبکه استفاده می‌شوند (بخش ۲-۶-۲-). ابر پارامترهای دیگری در بخش‌های ۲-۶-۸- و ۲-۶-۱۰- معرفی خواهند شد.



شکل (۲-۱۳) نمودار توابع انگیزش متداول در شبکه‌های عصبی [28].

۲-۶-۶- نورون‌ها و انگیزش

مفهوم پرسپترون، شبکه پرسپترونی، نمایش و کلیت یادگیری آن مطرح گردید. در اینجا با یک مسئله بنیادی در ارتباط با پرسپترون مواجه هستیم. به منظور یافتن بهترین مقادیر ممکن برای پارامترها، باید تغییرات کوچکی در وزن‌های مدل (W) و نیز بایاس‌ها (b) انجام گیرد تا خروجی در جهت مطلوب سوق یابد. اما از آنجایی که خروجی پرسپترون گسسته است؛ یک تغییر کوچک در پارامترها می‌تواند منجربه معکوس شدن خروجی کلی مدل شود. برای غلبه بر این چالش و عمومیت بخشیدن به شبکه پرسپترونی، مفهوم نورون معرفی و پرسپترون با نورون جایگزین می‌شود. حال خروجی تک نورون با قاعده زیر محاسبه می‌شود:

$$\hat{y} = \Phi(a), \qquad a = \sum_{i=0}^{n} w_i x_i + b \tag{۸-۲}$$

که Φ یک تابع غیر-خطی موسوم به تابع انگیزش است. هدف اصلی تابع انگیزش پیوسته‌سازی و خروج از حالت خطی (غیر-خطی نمودن) مدل است. تابع انگیزش معمولا یکی از توابع سیگموید[۱] ($\sigma(a)$)، تانژانت هذلولوی ($\tanh(a)$) یا تابع یکسوساز (ReLU)[۲] انتخاب می‌شود [28]. شکل (۲-۱۳) این توابع را در کنار یکدیگر نشان داده است. تابع ReLU که بیشترین استفاده را در سال‌های اخیر پیدا کرده است به‌صورت $ReLU(z) = max(0, a)$ تعریف می‌شود. این تابع در عین سادگی و سرعت محاسبه، غیر خطی بودن شبکه را تضمین می‌کند و چندین مشکل عدیده موجود در دو تابع قبلی را برطرف کرده است. نشان داده شده است که این تابع سرعت همگرایی[۳] شبکه را تا ۶ برابر تسریع می‌کند [29].

در شبکه چندلایه خروجی لایه‌های پنهان با تابع انگیزش محاسبه می‌شوند ولی خروجی نهایی (لایه خروجی) همانطور که قبلا هم اشاره شد (بخش ۲-۶-۲-) نیازی به تابع انگیزش ندارد. به جای آن معمولا از یک تابع طبقه‌بندی کننده مانند ***تابع بیشینه هموار***[۴] استفاده

---

[۱] sigmoid

[۲] rectified linear unit

[۳] converge

[۴] softmax function

می‌شود. تابع بیشینه هموار یک بردار **k**-تایی از اعداد حقیقی را به عنوان ورودی دریافت نموده و یک بردار **k**-تایی از مقادیر حقیقی در بازه [۰,۱] را به عنوان خروجی می‌دهد به‌طوری که جمع مؤلفه‌های آن برابر یک خواهد بود:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=0}^{k} e^{x_j}} \; for\; i = 1\; to\; k \qquad (۹-۲)$$

۲-۶-۷- مقدار دهی اولیه

قبل از آغاز فرایند آموزش، پارامترهای شبکه باید مقدار دهی اولیه شوند. این عمل می‌تواند به وسیله تولید تصادفی نمونه‌هایی از توزیع گاوسی انجام شود. اما اگر این مقادیر بسیار کوچک یا بسیار بزرگ انتخاب شوند، ممکن است شبکه نتواند همگرا شود؛ یعنی سیگنال‌ها به‌قدری ضعیف می‌شوند که برای تولید خروجی مفید نخواهند بود یا بالعکس خروجی به‌صورت نابهنجار تغییر شکل می‌دهد. در نتیجه مقدار دهی اولیه خوب تأثیر زیادی در تسریع روند آموزش شبکه دارد.

یک روش مناسب موسوم به مقداردهی اولیه Xavier [30] است که از توزیع گاوسی با میانگین صفر و واریانس $var(W) = \frac{2}{n_{in}+n_{out}}$ استفاده می‌کند که n_{in} و n_{out} تعداد نورون‌ها در لایه قبلی و بعدی هستند. روش جدیدتر برای شبکه‌هایی با تابع انگیزش ReLU در [31] معرفی شده که واریانس را با ضریب ریشه دوم n که n تعداد ورودی‌ها است کالیبره می‌کند. برای مقدار دهی اولیه بایاس‌ها نیز معمولا از مقادیر بسیار کوچک یا صفر استفاده می‌کنند.

۲-۶-۸- پس‌انتشار

به‌منظور آموزش عملی شبکه با کمینه کردن تابع گمشدگی، الگوریتم یادگیری ***پس‌انتشار***[۱] بر شبکه اعمال می‌شود. این الگوریتم مبتنی بر ***کاهش گرادیان***[۲] است که به‌صورت تکراری تلاش

[۱] backpropagation
[۲] gradient decent

می‌کند تا کمینه محلی یک تابع را با برداشتن گام‌های کوچکی در جهت خلاف (منفی) جهت گرادیان آن تابع پیدا کند. اعمال این روش به تابع گمشدگی یک قانون بروزرسانی را برای هر وزن و نیز هر بایاس به‌دست می‌دهد [28]:

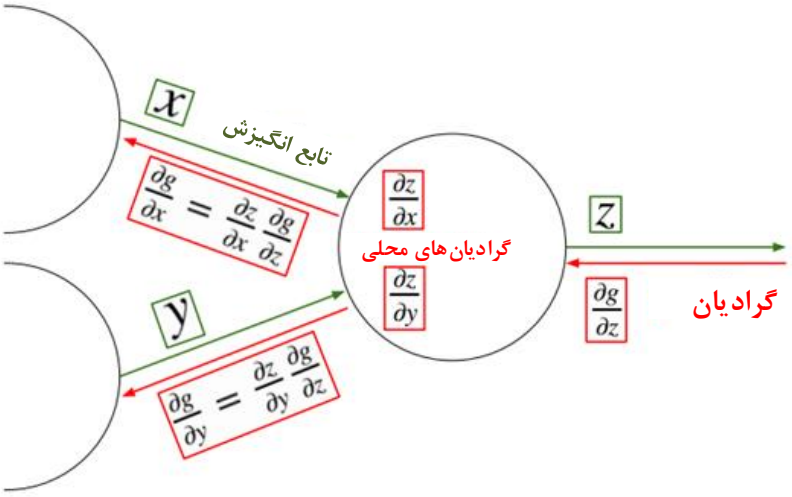
$$w_i^{(\tau+1)} = w_i^{(\tau)} - \ell.\frac{\partial L}{\partial w_i^{(\tau)}} \qquad (۱۰-۲)$$

$$b_j^{(\tau+1)} = b_j^{(\tau)} - \ell.\frac{\partial L}{\partial b_j^{(\tau)}} \qquad (۱۱-۲)$$

که در آن τ مهر زمانی و نماینده شماره تکرار بوده و $\ell > 0$ ضریب یادگیری است و مقدار تغییر در جهت شیب در هر تکرار را مشخص می‌نماید. به‌عبارت دیگر بعد از هر بار استنتاج و محاسبه مقدار L، در شبکه عقبگرد می‌کنیم و به آرامی مقدار هر پارامتر را در جهت کاهش خطا به مقدار جدید تغییر می‌دهیم. انجام یک مرحله مشخص با محاسبه گرادیان‌ها برای کل مجموعه آموزش زمان و فضا(حافظه)ی زیادی نیاز دارد. به همین خاطر گرادیان روی کل جمعیت با استفاده از یک نمونه کوچک‌تر تخمین زده می‌شود. این فن ***کاهش گرادیان تصادفی*** (SGD)[۱] نامیده می‌شود [28].

شکل (۲-۱۴) نحوه محاسبه گرادیان و پس‌انتشار آن در یک گراف محاسباتی را نشان می‌دهد. در کنار SGD که ساده ترین روش ممکن است، تعداد زیادی الگوریتم بهینه سازی مبتنی بر گرادیان وجود دارد. جزئیات بیشتری در [27] مطرح شده است.

[۱] stochastic gradient decent



شکل (۲-۱۴) یک مثال از الگوریتم پس‌انتشار روی یک گراف محاسباتی. در طول گذر جلو (استنتاج) x و y یک مقدار معین گرفته و بردار یا عدد خروجی z با استفاده از یک تابع ثابت (برای نمونه $z = x \odot y$) محاسبه می‌شود و مقدار گمشدگی کلی g حساب می‌شود. در گذر عقب (پس‌انتشار) قاعده مشتق زنجیری به‌صورت بازگشتی اعمال می‌شود تا تأثیر هر ورودی را بر خروجی نهایی گراف مشخص نماید. قاعده زنجیری بیان می‌کند که ابتدا باید گرادیان کلی g نسبت به z محاسبه شود و در گرادیان محلی هر ورودی ضرب شود و سپس به همین منوال رو به عقب ادامه پیدا کند تا به گرادیان نسبت به ورودی اولیه برسیم [27].

۲-۶-۹- شرایط توقف

فرایند آموزش می‌تواند تا بی‌نهایت ادامه داشته باشد. بنابراین بایستی یک قانون برای توقف آن تعریف شود. گزینه‌های زیادی وجود دارد که تحت چه شرایطی آموزش را خاتمه دهیم. همچنین ترکیب این شرایط نیز امکان‌پذیر است. برخی از این شرایط عبارتند از [28]:

- هنگامی که خطای مجموعه تأیید (برای تعداد مشخصی تکرار متوالی) کاهش پیدا نکند.
- هنگامی که تغییرات میزان خطا (برای تعداد مشخصی تکرار متوالی) کمتر از یک حد آستانه شود.
- هنگامی تعداد مشخصی ***دوره***[۱] طی شود (هر دوره برابر تعداد گام‌هایی است که برای مرور کل مجموعه آموزش مورد نیاز است).

[۱] epoch

- هنگامی که یک زمان مشخص از فرایند آموزش سپری شود.

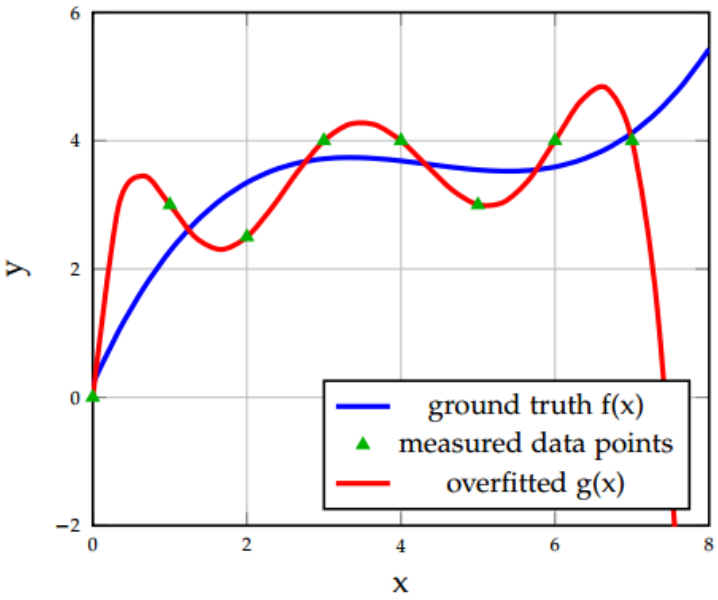
۲-۶-۱۰- منظم‌سازی

دربـاره دو تابع شبکه عصـبی f و گمشدگی L در یادگـیری بانـظارت صحبـت شد (بخش‌های ۲-۵-۱- و ۲-۶-۶-)، اما به تابع سوم یعنی تابع منظم‌سازی R کمتر پرداخته شد. یک چالش اساسی در یادگیری ماشینی این است که مدل باید بتواند نه تنها روی داده‌های مجموعه آموزش بلکه روی ورودی‌های جدید، که قبلا ندیده است یعنی برای مثال داده‌های مجموعه آزمون، هم خوب عمل کند. این قابلیت را ***تعمیم‌پذیری***[۱] یا کلیت‌بخشی گویند [18].

مشکل رایجی که هنگام آموزش شبکه عصبی باید جلوگیری شود اثر ***بیش‌برازندگی***[۲] است. بیش‌برازندگی یعنی حتی با کاهش خطا روی مجموعه آموزش خطای مجموعه‌های آزمون و تأیید به‌طور ناگهانی افزایش می‌یابد [18]. یک دلیل می‌تواند به اندازه کافی بزرگ نبودن اندازه مجموعه آموزش باشد. دلیل دیگر می‌تواند پیچیدگی مدل[۳] بسیار بالا باشد. پیچیدگی مدل معمولا با تعداد پارامترهای مدل مشخص می‌شود. در پیچیدگی بالا ممکن است مدل اختلال[۴]های موجود در ورودی‌ها را نیز لحاظ کند و متناسب آن شود، در نتیجه هنگام آزمون خطا بالا می‌رود [28]. برای مثال فرض شود می‌خواهیم تابع $g(x)$ را با تعدادی نقطه دارای اختلال نسبت به $f(x)$ برازش دهیم. هنگامی که مدل پارامترهای زیادی دارد، تبدیل به تابعی می‌شود که همه نقطه‌ها را کاملا برازش می‌دهد؛ متأسفانه همان طور که در شکل (۲-۱۵) نشان داده شده این تقریب خوبی برای تابع اصلی ما یعنی $f(x)$ نخواهد بود. از سوی دیگر کاهش پیچیدگی مدل ممکن است باز هم منجربه خروجی اشتباه شود؛ چرا که قدرت بازنمایی آن را کم می‌کند.

---

[۱] generalization
[۲] overfitting
[۳] model complexity
[۴] noise



شکل (۲-۱۵) یک نمایش از مسئله بیش‌برازندگی [28].

راهکارهای مختلفی برای مقابله با این مسئله پیشنهاد شده است. از این راهکارها تحت عنوان فنون منظم‌سازی یاد می‌شود. اغلب روش‌های شناخته شده پارامترهای دارای مقادیر بالا را که اثرات نوسانی شدیدی دارند، جریمه می‌کنند. معمولا تابع منظم‌سازی به‌صورت یک عبارت در به تابع گمشدگی اضافه می‌شود. این روش را ***فرسایش وزن***[۱] می‌نامند. روش منظم‌سازی نورمِ L2 برای هر وزن در شبکه عبارت $\frac{1}{2}\lambda w^2$ را به تابع گمشدگی اضافه می‌کند که در آن λ ضریب قدرت منظم‌سازی، یک ابر پارامتر، است.

روش دیگر برای منظم‌سازی dropout [32] بوده که بسیار مؤثر و ساده است. در طول آموزش dropout یک نورون را فقط با احتمال p (یک ابر پارامتر) فعال نگه می‌دارد و در غیر این صورت آن را صفر می‌کند. بنابراین موجب می‌شود تا شبکه اختلالاتی را که به تعمیم‌پذیری آن آسیب می‌زنند، یاد نگیرد.

[۱] weight decay

۲-۷- شبکه‌های عصبی مکرر

شبکه عصبی که تاکنون بحث شد ساده‌ترین شکل ممکن بود. به آن شبکه‌های رو به جلو یا وانیلی نیز می‌گویند. نارسایی شبکه‌های رو به جلو در حل مسائلی است که ترتیب ورودی در آن‌ها حائز اهمیت است، برای مثال در ترجمه ماشینی هر کلمه در یک جمله به کلمه(ها)ی قبلی خود وابسته است. در واقع ورودی ما به‌صورت یک ***توالی***[۱] $< x^{(1)}, x^{(2)}, \dots x^{(T)} >$ است. برای رفع این نارسایی‌ها مدل‌های مبتنی بر توالی یا همان *شبکه‌های عصبی مکرر معرفی* شدند [28].

RNNها کلاسی از شبکه‌های عصبی هستند که به‌صورت یک ***گراف جهت‌دار دوری*** بیان می‌شوند. به‌عبارت دیگر ورودی هریک از لایه(های) پنهان یا خروجی علاوه بر خروجی لایه قبل، شامل ورودی از مرحله قبل به‌صورت بازخورد نیز می‌شود. شکل (۲-۱۶) یک شبکه عصبی مکرر را نشان می‌دهد که در آن لایه پنهان از مراحل قبلی نیز بازخورد می‌گیرد. در هر مرحله‌زمانی t از ($t = 1$ تا $t = T$) یک بردار $x^{(t)}$ از توالی ورودی پردازش می‌شود. معادلات بروزرسانی (استنتاج) شبکه در t عبارتند از [18]:

$$a^{(t)} = Ux^{(t)} + Wh^{(t-1)} + b, \quad (۱۲-۲)$$

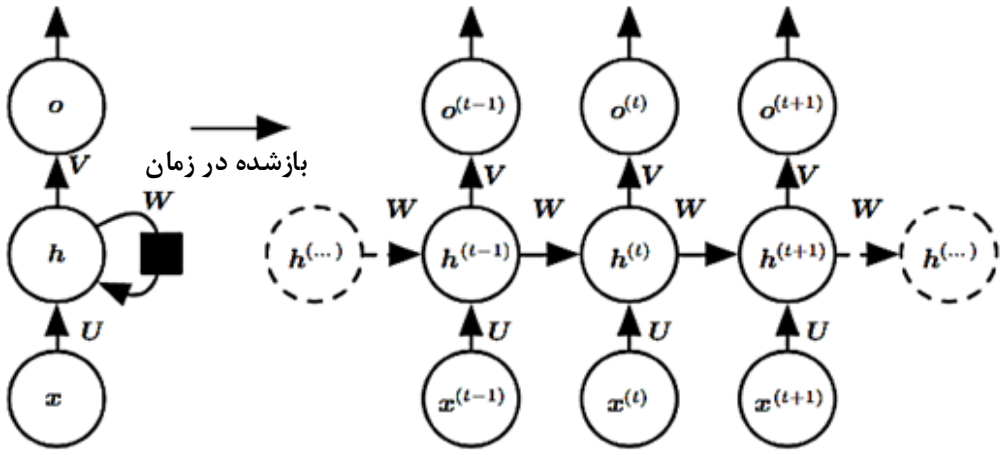
$$h^{(t)} = \Phi(a^{(t)}), \quad (۱۳-۲)$$

$$o^{(t)} = Vh^{(t)} + c, \quad (۱۴-۲)$$

$$\hat{y}^{(t)} = softmax(o^{(t)}), \quad (۱۵-۲)$$

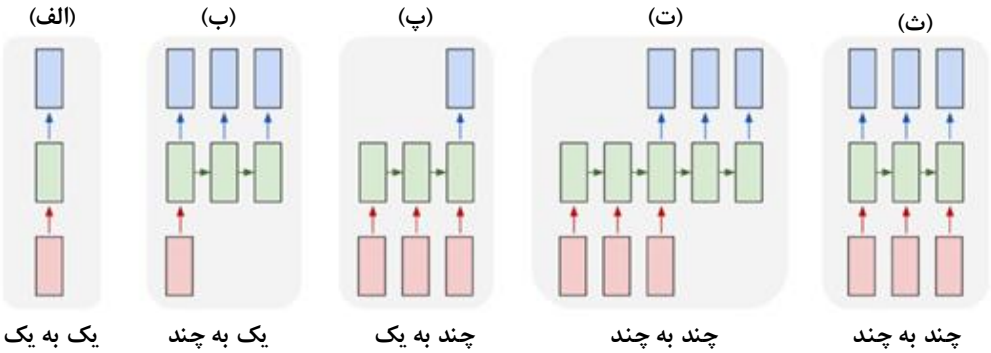
که در آن بردارهای b و c بایاس و ماتریس‌های U، V و W به‌ترتیب وزن یال‌های لایه ورودی به پنهان، پنهان به خروجی و پنهان به پنهان، پارامترهای شبکه هستند.

[۱] sequence



شکل (۲-۱۶) گراف محاسباتی مربوط به یک نوع شبکه عصبی مکرر که یک توالی ورودی از مقادیر x را به یک توالی خروجی از مقادیر o نگاشت می‌کند. فرض شده است که خروجی o احتمالات نرمال نشده است، بنابراین خروجی واقعی شبکه یعنی $\hat{y}$ از اعمال تابع بیشینه هموار روی o حاصل می‌شود. **چپ:** شبکه عصبی مکرر به‌صورت یال بازگشتی. **راست:** همان شبکه به‌صورت باز شده در زمان، به نحوی که هر گره با یک برچسب زمانی مشخص شده است [18].

در شکل (۲-۱۶) شبکه عصبی مکرر با یک لایه پنهان نشان داده شده است. اما می‌توان شبکه‌های مکرر عمیق با چندین لایه نیز داشت. همچنین طول توالی‌های ورودی و خروجی می‌تواند بسته به مسئله مورد نظر متفاوت باشد. در شکل (۲-۱۷) حالت‌های مختلفی از RNN از منظر توالی ورودی و خروجی نشان داده شده است [27].



شکل (۲-۱۷) طرح واره‌ای از حالت‌های مختلف شبکه عصبی مکرر. **(الف):** شبکه عصبی استاندارد، **(ب):** شبکه یک به چند، **(پ):** شبکه چند به یک، **(ت)** و **(ث):** شبکه‌های چند به چند [27].

۲-۷-۲- آموزش شبکه عصبی مکرر

الگوریتم پس‌انتشار برای آموزش شبکه عصبی مکرر هم قابل استفاده است. در واقع شبکه عصبی مکرر همان شبکه عصبی رو به جلو است که یک بازخورد از وزن‌های مرحله‌زمانی قبل می‌گیرد. در نتیجه خطا هنوز به‌صورت رو به عقب از مرحله‌زمانی t منتشر می‌شود. بسته به طول توالی ورودی و میزان منابع محاسباتی این انتشار تا مرحله‌زمانی $t = 1$ ادامه می‌یابد یا تحت محدودیت تعداد مشخصی گام متوقف می‌شود. در یک شبکه مکرر وزن مشترک W بین مراحل زمانی t و $t + 1$ داده شده است، داریم: $W^{(t)} = W^{(t+1)}$. متقابلا رابطه $\nabla W^{(t)} = \nabla W^{(t+1)}$ باید برآورده شود. این عمل می‌تواند با محاسبه گرادیان به‌طور مجزا برای هریک از مراحل زمانی صورت گیرد و نهایتا میانگین مجموع گرادیان‌ها به عنوان گرادیان در نظر گرفته شود [28]:

$$\frac{\partial L}{\partial W^{(t)}} + \frac{\partial L}{\partial W^{(t+1)}} \qquad (۱۶-۲)$$

این الگوریتم که می‌تواند به‌کل طول توالی ورودی اعمال شود، ***پس‌انتشار در زمان*** (BPTT)[۱] نام دارد. محدودیت منابع محاسباتی ایجاب می‌نماید که طول توالی ورودی مقدار مشخصی قرار داده شود یا پس‌انتشار در مرحله‌زمانی متوقف شود. بقیه مراحل الگوریتم مشابه بخش ۲-۶-۸- است.

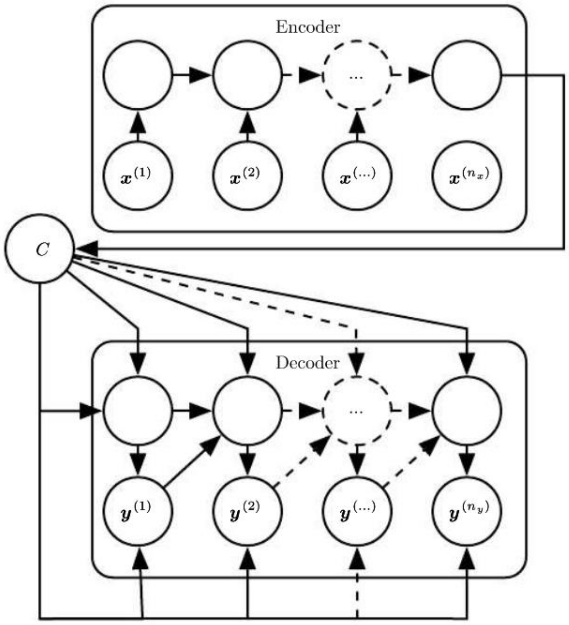
حافظه کوتاه‌مدت بلند. شبکه عصبی مکرر در حالت استاندارد، دارای نقطه ضعف اساسی است. در توالی‌های طولانی که وابستگی بین ورودی‌هایی با فاصله زیاد وجود دارد، این شبکه قادر به حفظ این وابستگی‌ها نیست به‌بیان دیگر گرادیان در دوره‌های زمانی بلند مدت تمایل به ناپدید شدن یا انفجار (زیاد شدن بی‌اندازه) دارد. در حالت ناپدید شدن گرادیان، یادگیری مدل متوقف می‌شود. مدل حافظه کوتاه‌مدت بلند (LSTM)[۲] برای مقابله با مشکلات فوق ایجاد شده است. هسته اصلی این مدل سلول حافظه c است که در آن بر خلاف شبکه مکرر استاندارد که خروجی هر واحد تنها با اعمال یک تابع انگیزش ایجاد می‌شود، خروجی با محاسبات پیچیده‌تری تعیین

[۱] backpropagation through time
[۲] long-short term memory

می‌شود که به‌ویژه هدف آنها منظم‌سازی و حفظ قدرت گرادیان است. توضیح جزئیات عملکرد این شبکه خارج از مبحث این سمینار است. امروزه به‌طور پیشفرض از LSTM در پیاده‌سازی شبکه عصبی مکرر استفاده می‌شود [33]. مطالعه جامعی روی RNN و انواع آن برای یادگیری توالی‌ها در [34] انجام شده است.

۲-۷-۳- مدل کدگذار-کدگشا

در شکل (۲-۱۷)ب نگاشت بردار ورودی با اندازه ثابت به یک توالی خروجی نشان داده شده است. همچنین در شکل (۲-۱۷)پ نگاشت یک توالی ورودی به یک بردار خروجی با اندازه ثابت نشان داده شده است. برای نگاشت یک توالی ورودی با طول متغیر به یک توالی خروجی با طول متغیر به‌نحوی که طول توالی‌های ورودی و خروجی الزاما یکتا نیست می‌توان از ایده ترکیب این دو شبکه با یکدیگر استفاده کرد (شکل (۲-۱۷)ت) [18].



شکل (۲-۱۸) مثالی از معماری شبکه عصبی مکرر در مدل کدگذار-کدگشا، که برای یادگیری تولید توالی خروجی $< y^{(1)}, \dots y^{(n_y)} >$ از روی توالی ورودی $< x^{(1)}, \dots x^{(n_x)} >$ به‌کار می‌رود [18].

چـــو[۱] و همکاران [35] یک مدل تحت عنوان کدگذار-کدگشا[۲] را معرفی کردند که از دو شبکه عصبی مکرر تشکیل شده است. یک شبکه کدگذار که یک توالی ورودی با بُعد متغیر را به یک نمایش بعد ثابت تبدیل می‌کند و یک شبکه کدگشا که یک توالی ورودی بُعد ثابت را می‌گیرد و به یک توالی ورودی با بعد متغیر تولید می‌کند. شبکه کدگشا توالی‌های خروجی را با استفاده از کاراکتر خروجی پیش‌بینی شده که در مرحله‌زمانی t به عنوان کاراکتر ورودی برای مرحله‌زمانی t + 1 تولید شده است، تولید می‌کند. یک بازنمایی از این معماری در شکل (۲-۱۸) نشان داده شده است. این معماری به ما اجازه می‌دهد تا یک توزیع شرطی را روی توالی خروجی بَعدی یاد بگیریم؛ یعنی، $p(< y^{(1)}, \dots y^{(n_y)} > | < x^{(1)}, \dots x^{(n_x)} >)$.

یک نمونه دیگر از مدل RNN کدگذار-کدگشا در [36] تحت عنوان روش توالی‌به‌توالی[۳] معرفی شده است. این مدل نتایج بسیار خوبی را در کاربردهایی نظیر ترجمه ماشینی و تشخیص گفتار (تبدیل گفتار به متن) نشان‌داده است. در فصل ســوم یک نمونه از کاربــرد این مدل در تولید خودکار داده آزمون فازی معرفی می‌شود.

۲-۸- نتیجه‌گیری

در این فصل مفاهیم ابتدایی آزمون نرم‌افزار مثل خطا، اشکال و خرابی مطرح شد. سپس آزمون فازی معرفی و معماری فازرها توضیح داده شد. فازرهای مبتنی بر قالب فایل به عنوان دسته‌ای مطرح از فازرها معرفی شدند و آسیب‌پذیری‌های قابل تشخیص توسط آنها با مثال‌هایی بیان شد. در ادامه مفاهیم مبنایی یادگیری ژرف و شبکه‌های عصبی و مدل‌های مولد بحث شد که می‌توانند برای تولید داده‌های آزمون به‌کار روند.

دو ویژگی مهم شبکه‌های عصبی عبارتند از تعمیم‌پذیری و تطبیق‌پذیری. تعمیم‌پذیری بدین معنی است که در صورت آموزش صحیح، شبکه برای ورودی‌های جدید نیز درست کار خواهد کرد. تطبیق‌پذیری یعنی در صورتی که داده‌ها تغییر کند، شبکه هم توانایی تغییر خواهد داشت.

---

[۱] Kyunghyun Cho

[۲] encoder-decoder

[۳] sequence-to-sequence

با توجه به افزایش قدرت محاسبات انجام حجم وسیعی از محاسبات در مسائل پیچیده، ارزان‌تر از نوشتن یک الگوریتم خاص می‌باشد.

شبکه‌های عصبی همچنین معایبی نیز دارند. از جمله اینکه آموزش آنها سخت و بسیار مستعد خطا است. در واقع دقت نتایج بستگی زیادی به مجموعه آموزش دارد؛ به نحوی که یک مجموعه آموزش ضعیف (کوچک یا نادرست) عملا شبکه غیرقابل استفاده‌ای را نتیجه می‌دهد. دیگر آنکه قوانین مشخصی برای طراحی یک شبکه جهت کاربردی خاص وجود ندارد و بلأخره اینکه معمولا نمی‌توان به فیزیک یا قانون حاکم بر مسئله حل‌شده توسط شبکه پی برد. در فصل بعد برخی کارهای مرتبط با مباحث این فصل بیان خواهد شد.

فصل ۳: مروری بر کارهای مرتبط

یک شب تاریک و طـوفانی بود. پاییز ۱۹۹۴، در آپارتمان خود در مادیـسون نشسته بودم. آن شب من از طریق خط تلفن به سیستم‌های یونیکس دانشگاه متصل شدم. با بارش سنگین باران اختلال زیادی روی خط اتصالی بود که در اجرای فرمان‌های ارسالی من دخالت می‌کرد. رقابتی بین سرعت نوشتن یک فرمان قبل از آنکه اختلال آن را خراب کند وجود داشت. چیزی که مرا شگفت زده می‌کرد این حقیقت بود که اختلال سبب ایجاد خرابی و سقوط برنامه‌ها می‌شد. و شگفت‌آورتر برنامه‌هایی بود که سقوط می‌کرد: ابزارهای رایج یونیکس!

دکتر بارتون میلر، مبدع آزمون فازی

۳-۱- مقدمه

در این فصل پیشینه برخی از کارهای انجام شده روی آزمون فازی قالب فایل بیان می‌شود. تمرکز اصلی بر روی روش‌های تولید داده استفاده شده در فازرهای مختلف است؛ زیرا، مهمترین جنبه آزمون فازی را تولید مورد آزمون تشکیل می‌دهد. روش‌های ذکر شده لزوما منحصر به فازرهای قالب فایل نیستند؛ به عنوان مثال در آزمون فازی پروتکل‌های ارتباطی شبکه نیز می‌توان برخی از این روش‌ها را استفاده کرد. بنابراین تنها هرجا که لازم بوده به‌عبارت قالب فایل اشاره شده است.

در ابتدا پیشینه‌ای از آزمون فازی و طبقه‌بندی جزئی‌تر روش‌های تولید داده بیان می‌شود. در ادامه روش‌های فازی مبتنی بر بازخورد و آگاه از برنامه مطرح می‌گردد. سپس روش‌های استخراج و درک گرامر ورودی، به‌خصوص روش مبتنی بر شبکه‌های عصبی به‌طور مفصل‌تری شرح داده می‌شود. مبانی نظری این شبکه‌ها در فصل قبل عنوان شدند و در این فصل کاربرد آنها برای تولید داده آزمون مطالعه شده است. در پایان فصل نیز خلاصه و نتیجه‌گیری مطالب آمده‌است.

۳-۲- پیشینه آزمون فازی

آزمون فازی نخستین بار توسط دکتر میلر[۱] و همکاران در دانشگاه ویسکانسین-مدیسن برای آزمون تعدادی از ابزارهای سیستم‌عامل یونیکس طراحی و اجرا شد [3]. ایده اولیه از این مشاهده نشأت گرفت که برخی ابزارهای رایج در یونیکس وقتی توسط یک اتصال دارای اختلال خط تلفن دستیابی می‌گردند، ناگهان سقوط می‌کنند[۲]. در این پژوهش یک ابزار ساده به‌نام ***فاز***[۳]، برای تولید رشته‌های ورودی تصادفی، که از طریق یک خط‌لوله یا شبه-پایانه قابل تزریق به SUT بودند، پیاده‌سازی شد. این فن سبب ایجاد خرابی و سقوط بیش از ٪۲۴ از ۸۸ ابزار مرسوم یونیکس

[۱] Barton P. Miller

[۲] crash

[۳] fuzz، واژه فاز در فارسی «گیج‌کردن» نیز معنی شده است به همین ترتیب فازر را «گیج‌ساز» می‌توان نامید. به‌دلیل اینکه این واژه‌ها کمتر مصطلح بوده در این سمینار استفاده نشده‌است. همچنین مفهوم فازی در اینجا با منطق‌فازی هیچ ارتباطی ندارد.

آزمون‌شده در طول آزمایش، گردید [2]. از آن پس آزمون فازی برای آزمون طیف وسیعی از نرم‌افزارها و برنامه‌های سیستمی و کاربردی از جمله مرورگرهای وب، چندرسانه‌ای‌ها، کامپایلرها و غیره، به‌کار گرفته‌شد و تا امروز پژوهش‌های متعددی در زمینه بهبود و تقویت آن صورت پذیرفته‌است.

اولین فازرهای ساخته‌شده رویکردی مبتنی بر جعبه سیاه داشتند. برخلاف ساده و مستقل از برنامه بودن، این دسته از فازرها مانند Peach، Sulley و Radamsa توانسته‌اند خطاهای بسیاری را در برنامه‌های کاربردی کشف کنند که تا قبل از این نهفته مانده‌بود [9]. به‌تدریج فازرها در دو رویکرد جعبه سفید [37] و جعبه خاکستری [19] هم مورد توجه قرار گرفتند. به موازات آن روش‌های مختلفی برای تولید خودکار داده آزمون در فازرها استفاده شد؛ به‌نحوی که می‌توان گفت آنچه یک فازر را از فازر دیگر متمایز می‌کند، روش به‌کار رفته در آن برای تولید داده آزمون است. در بخش‌های آتی تلاش شده تا مهمترین روش‌ها به‌صورت طبقه‌بندی‌شده و ضمن رعایت سیر تاریخی آنها ارایه گردند. قبل از آن اما مختصری به آسیب‌پذیری‌های رایج کشف‌شده قالب‌های فایل می‌پردازیم.

جدول (۳-۱) انواع رایج برنامه‌های آسیب‌پذیر و مثال‌هایی از آسیب‌پذیری‌های کشف شده در قالب فایل‌های آنها [1].

برنامه کاربردی	نام آسیب‌پذیری	پیوند وب
محصولات مجموعه Office	آسیب‌پذیری سرریز میانگیر Microsoft HLINK.DLL Hyperlink Object Library	*https://www.cvedetails.com/vulnerability-list/vendor_id-26/product_id-8076/Microsoft-Hyperlink-Object-Library.html*
ضدویروس	آسیب‌پذیری سرریز میانگیر پویشگر فایل CHM موتور ضدویروس kaspersky	*https://vulners.com/osvdb/OSVDB:19912*
پخش‌کننده چندرسانه‌ای	آسیب‌پذیری سرریز پشته پویشگر Winamp m3u	*http://www.exploitalert.com/view-details.html?id=14292*

۳-۲-۲- آسیب‌پذیری‌های کشف‌شده در قالب‌های فایل

تعداد زیادی آسیب‌پذیری در پویشگرهای قالب فایل سمت مشتری[۱] در فاصله سال‌های ۲۰۰۵ و ۲۰۰۶ و تعداد دیگری نیز در سال اخیر کشف شدند. بسیاری از آنها آسیب‌پذیری‌های روز-صفر بودند. تعداد زیادی آسیب‌پذیری برای قالب فایل وجود دارد و همچنین راه‌های گوناگونی برای بهره‌برداری از آنها می‌توان یافت. یک سناریو حمله می‌تواند ارسال یک فایل مخرب از طریق مرورگر وب به کاربر و در اختیار گرفتن صفحه متعلق به وی باشد. تعدای از آسیب‌پذیری‌های تشخیص داده شده در قالب فایل‌های مختلف در جدول (۳-۱) آمده‌است [1].

۳-۳- سازوکارهای تولید داده

در بخش ۲-۳-۳- دو روش اصلی استفاده‌شده برای تولید خودکار داده آزمون در فازرها، روش مبتنی بر جهش و مبتنی بر تولید، معرفی شد. از آنجا که هدف اصلی در آزمون فازی تزریق یک ورودی ناخواسته (معمولا نیمه-معتبر) به SUT است، سازوکارهایی برای ایجاد داده آزمون جدید وجود دارد که در هر دو روش به‌کار برده می‌شوند. به‌عبارت دیگر این موارد که در این نوشتار از آن به سازوکار[۲] یاد می‌کنیم میزان هوشمندی به‌کار گرفته شده در روش مورد نظر را بیان می‌کنند. در زیر مهمترین این سازوکارها شرح داده شده است [13] و [14].

سازوکار تصادفی محض. فازرهای اولیه مبتنی بر روش تصادفی بودند. این فازرها به نوع و ساختار اطلاعات ورودی SUT دقتی نداشتند. در نوع مبتنی بر جهش، فازر تعدادی بایت از یک مورد آزمون موجود را به‌صورت تصادفی انتخاب و آنها را با مقادیر تصادفی جایگزین می‌کند. در نوع مبتنی بر تولید، فازر یک جریان تصادفی از بایت‌ها را تولید می‌کند. ابزار فاز [3] از این نوع است.

---

[۱] client-side

[۲] mechanism

سازوکار قالب‌محور. در این سازوکار فازر اطلاعات حداقلی از ساختار ورودی SUT دارد. این اطلاعات که شامل محل و نوع (تعدادی از) اجزای یک ورودی معتبر است؛ داخل یک قالب[۱] ذخیره می‌شود. وقتی SUT یک ورودی با ساختار دقیق را می‌پذیرد، استفاده از قالب ورودی‌هایی را تولید می‌کند که امکان نفوذ به مسیرهای عمیق‌تر را فراهم می‌نماید. فازر xwinjig [4] یک فازر قالب‌محور است که هر دو روش مبتنی بر جهش و مبتنی بر تولید را پشتیبانی می‌کند.

سازوکار گرامر محور. این سازوکار یک گرامر که حداقل بخش‌هایی از زبان ورودی SUT را پوشش می‌دهد، نیاز دارد. فازر معرفی شده در [38] از گرامر برای تولید کد و آزمون کامپایلرها استفاده می‌کند. این سازوکار در روش مبتنی بر تولید به‌کار می‌رود؛ اما قابل اعمال به روش‌های مبتنی بر جهش نیز هست.

سازوکار اکتشاف‌محور. این سازوکار روش‌های جست‌وجوی اکتشافی[۲] و تکاملی موجود در هوش مصنوعی را وارد حوزه آزمون فازی کرده‌است. به‌طور معمول از خروجی اثر یک مورد آزمون در تولید مورد آزمون بعدی بازخورد گرفته می‌شود. این بازخورد شامل میزان خوب بودن ورودی آزمون قبلی است. سیستم فازینگ تکاملی (EFS)[۳] [19] از الگوریتم ژنتیک برای جهش داده‌ها از یک مجموعه داده تصادفی اولیه، استفاده می‌کند. AFL[۴] [39] نیز از الگوریتم‌های تکاملی استفاده می‌کند. به‌طور دقیق‌تر AFL هر مورد آزمونی را که یک مسیر جدید کشف کند، نگهداری کرده و آن را برای تولید مورد آزمون بعدی جهش می‌دهد.

۳-۴- روش‌های آگاه از برنامه

حلقه بازخوردی که در سازوکارهای اکتشاف‌محور به‌کار می‌رود تلاشی در راستای مرتبط ساختن رفتار SUT با ساختار ورودی، به منظور بهبود تولید داده نمی‌کند. در واقع همچنان ورودی بدون اطلاع از ساختار و نیز SUT، در حال جهش است و کشف مسیرهای جدید تنها با خوش‌شانسی

---

[۱] template

[۲] heuristic، در فارسی «آوینی» هم ترجمه شده‌است.

[۳] evolutionary fuzzing system

[۴] american fuzzy lop

صورت می‌گیرد. با این وجود سازوکار اکتشافی جدیدتر بوده و تلاش‌های اخیر معطوف به استفاده از آن بوده‌اند.

اخیرا رویکردهای *اجرای نمادین*[۱] و *اجرای واقعی-نمادین*[۲] (اجرای واقعی توأم با اجرای نمادین) در حوزه ***فازینگ هوشمند***[۳] موفقیت خوبی کسب کرده‌اند. Mayhem [40] از CMU[۴] با استفاده از چندین فن تحلیل برنامه شامل فنون فوق، قادر به کشف خودکار آسیب‌پذیری در کد دودویی است. Driller [41] نیز با توسعه AFL، از فنون ترکیبی برای حل محدودیت‌های انشعاب جهت کشف مسیرهای عمیق استفاده می‌کند. این روش‌ها در عین حال نیازمند تحلیل سنگین و زمانبر کد هستند. علاوه بر این در حالی که ترکیب فازینگ و اجرای نمادین رضایت بخش بوده و زمینه تحقیقاتی مورد علاقه‌ای محسوب می‌شود، این فنون با چالش اساسی مقیاس‌پذیری مواجه هستند. یعنی برای برنامه‌های خیلی بزرگ با ورودی‌های پیچیده مناسب نیستند.

دو معیار اطلاع از برنامه و مقیاس‌پذیری فازرها در تضاد با یکدیگر هستند. ابزار VUzzer [9] یک فازر تکاملی آگاه از برنامه است که نیاز به هیچگونه دانش قبلی از برنامه و قالب ورودی آن ندارد. VUzzer به‌منظور دستیابی به پوشش کد بیشینه و کشف مسیرهای عمیق، از ویژگی‌های جریان کنترلی و جریان داده برنامه که با تحلیل‌های ایستا و پویا حاصل می‌شود، بهره می‌گیرد. این امکان سبب می‌شود تا ورودی در مقایسه با فازرهای قبلی سریع‌تر نیز تولید شود. به‌طور کلی VUzzer یک فازر مبتنی بر قالب فایل با رویکرد مبتنی بر جعبه خاکستری، مبتنی بر جهش و مبتنی بر پوشش[۵] است.

---

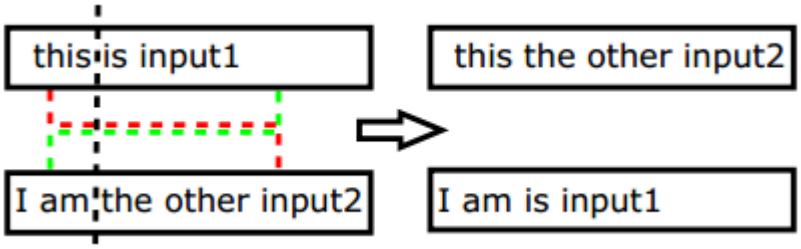
[۱] symbolic

[۲] رویکرد موسوم به concolic که ترکیب اجرای concrete و symbolic است.

[۳] smart fuzzing

[۴] Carnegie Mellon University

[۵] coverage-based



شکل (۳-۱) عملیات تقاطع در VUzzer [9].

تولید داده ورودی در VUzzer طبق الگوریتم ژنتیک از دو بخش تشکیل می‌شود: تقاطع[۱] و جهش. تقاطع یک عمل ساده است که در آن دو والد از جمعیت قبلی انتخاب شده و دو فرزند جدید ایجاد می‌نماید. شکل (۳-۱) عمل تقاطع را با یک مثال ساده توضیح می‌دهد.

جهش یک عملیات نسبتاً پیچیده‌تر است که شامل چهار ریزعمل در جهت تغییر ورودی قدیم به جدید است. این عملیات در [9] شرح داده شده است. در مجموع عملیات شامل جایگذاری مقادیر مرزی در قسمت‌های اعداد صحیح و تغییر عملوندهای مقایسه‌ای و نهایتا جایگذاری بایت‌های جادویی در آدرس‌های نسبی درست خود است که در مرحله تحلیل حاصل شده است.

۳-۵- روش‌های استخراج و درک گرامر

سازوکار گرامر محور در تولید داده آزمون بحث شد. کار بر روی تولید داده آزمون خودکار مبتنی بر گرامر در دهه ۱۹۷۰م. شروع گردید و با آزمون مبتنی بر مدل در ارتباط نزدیک هست [8]. بیشترین روش‌های مبتنی بر تولید در آزمون فازی گرامر محور هستند. این روش همچنین با آزمون فازی جعبه سفید ترکیب شده است. روش‌های مبتنی بر جهش با وجود استفاده از بازخورد و دیگر فنون معرفی شده در بخش قبلی، همچنان برای ورودی‌های با ساختار بسیار پیچیده دارای نارسایی پوشش عمیق کد هستند. در واقع ورودی تولید شده توسط این روش‌ها ممکن است در مراحل ابتدایی پردازش توسط پویشگر رد شود. فازینگ گرامر محور برای این شکل از ورودی‌ها خیلی مناسب به‌نظر می‌رسد؛ اما به‌طور کامل خودکارسازی نشده است. تلاش‌هایی در

[۱] crossover

جهت خودکارسازی درک و استخراج گرامر و سپس تولید ورودی آزمون از روی آن انجام شده که در ذیل اشاره می‌شود.

بستانی[۱] و همکاران [42] الگوریتمی برای تولید یک گرامر مستقل از متن روی یک مجموعه از ورودی‌های نمونه داده شده ارایه کرده‌اند، که در نهایت برای تولید داده‌های جدید مورد نیاز آزمون فازی استفاده می‌شود. این الگوریتم یک مجموعه از مراحل تعمیم‌پذیری را با معرفی ساختارهای تکراری و متناوب برای عبارت‌های منظم به‌کار می‌بندد و غیر پایانه‌ها را برای گرامر مستقل از متن در هم ادغام می‌نماید که به‌نوبه خود یک گرامر یکنواخت از زبان ورودی به‌دست می‌دهد؛ اما، این روش برای قالب‌هایی مثل PDF که ساختار نسبتا مسطح (غیر تو در تو) ولی در عین حال محتوای مختلفی از انواع و جفت‌های کلید-مقدار دارند، مناسب نیست.

AUTOGRAM [43] نیز به‌صورت غیر-احتمالاتی یک گرامر مستقل از متن را یاد می‌گیرد. یک مجموعه ورودی داده‌شده و به‌صورت پویا مشخص می‌شود که چگونه ورودی‌ها در برنامه پردازش می‌شوند. در واقع برنامه تحت آزمون با آلودگی پویا[۲] مشاهده می‌شود که حافظه را با قطعات ورودی که از آنها می‌آیند، برچسب‌گذاری می‌کند. بخش‌هایی از ورودی‌ها که توسط برنامه پردازش می‌شود، نهادهای نحوی در گرامر می‌شوند. Tupni [44] یک سیستم دیگر است که قالب ورودی را از روی نمونه‌های آن مهندسی معکوس می‌کند. برای این کار از سازوکار ردیابی آلودگی[۳] استفاده می‌کند که داده‌ساختارها را با آدرس فضای آدرس برنامه مرتبط می‌کند.

۳-۵-۱- تحلیل برنامه مبتنی بر شبکه‌های عصبی

در پژوهش‌های اخیر تمایل زیادی به استفاده از شبکه‌های عصبی برای تحلیل و تولید برنامه‌ها به‌وجود آمده‌است. چندین معماری عصبی برای یادگیری الگوریتم‌های ساده‌ای مثل مرتب‌سازی و کپی‌برداری آرایه ارایه شده‌اند. Neural FlashFill معماری‌های عصبی جدیدی را برای

---

[۱] Osbert Bastani
[۲] dynamic taint
[۳] taint

کدگذاری نمونه‌های ورودی-خروجی و تولید برنامه‌های مبتنی بر عبارت‌های منظم در زبان‌های-خاص‌دامنه[۱] استفاده می‌کند. چندین مدل کدگذار-کدگشا برای یادگیری به‌منظور تصحیح خطاهای نحوی در برنامه‌ها توسعه داده شده‌اند. این فنون یک مدل کدگذار-کدگشا را بر روی یک مجموعه برنامه‌های صحیح یاد می‌گیرند و سپس مدل یادگرفته شده را برای پیش‌بینی تصحیح نحو در برنامه دارای خطا به‌کار می‌برند. برخی کارهای مرتبط نیز برنامه‌های اسمبلی را بهینه‌سازی می‌کنند [8].

۳-۵-۲- یادگیری و فاز

مقاله یادگیری و فاز[۲] [8] روش جدیدی را برای تولید داده آزمون جهت استفاده در آزمون فازی بر مبنای RNN و مدل کدگذار-کدگشا ارایه کرده است. در این مقاله ساختار فایل PDF برای آزمون انتخاب شده است. ایده اصلی یادگیری یک مدل مولد زبان روی مجموعه‌ای از ویژگی‌های اشیای PDF با داشتن مجموعه‌ای از نمونه‌های اولیه است. مدل کدگذار-کدگشا اجازه یادگیری متن با طول دلخواه را برای پیش‌بینی توالی بعدی کاراکترها، می‌دهد. این مدل در مقایسه با رویکردهای سنتی n-gram که محدود به طول رشته متناهی هستند بهتر است.

با داشتن یک پیکره[۳] بزرگ از اشیای PDF مدل کدگذار-کدگشا می‌تواند در یک حالت بدون راهبر (بدون نظارت) آموزش داده شود تا مدل مولدی را برای تولید اشیای داده جدید PDF، با استفاده از توالی‌های ورودی و خروجی یاد بگیرد. توالی‌های ورودی وابسته به توالی‌هایی از کاراکترها در اشیای PDF است و توالی‌های مرتبط با خروجی از راه شیفت[۴] توالی‌های ورودی به‌میزان یک مکان حاصل می‌شود. مدل یادگیری شده سپس می‌تواند برای تولید توالی‌های جدید (اشیای PDF) به‌وسیله نمونه‌برداری از توزیع داده شده با یک پیشوند شروع خاص (نظیر "obj")، استفاده شود. ساختار اشیای داده‌ای اسناد PDF در **پیوست الف** سمینار شرح داده شده است.

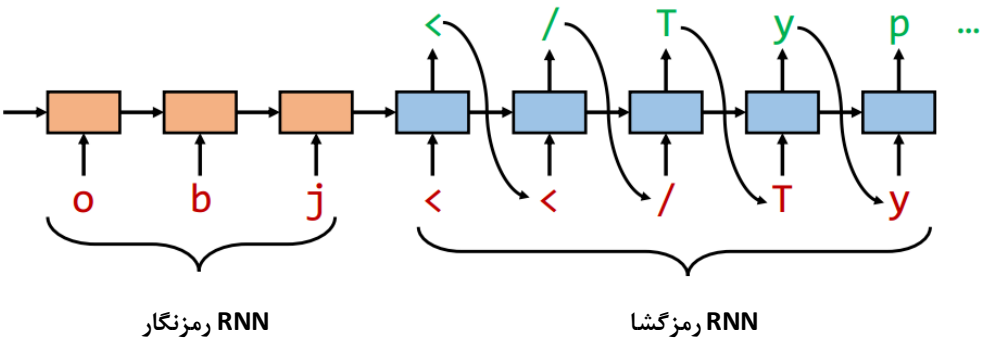
---

۱ domain specific language

۲ learn and fuzz

۳ corpus

۴ shift



شکل (۳-۲) یک مدل توالی به توالی برای یادگیری اشیای PDF [8].

آموزش شبکه. مدل کدگذار-کدگشا با استفاده از تکه اشیای PDF که هر یک به‌صورت توالی‌ای از کاراکترها در نظر گرفته شده است، آموزش می‌بینند. برای آموزش، ابتدا همه فایل‌های حاوی اشیای PDF به عنوان یک توالی از کاراکترها به یکدیگر الحاق[۱] می‌شوند که منجربه ایجاد یک توالی بزرگ از کاراکترها به‌صورت $\tilde{S} = s_1 + s_2 + \cdots + s_n$ خواهدشد. سپس توالی به چند زیر توالی آموزشی با طول ثابت d شکسته می‌شود به نحوی که iامین نمونه آموزشی عبارت است از: $t_i = \tilde{S}[i \times d : (i+1) \times d]$ که $S[l : u]$ یک زیرتوالی از S بین اندیس‌های l و u را نشان می‌دهد. توالی خروجی برای هر توالی آموزشی عبارت است از توالی ورودی که یک واحد به سمت راست شیفت داده شده است؛ یعنی، $o_t = \tilde{S}[i \times d + 1 : (i+1) \times d + 1]$. مدل سپس به‌صورت *انتها به انتها*[۲] آموزش داده می‌شود تا مدل مولد را روی مجموعه همه توالی‌های آموزشی موجود، یاد بگیرد. یک بازنمایی از نحوه آموزش شبکه در شکل (۳-۲) نشان داده شده است.

از آنجایی که مدل بحث شده در حالت بدون نظارت آموزش می‌بینند، برچسب‌های تولید شده به‌طور صریح برای مشخص کردن این که مدل یادگرفته شده چه‌قدر خوب عمل می‌کند، آزموده نشدند. به‌جای آن چندین مدل آموزش داده می‌شود که در تعداد دوره‌ها (بخش ۲-۶-۹-) متفاوت هستند. مدل در پنج دوره با تعدادهای مختلف ارزیابی شده است: ۱۰، ۲۰، ۳۰، ۴۰ و ۵۰. در تنظیمات شبکه یادگیری روی هر دوره حدود ۱۲ دقیقه طول می‌کشد و بنابراین مدل با ۵۰ عدد

[۱] concatenate

[۲] end-to-end، یعنی تنها یک مدل که ورودی را پذیرفته و خروجی را تولید می‌کند.

حدودا ۱۰ ساعت زمان نیاز دارد تا کامل شود. معماری شبکه در نظر گرفته شده برای مدل، LSTM با ۲ لایه مخفی که هر لایه ۱۲۸ نورون (حالت مخفی) دارد، است.

تولید اشیای جدید. از مدل یادگیری شده برای تولید اشیای جدید PDF استفاده خواهد شد. راهبردهای مختلفی برای تولید اشیا، بسته به راهبرد نمونه‌برداری که برای نمونه‌برداری از توزیع یادگیری شده به‌کار می‌رود، وجود دارند. با یک پیشوند از توالی "obj" (که شروع یک شی را مشخص می‌کند) شروع و سپس مدل را پرس‌وجو کرده تا یک توالی از کاراکترهای خروجی تولید کنیم تا زمانی که مدل "endobj" را که وابسته به پایان یک نمونه از شی است، تولید نماید. سه راهبرد مختلف نمونه‌برداری از توزیع احتمالی برای تولید اشیای جدید به‌کار بسته شده است:

۱. ***NoSample.*** در این راهبرد تولید، توزیع یادگرفته شده به‌کار می‌رود تا به‌صورت حریصانه بهترین کاراکتر را برای یک پیشوند خاص پیش‌بینی کند. این راهبرد برای تولید اشیای PDFای که خوش-شکل[۱] و سازگار[۲] هستند، بسیار نتیجه‌بخش است ولی تعداد اشیائی را که می‌توان تولید کرد، محدود می‌کند. با دادن یک پیشوند مثل "obj" بهترین توالی از کاراکترهای بعدی به‌صورت یکتا مشخص می‌شود و بنابراین این راهبرد همان شی را نتیجه می‌دهد. این محدودیت مغایر با اهداف آزمون فازی بوده و مانع مفید بودن استفاده از این راهبرد در آزمون فازی می‌شود.

۲. ***Sample.*** در این راهبرد تولید، به‌جای انتخاب بالاترین کاراکتر پیش‌بینی شده، از توزیع یادگرفته شده، برای نمونه‌برداری کاراکتر بعدی در توالی‌ای که پیشوند آن داده شده است، استفاده می‌شود. راهبرد نمونه‌برداری قادر به تولید مجموعه گوناگونی از اشیا با ترکیب الگوهای مختلفی که از اشیای موجود یادگرفته شده است، خواهد بود. به دلیل نمونه‌برداری اشیای تولید شده، تضمینی نیست که آنها خوش-شکل باشند، که این ویژگی از منظر آزمون فازی مفید است.

۳. ***SampleSpace.*** این راهبرد ترکیب دو راهبرد قبلی است. SampleSpace توزیع احتمالی برای تولید کاراکتر بعدی را تنها زمانی که توالی پیشوند فعلی با فضای خالی خاتمه می‌یابد،

---

[۱] well-formed

[۲] consistent

نمونه‌برداری می‌کند. برای میان توکن‌ها (یعنی پیشوندهایی که با فضای خالی خاتمه نمی‌یابند) بهترین کاراکتر را انتخاب می‌کند که شبیه به راهبرد اول است. این راهبرد به منظور تولید ورودی‌های خوش-شکل‌تر اشیای PDF در مقایسه با راهبرد دوم استفاده می‌شود. چون نمونه‌برداری تنها به پایان کاراکترهای فضای خالی محدود شده است؛ نمونه‌های جدید در این روش نیز ساخته خواهد شد.

اعمال آزمون فازی. الگوریتم SampleFuzz در شکل (۳-۳) نشان داده شده است. این الگوریتم مدل یادگرفته شده $D(x, \theta)$، احتمال فازینگ یک کاراکتر t_{fuzz} و احتمال آستانه p_t که تصمیم می‌گیرد آیا کاراکتر پیش‌بینی شده اصلاح شود، را به عنوان ورودی می‌پذیرد. در زمان تولید توالی خروجی seq، الگوریتم مدل یادگرفته شده را نمونه‌برداری می‌کند تا کاراکتر بعدی یعنی c و احتمال آن $p(c)$ را در یک مرحله زمانی مشخص t به‌دست آورد. اگر احتمال $p(c)$ از آستانه فراهم شده توسط کاربر (p_t) بالاتر باشد؛ یعنی، اگر مدل مطمئن باشد که کاراکتر بعدی به احتمال زیاد c است، الگوریتم تصمیم می‌گیرد تا یک کاراکتر دیگر c' را که حداقل احتمال P(c') در مدل یادگرفته شده را دارد جایگزین c کند. البته این اصلاح (فازینگ) تنها در صورتی که عدد تصادفی تولید شده p_{fuzz} از ورودی t_{fuzz} مقدار بیشتری داشته باشد، انجام می‌شود، که بدین ترتیب اجازه می‌دهد تا کاربر بتواند بر روی احتمال (درصد) کاراکترهای فاز شده کنترل داشته باشد.

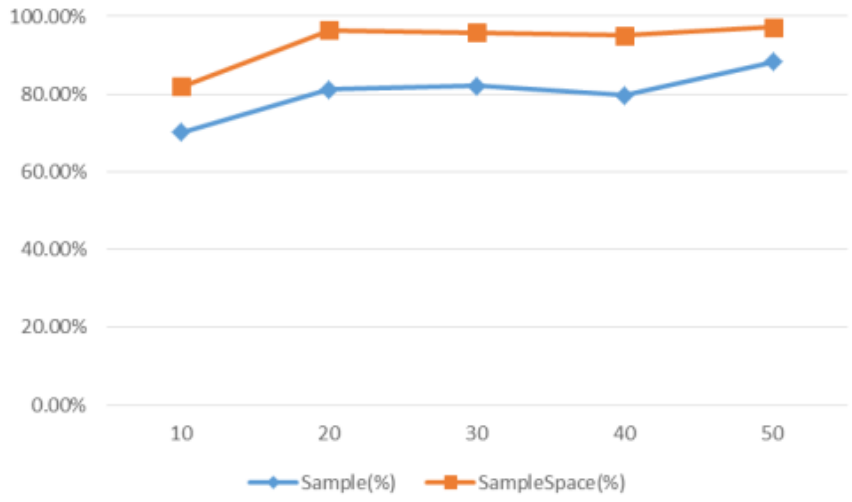
Algorithm SampleFuzz$(\mathcal{D}(\mathrm{x}, \theta), t_{\mathrm{fuzz}}, p_t)$

```
seq := "obj "
while ¬ seq.endswith("endobj") do
  c,p(c) := sample(D(seq,θ)) (* Sample c from the learnt distribution *)
  p_fuzz := random(0, 1) (* random variable to decide whether to fuzz *)
  if p_fuzz > t_fuzz ∧ p(c) > p_t then
    c := argmin_c' {p(c') ~ D(seq, θ)} (* replace c by c' (with lowest likelihood) *)
  end if
  seq := seq + c
  if len(seq) > MAXLEN then
    seq := "obj " (* Reset the sequence *)
  end if
end while
return seq
```

شکل (۳-۳) الگوریتم SampleFuzz برای نمونه‌برداری و سپس فازینگ داده آزمون ورودی [8].

یک نکته کلیدی الگوریتم SampleFuzz امکان قرار دادن کاراکترهای ناخواسته در اشیا تنها در مکان‌هایی که مدل بسیار مطمئن است، به‌منظور گمراه‌سازی پویش‌گر است. الگوریتم همچنین اطمینان می‌یابد که طول شی با عدد MAXLEN محدود شود. توجه شود که این شرط حلقه while الگوریتم تضمین نمی‌کند که همواره خاتمه یابد اما در آزمایش‌های انجام شده در عمل، همواره پایان یافته‌است.

نتایج. این مقاله PDFخوانِ جاسازی شده در مرورگر Edge شرکت مایکروسافت را مورد آزمون فازی قرار داده‌است. چیدمان آزمایش‌ها و نیز تحلیل جامع نتایج در [8] آمده است. برای هر اجرا، به‌صورت خودکار بررسی می‌شود که آیا خطای نحوی در سابقه اجرا ثبت شده است یا خیر. اگر پیامی نباشد آزمون گذر[۱] موفق داشته است. نرخِ‌گذر موردهای آزمون تولیدی برحسب تعداد دوره‌ها در نمودار شکل (۳-۴) نشان داده شده است. نرخ‌گذر بالا در این آزمایش‌ها حاکی از آن است که فرایند یادگیری به‌خوبی توانسته‌است ساختار پیچیده اشیای PDF را یاد بگیرد و ورودی‌های معتبری فراهم سازد. از سوی دیگر میزان پوشش کد SUT توسط این ورودی‌ها، بررسی و اندازه‌گیری شده که مقدار قابل قبولی را ارایه داده‌است.

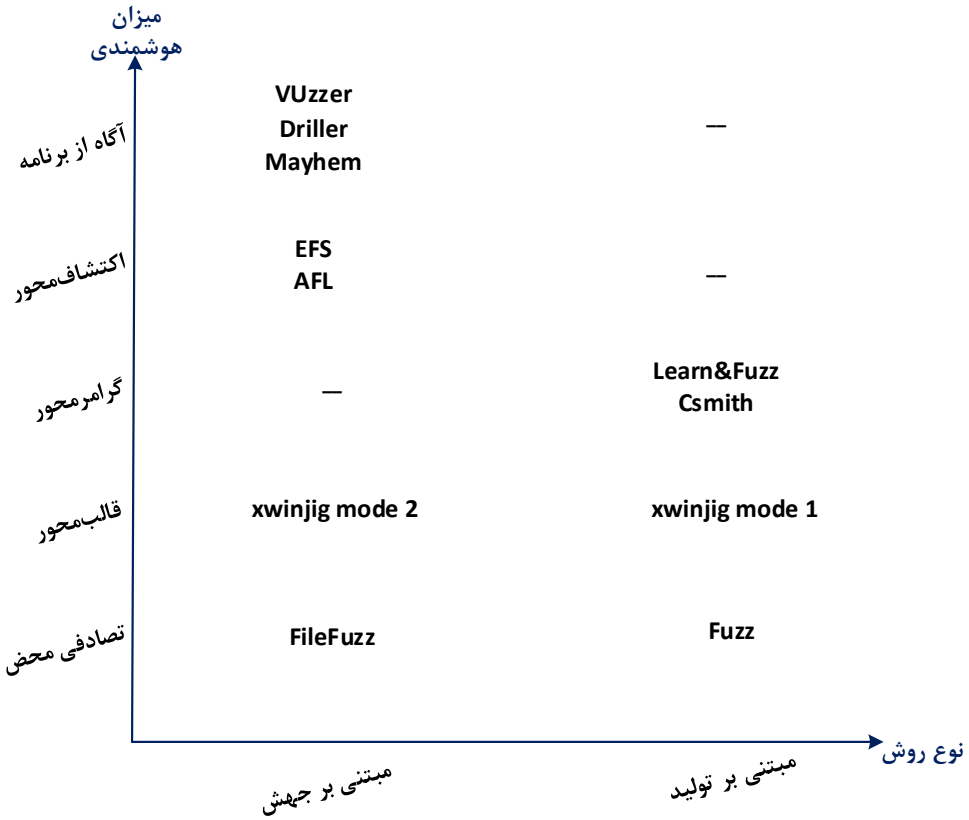


شکل (۳-۴) نرخ‌گذر اندازه‌گیری شده برای راهبردهای نمونه‌برداری Sample و SampleSpace در دوره‌های ۱۰ تا ۵۰.

[۱] pass

جدول (۳-۲) ابزارها و چارچوب‌های برنامه‌نویسی شناخته شده در زمینه یادگیری ژرف.

نام چارچوب	تارنمای توسعه‌دهنده چارچوب
TensorFlow	https://www.tensorflow.org/
Theano	http://deeplearning.net/software/theano/
Cognitive Toolkit (CNTK)	https://www.microsoft.com/en-us/cognitive-toolkit/
Keras	https://keras.io/



شکل (۳-۵) طبقه‌بندی فازرها بر اساس روش‌های خودکار تولید داده به‌کار رفته در آنها. محور افقی نوع روش و محور عمودی سازوکار تولید یا میزان هوشمندی روش را نشان می‌دهد. روش‌های ترکیبی در محور افقی درنظرگرفته نشده‌اند. همچنین طبقه‌بندی بر روی تنها بر روی عناوین فازرهای مطرح در حوزه این سمینار انجام شده است.

۳-۵-۳- ابزارهای یادگیری ژرف

با همه‌گیر شدن و گسترش استفاده از مفاهیم یادگیری ژرف و شبکه‌های عصبی ژرف، ابزارها و چارچوب‌های زیادی برای این منظور توسعه داده شده‌اند. تعدادی از این چارچوب‌ها در جدول (۳-۲) ذکر شده‌اند. نکته قابل ذکر پشتیبانی تمامی آنها از CUDA[۱] است. CUDA یک سکو[۲]ی پردازش موازی برروی GPU است که توسط شرکت Nvidia ابداع شده است [45]. استفاده از CUDA در وظایف یادگیری ژرف که نیازمند اعمال محاسباتی بسیاری هستند، آموزش و استفاده سریع از شبکه‌های عصبی ژرف را میسر ساخته‌است. چارچوب Keras که در جدول (۳-۲) معرفی شده است، یک چارچوب سطح بالا است که بر روی سه‌چارچوب دیگر ذکر شده در این جدول نصب گردیده و برای طراحی و پیاده‌سازی RNNها بسیار مناسب است.

۳-۶- نتیجه‌گیری

در این فصل تعدادی از مهمترین روش‌های به‌کار رفته برای تولید آزمون در فازرهای مبتنی بر قالب فایل مطرح و بحث شد. می‌توان روش‌های مذکور در این فازرها را باتوجه به دو محور اساسی طبقه‌بندی کرد. محور اول شامل نوع کلی روش تولید داده و محور دوم میزان هوشمندی روش در نظر گرفته شده است. در شکل (۳-۵) این طبقه‌بندی نشان داده شده است. در هر قسمت سعی شده یک فازر کاربردی قرار گیرد. بدیهی است که انجام چنین طبقه‌بندی برای برخی فازرها ممکن است مشکل و نادقیق باشد؛ زیرا، امروز بیشتر روش‌های مرکب و آمیخته از تعدادی روش ساده‌تر هستند. با این حال با تقریب خوبی برای کلیه فازرهای موجود قابل اعمال است. ایجاد یک طبقه‌بندی می‌تواند در اشراف به موضوع و درک جایگاه فازرهای مختلف کمک شایانی نماید. یک طبقه‌بندی جامع‌تر از فازرها در [13] بیان شده است.

روش مبتنی بر مدل RNN کدگذار-کدگشا را می‌توان نوعی روش گرامر محور دانست که خودکارسازی فرایند درک گرامر (ساختار) روی آن اعمال شده است. بنابراین یک روش کاملا

[۱] compute unified device architecture
[۲] platform

خودکار برای تولید داده و آزمون فازی است. در حال حاضر این روش تنها بر روی اشیای داده‌ای قالب فایل PDF و در حالت بسیار ابتدایی پیاده‌سازی شده است. این امکان وجود دارد که آن را بر روی دیگر قالب‌های فایل نیز اعمال کرد. در فصل بعد برخی چالش‌های موجود در این روش و نیز برخی از موضوعات کاری آتی در این زمینه مطرح و بحث خواهد شد.

فصل ۴: نتیجه‌گیری و کارهای آتی

مسئله مهم این است که دست از سؤال پرسیدن نکشید. کنجکاوی دلیلی است برای وجود داشتن.

آلبرت انیشتین

۴-۱- نتیجه‌گیری

آزمون فازی به عنوان یکی از فنون آزمون نرم‌افزار در کشف خطاهای امنیتی و آسیب‌پذیری‌ها بسیار مؤثر بوده است. در این پژوهش سعی شد ضمن معرفی مفاهیم مقدماتی مربوط به این فن آزمون، به‌طور خاص روش‌های تولید خودکار داده آزمون، بررسی گردند. بیشتر نرم‌افزارهای کاربردی یک فایل را به‌عنوان ورودی پذیرفته و پردازش می‌کنند. تنوع در تولید این فایل‌ها می‌تواند جهت کشف خطا و آسیب‌پذیری‌ها امید بخش باشد. به‌همین دلیل فازرهای مبتنی بر قالب فایل برای این منظور مورد مطالعه قرار گرفتند.

اهمیت آزمون فازی و مسئله پوشش کد در کشف خطا در فصل اول بیان شد. در فصل دوم معماری فازرها، ساختار حافظه و آسیب‌پذیری‌ها بحث شد. دیدیم تولید خودکار داده در آزمون فازی نقش بسیار مهمی دارد. کشف یک خطا و به‌دنبال آن آسیب‌پذیری منوط به داشتن یک مورد آزمون آشکار کننده آن خطا خواهد بود. اگر ورودی تولیدشده ساختار مناسبی نداشته باشد آنگاه توسط پویش‌گر نحوی در مرحله اول پردازش رد می‌شود. پس تنها کد پویش‌گر اجرا می‌شود و امکانی برای کشف خطاهای کدهای لایه عمیق‌تر نیست. روش‌های گرامر محور در رسیدن به کدهای عمیق موفق بوده‌اند اما خودکارسازی آنها زمانبر و پُر هزینه و نیازمند دسترسی به مشخصه‌های قالب فایل است.

یادگیری ژرف که در سال‌های اخیر پیشرفت چشمگیری پیدا کرده است می‌تواند برای یادگیری ساختار قالب فایل‌های پیچیده نیز بهره‌برداری شود. از شبکه‌های عصبی ژرف، برای یادگیری وظایف این حوزه استفاده می‌شود. پژوهش انجام شده نشان می‌دهد مدل RNN کدگذار-کدگشا در یادگیری وظایف مبتنی بر توالی بسیار اثر بخش بوده است. ایده استفاده از مدل مذکور در آزمون فازی در [8] مطرح و از آن برای آزمون PDFخوان مرورگر Edge استفاده شده است. نتایج به‌دست آمده رسیدن به پوشش کد و نرخ گذر بالا را نشان می‌دهد. این در حالی است که PDF قالب فایل بسیار پیچیده‌ای به‌شمار می‌آید. ویژگی بارز این روش خودکار بودن فرایند یادگیری ساختار و نیز فرایند تولید داده آزمون است.

مفاهیم پایه شبکه‌های عصبی ژرف بسیار گسترده و مبتنی بر علوم آمار و احتمالات، بهینه‌سازی و محاسبات عددی بوده که در نگاه اول ارتباطی با مفاهیم حوزه سیستم، امنیت و آزمون نرم‌افزار ندارند. کلیات و مبانی نظری این شبکه‌ها در فصل دوم و جزئیات روش یادگیری و فاز در فصل سوم بیان گردید.

روش‌های دیگری برای تولید داده آزمون وجود دارد از جمله روش‌های اکتشافی و آگاه از برنامه که بررسی شدند. در این روش‌ها بیشتر از الگوریتم‌های تکاملی مثل ژنتیک برای تولید داده آزمون استفاده شده است که برای فایل‌هایی دودویی محض مناسب به نظر می‌رسد. تعدادی از این روش‌ها نیز در فصل سوم با ذکر نام معرفی شدند.

از زمان ابداع فن فازی در آزمون نرم‌افزار، تعداد زیادی خطا و آسیب‌پذیری در برنامه‌های مختلف با این فن تشخیص داده شده است. از جمله در سیستم عامل‌های یونیکس و ویندوز و نیز در مرورگرهای وب. الگوریتم‌های هوش مصنوعی در بهبود فازرها نقش قابل توجهی را بازی کرده‌اند و مطالعات جدید به سمت آنها معطوف شده است. با این اوصاف کماکان چالش‌ها و مسائل متعددی پیشروی پژوهشگران این حوزه وجود دارد. در ادامه فصل برخی از این مسائل حل نشده و چالش‌های موجود بیان می‌گردد.

۴-۲- مسائل باز و کارهای قابل انجام

در این سمینار به‌دنبال روش‌های تولید داده آزمونی بودیم که پوشش کد را افزایش داده و در مراحل ابتدایی نیز رد نشود؛ یعنی تلف‌شده تلقی نگردد. بخش عمده این سمینار مطالعه نحوه بهره‌گیری از مدل‌های مولد شبکه‌های عصبی برای تولید داده آزمون مورد نیاز فازر مبتنی بر قالب فایل بود. چندین مسئله باز و زمینه پژوهشی جالب برای کارهای آتی مرتبط با مباحث خودکارسازی یادگیری گرامر، تولید داده آزمون و بلاخره انجام فازینگ وجود دارد که به صورت فهرست‌بار در زیر به آنها اشاره می‌کنیم.

- در حالی که تمرکز در [8] بر روی یادگیری اشیای PDF است، که تنها قسمتی از مشخصه‌های فایل PDF را تشکیل می‌دهند؛ نحوه یادگیری سایر قسمت‌های سطح بالاتر در سلسله مراتب

فایل PDF نظیر جدول ارجاع متقابل، بدنه اشیا و بخش توالی در بدنه که شامل پیچیدگی‌ها و گوناگونی بیشتری هستند، می‌تواند مد نظر قرار گیرد. شاید ترکیب برخی جنبه‌های فنون استنتاج منطقی و شبکه‌های عصبی به‌قدر کافی برای این منظور قدرتمند باشند.

- الگوریتم SampleFuzz [8] ناآگاه از SUT عمل می‌کند؛ یعنی، در زمره روش‌های آگاه از برنامه بحث شده [9] نیست. برای نیل به یادگیری ساختار و پوشش کد بهتر می‌توان از بازخورد برنامه نسبت‌به ورودی‌های تولید شده استفاده کرد و این بازخورد را با خروجی مدل مولد یادگیری شده ترکیب نمود، بدین ترتیب الگوریتم آگاهانه‌ای در راستای پوشش بهتر کد خواهیم داشت.
- ایجاد یک الگوریتم جدید برای اعمال فازینگ مشابه SampleFuzz می‌تواند در پوشش کد و کشف آسیب‌پذیری‌ها موفق‌تر باشد. برای مثال در هنگام تولید توالی عددی، به جای استفاده از مدل مولد از یک مقدار مرزی استفاده کرد. افزون بر این مجموعه آموزش استفاده شده در فرایند آموزش شبکه را می‌توان در شیوه‌های مختلفی تنوع بخشید مثلا در آن از داده‌هایی استفاده شود که قبلا منجربه ایجاد خرابی در SUTهای دیگر شده‌اند. این امر احتمالا امکان خرابی و کشف خطا در SUT جدید را افزایش می‌دهد.
- یادگیری ساختار با استفاده از شبکه‌های عصبی روی برخی بخش‌های غیر دودویی قالب فایلPDF اعمال شده است. این الگوریتم را می‌توان به سایر قالب‌های فایل که محتوی بخش‌های غیر دودویی هستند یا قالب‌های متنی مثل زبان‌های نشانه‌گذاری[1] نیز اعمال نمود؛ برای مثال یادگیری فایل‌های HTML و XML برای فازینگ مرورگرهای وب و نیز یادگیری فایل‌های حاوی کد منبع زبان‌های برنامه‌نویسی مثل C، JavaScript و غیره (به‌خصوص زبان‌های نو) برای فازینگ کامپایلرها و مفسرهای آنها می‌تواند ارزشمند باشد. در این مورد یکی از چالش موجود بالا بردن دقت یادگیری شبکه است و مواجه شدن با مسئله پیچیدگی مدل است به‌طوری‌که مثلا بتوان در وهله نخست فایل‌هایی با نحو و معنای معتبر ایجاد نمود.
- مسئله باز و چالش انگیز دیگر اعمال روش یادگیری به فایل‌های دودویی و بررسی نحوه یادگیری آماری ساختار اینگونه فایل‌ها است. به‌عنوان نمونه فایل‌های چندرسانه‌ای برای این منظور بسیار جالب به‌نظر می‌آیند. باید در نظر داشت که ایجاد مدل یادگیری کاملا مستقل

[1] mark-up

از قالب فایل است و به همین دلیل یک مدل مناسب خواهد توانست قالب‌های دودویی مختلفی را یاد بگیرد. چالش اصلی احتمالا چگونگی آماده‌سازی و بازنمایی داده‌های ورودی شبکه است؛ زیرا در اینجا با داده‌های متنی مبتنی برکاراکتر روبه‌رو نیستم.

- روش یادگیری و فاز همچنین می‌تواند با فازرهای مبتنی بر جهش نیز ترکیب شود. یعنی ابتدا داده جدیدی براساس مدل مولد تولید و سپس مکان مناسبی از آن جهش داده شود. هدف اینگونه جهش تنها نباید افزایش سطح و عمق پوشش کد باشد (که در مورد دوم این فهرست بدان اشاره شد)؛ بلکه می‌تواند برای تزریق مقادیر مرزی و ناخواسته در جاهای تشخیص داده شده استفاده شود. چنانچه قبلا هم دیدیم، مقادیر مرزی و خارج از محدوده در صورت کنترل ناکافی به‌راحتی موجبات سرریزهای مختلف حافظه و آسیب‌پذیری را فراهم می‌آورند.
- روش یادگیری و فاز را می‌توان به غیر از قالب‌های فایل به آزمون فازی دیگر حوزه‌ها نیز اعمال نمود. به‌عنوان نمونه یادگیری ساختار پروتکل‌های شبکه برای تولید داده در فازرهای مبتنی بر شبکه؛ به‌ویژه در آزمون پروتکل‌ها با ساختار ناشناخته مثل بات‌نت‌ها قابل توجه است. در این مورد یادگیری ممکن است به اهداف مهندسی معکوس کمک نماید.

۴-۳- موضوع مورد نظر برای پایان‌نامه

شناسایی آسیب‌پذیری‌های نرم‌افزارها همواره حائز اهمیت بوده است. آزمون فازی و روش‌های خودکار تولید داده آزمون برای آن، در این حیطه بسیار موفق عمل کرده‌اند به همین دلیل تلاش در راستای بهبود این روش‌ها از چشم‌انداز پژوهشی مطلوبی برخوردار است. مسائل باز و کارهای قابل انجام در بخش قبلی فهرست شدند. به‌عنوان موضوع مورد نظر برای پایان‌نامه مورد سوم فهرست مذکور، یعنی بهبـود الگوریتم فاز، مناسب و نوآورانه به‌نظر می‌آید. برمبنای شرح مسـئله بیان‌شده در بخش ۱-۲- به‌دنبال اهداف زیر هستیم:

- ایجاد یک مدل برای یادگیری و تولید ساختار فایل‌های پیچیده مثل PDF،
- ایجاد یک روش فازینگ یا همان بد-شکل‌سازی فایل‌های تولید شده،
- نرخ گذر (گذر از پویشگر اولیه) بالاتر و به تبع آن رسیدن به مسیرهای اجرایی عمیق برنامه،
- و درنهایت کشف آسیب‌پذیری‌های احتمالی موجود در SUT.

مراجع

[1] M. Sutton, A. Greene, and P. Amini, *Fuzzing brute force vulnerability discovery*, 1st ed. Addison-Wesley, 2007.

[2] A. Kettunen, “Test harness for web browser fuzz testing,” University of Oulu, 2014.

[3] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[4] B. P. Miller *et al.*, “Fuzz revisited - A re-examination of the reliability of unix utilities and services,” *October*, vol. 1525, no. October 1995, pp. 1–23, 1995.

[5] J. E. Forrester and B. P. Miller, “An empirical study of the robustness of Windows NT applications using random testing,” *Proc. 4th USENIX Wind. Syst. Symp.*, no. August, pp. 59–68, 2000.

[6] B. P. Miller, G. Cooksey, and F. Moore, “An empirical study of the robustness of MacOS applications using random testing,” *Proc. 1st Int. Work. Random Testing, RT’06*, vol. 2006, no. March 2017, pp. 46–54, 2006.

[7] S. Veggalam, S. Rawat, I. Haller, and H. Bos, “IFuzzer : An evolutionary interpreter fuzzer using genetic programming,” pp. 1–20.

[8] P. Godefroid, H. Peleg, and R. Singh, “Learn&Fuzz: Machine learning for input fuzzing,” *Microsoft Res.*, 2017.

[9] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “VUzzer: Application-aware evolutionary fuzzing,” no. March, 2017.

[10] U. Kargén and N. Shahmehri, “Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing,” *Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015*, pp. 782–792, 2015.

[11] P. Ammann and J. Offutt, *Introduction to software testing*, no. 1. 2008.

[12] H. C. Kim, Y. H. Choi, and D. H. Lee, “Efficient file fuzz testing using automated analysis of binary file format,” *J. Syst. Archit.*, vol. 57, no. 3, pp. 259–268, 2011.

[13] R. Mcnally, K. Yiu, and D. Grove, “Fuzzing: the state of the art,” *DSTO Def. Sci. Technol. Organ.*, p. 55, 2012.

[14] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for software security testing and quality assurance*, 1st ed. Artech House, 2008.

[15] C. Miller and Z. Peterson, “Analysis of mutation and generation-based fuzzing,” *White Pap. Indep. Secur. …*, pp. 1–7, 2007.

[16] G. Evron and N. Rathaus, *Open source fuzzing Tools*. 2007.

[17] D. Lesko and M. Tejfel, “A domain based new code coverage metric and a related automated test data generation method,” vol. 36, pp. 217–240, 2012.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016.

[19] J. D. Demott, “Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing,” 2007.

[20] E. Dubrova, *Fault-tolerant design*. 2013.

[21] Paul C. Jorgensen, *Software testing a craftsman's approach*, Fourth Edi., vol. 47, no. (2). CRC Press Taylor & Francis Group, 2014.

[22] M. E. Khan and F. Khan, "A comparative study of white box , black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl.*, vol. 3, no. 6, pp. 12–15, 2012.

[23] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," *Usenix*, p. 38, 2012.

[24] W. Underwood, "Grammar-based specification and parsing of binary file formats," *Int. J. Digit. Curation*, vol. 7, no. 1, pp. 95–106, 2012.

[25] "Internet security glossary." [Online]. Available: https://tools.ietf.org/html/rfc2828. [Accessed: 11-Oct-2017].

[26] D. Shiffman, S. Fry, and Z. Marsh, *The nature of code*. D. Shiffman, 2012.

[27] A. Karpathy, "Connecting images and natural language," Stanford University, 2016.

[28] B. Sautermeister, "Deep learning approaches to predict future frames in videos," 2016.

[29] A. Krizhevsky, I. Sutskever, and H. Geoffrey E., "ImageNet classification with deep convolutional neural networks," *Adv. Neural Inf. Process. Syst. 25*, pp. 1–9, 2012.

[30] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Proc. 13th Int. Conf. Artif. Intell. Stat.*, vol. 9, pp. 249–256, 2010.

[31] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: surpassing human-level performance on imagenet classification," *Proc. IEEE Int. Conf. Comput. Vis.*, vol. 2015 Inter, pp. 1026–1034, 2015.

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, 2014.

[33] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Trans. Neural Networks Learn. Syst.*, 2016.

[34] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," pp. 1–38, 2015.

[35] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014.

[36] Q. V. Le Ilya Sutskever, Oriol Vinyals, I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Nips*, pp. 1–9, 2014.

[37] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *ACM SIGPLAN Not.*, vol. 43, no. 6, p. 206, 2008.

[38] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs

in C compilers," *ACM SIGPLAN Not.*, vol. 47, no. 6, p. 283, 2012.

[39] "American fuzzy lop." [Online]. Available: http://lcamtuf.coredump.cx/afl/. [Accessed: 11-Oct-2017].

[40] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," *Proc. - IEEE Symp. Secur. Priv.*, pp. 380–394, 2012.

[41] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic sxecution," *Ndss*, no. February, pp. 21–24, 2016.

[42] O. Bastani, A. Aiken, and P. Liang, "Synthesizing program input grammars," pp. 1–17, 2016.

[43] P. Torres-Tram??n, H. Hromic, B. Walsh, B. R. Heravi, and C. Hayes, "Mining input grammars from dynamic taints," *CEUR Workshop Proc.*, vol. 1691, pp. 64–66, 2016.

[44] W. Cui *et al.*, "Tupni: automatic reverse engineering of input formats," *Conf. Comput. Commun. Secur.*, p. 11, 2008.

[45] "CUDA | GeForce." [Online]. Available: https://www.geforce.com/hardware/technology/cuda. [Accessed: 12-Oct-2017].

پیوست الف: ساختار اسناد PDF

معرفی. در فصل سوم روشی تحت عنوان یادگیری و فاز بحث شد که در آن ساختار اشیای داده‌ای اسناد PDF مورد یادگیری و آزمون فازی قرار گرفته‌اند. در این پیوست به‌طور خلاصه ساختار کلی یک سند PDF را به‌عنوان یک ساختار فایل پیچیده بیان می‌کنیم. ویژگی‌های کامل قالب PDF بیش از **۱۳۰۰** صفحه است. بیشتر این ویژگی‌ها – در حدود ۷۰ درصد- در ارتباط با توضیح اشیای داده و ارتباط آنها بین بخش‌های مختلف یک سند PDF هست. تمرکز اصلی در این قسمت نیز بر روی ساختار اشیای داده‌ای خواهد بود.

فایل‌های PDF در یک قالب متنی کدگذاری می‌شوند که ممکن است شامل جریان‌های دودویی مانند عکس و غیره باشند. یک فایل PDF ترتیبی از دستِ‌کم یک بدنه PDF است. یک بدنه PDF از سه بخش تشکیل شده است: *اشیا*[۱]، *جدول ارجاع-متقابل*[۲] و *trailer*.

اشیا. داده و فراداده در پرونده‌های PDF در یک واحد اولیه که اشیا نامیده می‌شود سازماندهی می‌شوند. اشیا همگی قالب مشابهی دارند که در شکل (الف-۱)الف نشان داده شده است و همچنین یک ساختار بیرونی مشترک هم دارند. اولین خط یک شی شناسه آن است، برای ارجاع‌های غیر مستقیم. در ادامه عدد تولیدی آن است که اگر شی با یک نسخه جدید تر رونویسی شود، افزایش می‌یابد. در پایان عبارت "obj" که شروع یک شی را مشخص می‌کند. "endobj" پایان شی را مشخص می‌کند.

شی نشان داده شده در شکل (الف-۱)الف، حاوی یک ساختار فرهنگ‌لغت[۳] است که بین علامت‌های << و >> واقع‌شده و شامل کلیدهایی می‌شود که با / شروع شده و در ادامه مقادیر آنها آمده است. [3 0 R] یک ارجاع به یک شی در همین سند با شناسه ۳ و عدد تولیدی ۰ است. از آنجایی که یک سند ممکن است خیلی بزرگ باشد. شیئی که به آن ارجاع داده شده است از طریق یک جدول دسترسی تصادفی قابل دستیابی است.

---

[۱] objects

[۲] cross-reference table

[۳] dictionary

```
2 0 obj
<<
/Type /Pages
/Kids [ 3 0 R ]
/Count 1
>>
endobj
```

الف

```
xref
0 6
0000000000 65535 f
0000000010 00000 n
0000000059 00000 n
0000000118 00000 n
0000000296 00000 n
0000000377 00000 n
0000000395 00000 n
```

ب

```
trailer
<<
/Size 18
/Info 17 0 R
/Root 1 0 R
>>
startxref
3661
```

پ

شکل (الف-۱) بریده‌هایی از یک سند PDF درست. **(الف)**: یک شی نمونه ساده. (ب): جدول ارجاع متقابل. **(پ)**: یک trailer.

```
125 0 obj
[680.6 680.6]
endobj
```

الف

```
88 0 obj
(Related Work)
endobj
```

ب

```
75 0 obj
4171
endobj
```

پ

```
47 1 obj
[false 170 85.5 (Hello) /My#20Name]
endobj
```

ت

شکل (الف-۲) انواع مختلف اشیای داده‌ای موجود در سند PDF.

نمونه‌های دیگری از اشیای قابل روئیت در اسناد PDF در شکل (الف-۲) آمده‌است. شی نشان داده شده در شکل (الف-۲)الف، شامل محتوای [680.6, 680.6] است که یک شی ارایه است. هدف آن نگهداری مختصاتی است که توسط شی دیگری مورد مراجعه قرار می‌گیرد. شکل (الف-۲)ب، یک ثابت رشته‌ای است که یک bookmark را در یک سند PDF نگهداری می‌کند. شکل (الف-۲)پ، یک شی عددی است. شکل (الف-۲)ت. یک شی حاوی یک آرایه چند نوعی است. مثال‌های گوناگونی از اشیا وجود دارد که به هم به صورت مجزا و هم به صورت بلاک‌های اولیه

تشکیل دهنده سایر اجزا به‌کار می‌روند (برای مثال فرهنگ‌لغت موجود در شی شکل (الف-۱)الف شامل یک آرایه می‌شود). قواعد تعریف و ترکیب اشیا بیشترین بخش از توضیحات ویژگی‌های قالب PDF را تشکیل می‌دهند.

جدول ارجاع متقابل. این جدول در بدنه PDF شامل آدرس اشیای مورد ارجاع قرار گرفته داخل یک سند به صورت بایت می‌باشد. شکل (الف-۱)ب، یک مدل از این جدول را که حاوی یک زیربخش که شامل ۵ شی با شناسه یک تا ۵ و یک مکان نگهدارنده برای شناسه صفر که هرگز به یک شی ارجاع نمی‌دهد، است.

Trailer. این قسمت از بدنه فایل PDF شامل یک فرهنگ‌لغت از اطلاعات بدنه و startxref که آدرس شروع جدول ارجاعات است، می‌شود (مجدداً در بین علامت‌های << و >> قرار می‌گیرد). این امر اجازه می‌دهد تا بدنه از پایان با خواندن startxref پویش شود و سپس به جدول ارجاعات بازگشته و آن را پویش می‌کند و تنها اشایی پویش می‌شوند که نیاز هستند. شکل (الف-۱)پ یک trailer را نشان می‌دهد.

بروزرسانی یک سند. اسناد PDF می‌توانند به‌صورت مرحله‌ای (افزایشی) بروزرسانی شوند. این بدان معنی است که اگر نویسنده PDF بخواهد اطلاعات داخل شی ۱۲ را بروزرسانی کند، یک بدنه جدید را آغاز می‌کند. داخل آن شی جدید را می‌نویسد و عدد تولیدی را یک واحد نسبت به عدد تولیدی شی قدیم افزایش می‌دهد و برای شی جدید می‌نویسد. سپس یک جدول ارجاع جدید را که به شی جدید اشاره می‌کند می‌نویسد و این بدنه جدید را به سند قبلی الصاق می‌کند.

واژه‌نامه

واژه‌نامه فارسی به انگلیسی

واژه‌ی فارسی	معادل انگلیسی
ابزار گذاری	Instrumenting
اختلال	Noise
ارتباط‌گرایی	Connectionism
استنتاج	Inference
اشاره‌گر	Pointer
اشکال	Error
اعتبارسنجی	Validation
اکتشافی (آوینی)	Heuristic
الحاق	Concatenate
الگو	Pattern
انتها به انتها	End to End
آدرس نسبی	Offset
آزمون رفتاری	Behavioral Testing
آزمون ساختاری	Structural Testing
آزمون عملیاتی	Functional Testing
آزمون فازی	Fuzz Testing
آسه	Axon
آسیب‌پذیری	Vulnerability
آلودگی	Taint
آلودگی پویا	Dynamic Taint
آنتروپی متقاطع دودویی	Binary Cross-Entropy
بایاس	Bias
بایت‌های جادویی	Magic Bytes
برچسب	Label
برنامه شناسنده	Recognizer Program
برنامه مولد	Generator Program
بهره‌برداری	Exploit
بیش‌برازندگی	Overfitting
پایگاه دانش	Knowledge Base
پرداخت	Render
پرسپترون	Perceptron
پرسپترون چندلایه	Multi-Layer Perceptron
پس‌انتشار (انتشار پس‌رو)	Backpropagation

پس‌انتشار در زمان	Backpropagation Through Time (BPTT)
پوشش جمله	Statement Coverage
پوشش شاخه	Branch Coverage
پوشش فضای ورودی	Input Space Coverage
پوشش گراف	Graph Coverage
پوشش مسیر	Path Coverage
پوشش منطق	Logic Coverage
پویش	Parse
پویشگر	Parser
پیچیدگی مدل	Model Complexity
پیکره	Corpus
پیمانه	Module
تابع انگیزش	Activation Function
تابع بیشینه هموار	Softmax Function
تابع خطا	Error Function
تابع عینی (تابع هدف)	Objective Function
تابع گمشدگی	Loss Function
تابع هزینه	Cost Function
تابع یکسوساز	Rectified Linear Unit (ReLU)
تبدیل عکس به نوشته	Optical Character Recognition (OCR)
ترجمه ماشینی	Machine Translation (MT)
تصمیم‌ناپذیر	Undecidable
تطبیقی	Adaptive
تعمیم‌پذیری	Generalization
تقاطع	Crossover
تکاملی	Evolutionary
توالی	Sequence
توالی‌به‌توالی	Sequence to Sequence
جامعیت	Integrity
جعبه‌خاکستری	Gray-Box
جعبه‌سفید	White-Box
جعبه‌سیاه	Black-Box
جعبه‌شیشه‌ای	Glass-Box
حافظه کوتاه‌مدت بلند	Long-Short Term Memory (LSTM)
حسابرسی کد	Code Auditing
خرابی	Failure
خطا	Fault

خطای طراحی	Design Fault
خطای مطلق میانگین	Mean Absolute Error
خطای میانگین مربعات	Mean Squared Error
خوش-شکل	Well-Formed
خوشه‌بندی	Clustering
داده آزمون	Test Data
داده‌های خام	Raw Data
دارینه	Dendrite
دسترسی‌پذیری	Availability
دوره	Epoch
رگرسیون لَجیستیک	Logistic Regression
روبرداری	Dump
روزِ صفر	Zero-Day
زبان خاص دامنه	Domain Specific Language
سازگار	Consistent
سازوکار	Mechanism
سایبرنتیک	Cybernetics
سرریز	Overflow
سقوط	Crash
سکو	Platform
سمت-مشتری	Client-Side
سیگموید	Sigmoid
شبکه رو به جلو	Feedforward Network
شبکه وانیلی	Vanilla Network
شبکه‌های عصبی ژرف	Deep Neural Networks
شبکه‌های عصبی مکرر	Recurrent Neural Networks (RNN)
شرط تودرتو	Nested Condition
شیفت	Shift
طبقه‌بندی	Classification
فازر	Fuzzer
فازینگ	Fuzzing
فازینگ هوشمند	Smart Fuzzing
فرسایش ورزن	Weight Decay
قالب	Template
قالب فایل	File Format
کاهش گرادیان	Gradient Decent
کاهش گرادیان تصادفی	Stochastic Gradient Decent (SGD)

کدگذار-کدگشا	Encoder-Decoder
گذر	Pass
گراف جریان داده	Data Flow Graph
گراف جریان کنترلی	Control Flow Graph
گراف جهت دار بدون دور	Directed Acyclic Graph (DAG)
گراف دوبخشی کامل	Complete Bipartite Graph
ماتریس طراحی	Design Matrix
ماشین بردار پشتیبان	Support Vector Machine (SVM)
مبتنی بر تکه	Chunk-Based
مبتنی بر تولید	Generation-Based
مبتنی بر جهش (جابه‌جایی)	Mutation-Based
مبتنی بر دایرکتوری	Directory-Based
مجموعه آزمون	Test Set
مجموعه آموزش	Training Set
مجموعه تأیید	Validation Set
محرمانگی	Confidentiality
مدل مولد	Generative Model
مشخصه	Specification
معتبر	Valid
معیارهای پوشش	Coverage Criteria
مفاهیم	Concepts
مقیاس‌پذیری	Scalability
ممانعت از سرویس	Denial of Service (DoS)
ممانعت از سرویس توزیع‌شده	Distribute Denial of Service (DDos)
منظم‌سازی	Regularization
مورد آزمون	Test Case
میانگیر	Buffer
ناظر (راهبر)	Supervisor (Teacher)
نرم‌افزار تحت آزمون	Software Under Test (SUT)
نشان‌گر	Marker
نقص	Defect / Bug
نمادین	Symbolic
نورون (عصب)	Neuron
نیمه نظارتی	Semi-Supervised
واحد	Unit
واقعی	Concrete
واقعی-نمادین	Concolic

ورودی ناخواسته	Unexpected Input
وظیفه	Task
ویژگی	Feature
هدف	Target
همایه	Synapse
همگرایی	Converge
یادگیری بانظارت	Supervised Learning
یادگیری بدون نظارت	Unsupervised Learning
یادگیری تقویتی	Reinforcement Learning
یادگیری ژرف	Deep Learning
یادگیری ماشین	Machine Learning (ML)

واژه نامه انگلیسی به فارسی

واژه‌ی انگلیسی	معادل فارسی
Activation Function	تابع انگیزش
Adaptive	تطبیقی
Availability	دسترسی‌پذیری
Axon	آسه
Backpropagation	پس‌انتشار (انتشار پس‌رو)
Backpropagation Through Time (BPTT)	پس‌انتشار در زمان
Behavioral Testing	آزمون رفتاری
Bias	بایاس
Binary Cross-Entropy	آنتروپی متقاطع دودویی
Black-Box	جعبه‌سیاه
Branch Coverage	پوشش شاخه
Buffer	میانگیر
Chunk-Based	مبتنی بر تکه
Classification	طبقه‌بندی
Client-Side	سمت-مشتری
Clustering	خوشه‌بندی
Code Auditing	حسابرسی کد
Complete Bipartite Graph	گراف دوبخشی کامل
Concatenate	الحاق
Concepts	مفاهیم
Concolic	واقعی-نمادین
Concrete	واقعی
Confidentiality	محرمانگی
Connectionism	ارتباط‌گرایی
Consistent	سازگار
Control Flow Graph	گراف جریان کنترلی
Converge	همگرایی
Corpus	پیکره
Cost Function	تابع هزینه
Coverage Criteria	معیارهای پوشش
Crash	سقوط
Crossover	تقاطع
Cybernetics	سایبرنتیک

Data Flow Graph	گراف جریان داده
Deep Learning	یادگیری ژرف
Deep Neural Networks	شبکه‌های عصبی ژرف
Defect / Bug	نقص
Dendrite	دارینه
Denial of Service (DoS)	ممانعت از سرویس
Design Fault	خطای طراحی
Design Matrix	ماتریس طراحی
Directed Acyclic Graph (DAG)	گراف جهت دار بدون دور
Directory-Based	مبتنی بر دایرکتوری
Distribute Denial of Service (DDos)	ممانعت از سرویس توزیع‌شده
Domain Specific Language	زبان خاص دامنه
Dump	روبرداری
Dynamic Taint	آلودگی پویا
Encoder-Decoder	کدگذار-کدگشا
End to End	انتها به انتها
Epoch	دوره
Error	اشکال
Error Function	تابع خطا
Evolutionary	تکاملی
Exploit	بهره‌برداری
Failure	خرابی
Fault	خطا
Feature	ویژگی
Feedforward Network	شبکه رو به جلو
File Format	قالب فایل
Functional Testing	آزمون عملیاتی
Fuzz Testing	آزمون فازی
Fuzzer	فازر
Fuzzing	فازینگ
Generalization	تعمیم‌پذیری
Generation-Based	مبتنی بر تولید
Generative Model	مدل مولد
Generator Program	برنامه مولد
Glass-Box	جعبه‌شیشه‌ای
Gradient Decent	کاهش گرادیان
Graph Coverage	پوشش گراف

Gray-Box	جعبه‌خاکستری
Heuristic	اکتشافی (آوینی)
Inference	استنتاج
Input Space Coverage	پوشش فضای ورودی
Instrumenting	ابزار گذاری
Integrity	جامعیت
Knowledge Base	پایگاه دانش
Label	برچسب
Logic Coverage	پوشش منطق
Logistic Regression	رگرسیون لَجیستیک
Long-Short Term Memory (LSTM)	حافظه طولانی مدت
Loss Function	تابع گمشدگی
Machine Learning (ML)	یادگیری ماشین
Machine Translation (MT)	ترجمه ماشینی
Magic Bytes	بایت‌های جادویی
Marker	نشان‌گر
Mean Absolute Error	خطای مطلق میانگین
Mean Squared Error	خطای میانگین مربعات
Mechanism	سازوکار
Model Complexity	پیچیدگی مدل
Module	پیمانه
Multi-Layer Perceptron	پرسپترون چندلایه
Mutation-Based	مبتنی بر جهش (جابه‌جایی)
Nested Condition	شرط تودرتو
Neuron	نورون (عصب)
Noise	اختلال
Objective Function	تابع عینی (تابع هدف)
Offset	آدرس نسبی
Optical Character Recognition (OCR)	تبدیل عکس به نوشته
Overfitting	بیش‌برازندگی
Overflow	سرریز
Parse	پویش
Parser	پویشگر
Pass	گذر
Path Coverage	پوشش مسیر
Pattern	الگو
Perceptron	پرسپترون

Platform	سکو
Pointer	اشاره‌گر
Raw Data	داده‌های خام
Recognizer Program	برنامه شناسنده
Rectified Linear Unit (ReLU)	تابع یکسوساز
Recurrent Neural Networks (RNN)	شبکه‌های عصبی مکرر
Regularization	منظم‌سازی
Reinforcement Learning	یادگیری تقویتی
Render	پرداخت
Scalability	مقیاس‌پذیری
Semi-Supervised	نیمه نظارتی
Sequence	توالی
Sequence to Sequence	توالی‌به‌توالی
Shift	شیفت
Sigmoid	سیگموید
Smart Fuzzing	فازینگ هوشمند
Softmax Function	تابع بیشینه هموار
Software Under Test (SUT)	نرم‌افزار تحت آزمون
Specification	مشخصه
Statement Coverage	پوشش جمله
Stochastic Gradient Decent (SGD)	کاهش گرادیان تصادفی
Structural Testing	آزمون ساختاری
Supervised Learning	یادگیری بانظارت
Supervisor (Teacher)	ناظر (راهبر)
Support Vector Machine (SVM)	ماشین بردار پشتیبان
Symbolic	نمادین
Synapse	همایه
Taint	آلودگی
Target	هدف
Task	وظیفه
Template	قالب
Test Case	مورد آزمون
Test Data	داده آزمون
Test Set	مجموعه آزمون
Training Set	مجموعه آموزش
Undecidable	تصمیم‌ناپذیر
Unexpected Input	ورودی ناخواسته

Unit	واحد
Unsupervised Learning	یادگیری بدون نظارت
Valid	معتبر
Validation	اعتبارسنجی
Validation Set	مجموعه تأیید
Vanilla Network	شبکه وانیلی
Vulnerability	آسیب‌پذیری
Weight Decay	فرسایش وزن
Well-Formed	خوش-شکل
White-Box	جعبه‌سفید
Zero-Day	روزِ صفر

❊❊❊

Abstract

Fuzz testing techniques require specific test data to discover software faults and vulnerabilities. Here, the main difficulty is the complexity of real-world applications and their inputs. On the one hand, if all the execution paths are not covered, those vulnerabilities laid in the uncovered paths could not be detected. On the other hand, checking all execution paths in the program is a time consuming and expensive process. Moreover, surveys show that most of the test cases go through the same superficial paths. Hence, it is observed that typical fuzz testing has a poor code coverage. There are also fuzzers that have a relatively better code coverage but are not scalable.

To solve the above issues, we need to study various methods of test data generation, especially for data with highly complex structure. Most of the real world applications, such as web browsers, take a file with specific format as input. Well-formed file generation make it possible to penetrate the depth of the program, and cover new paths. In this way, the possibility of finding faults and vulnerabilities increase.

Pure random and mutation-based methods are not suitable for complex input structures. Generation–based methods provide high code coverage but they are time consuming, expensive and based on available specifications of file format. Heuristic and application-aware methods, which are common in research, have solved some problems, but they are quietly slow fuzzing strategies. In this study, the mentioned methods are introduced and compared. Also deep learning based method and RNN generative models that are suitable solutions in learning complex task are discussed. Finally, we can say AI based solutions, especially machine learning, are the state of the art of file format fuzzers and have a clear horizon to deal with existing and emerging challenges.

Keywords: Fuzz testing, test data, code coverage, deep learning, neural networks.

Iran University of Science and Technology
School of Computer Engineering

A survey of automatic test data generation methods for using in file format fuzzers

A Thesis Submitted in Partial Fulfillment of the Requirement for the Degree of Master of Science in Computer Engineering - Software Engineering

By:
Morteza Zakeri Nasrabadi

Supervisor:
Dr. Saeed Parsa

October 2017