دانشگاه صنعتی شریف

Skein Hashing

پروژه مستندسازی پیاده‌سازی سخت افزاری الگوریتم به زبان *Verilog*

گروه ۴

سید پارسا اسکندر

کیمیا یزدانی

الهه خدایی

وحید زهتاب

آروین آذرمینا

استاد: فرشاد بهاروند

بهار ۹۸

فهرست مطالب

فصل ۱

معرفی الگوریتم Skein

۱.۱ مقدمه

دردنیای امروز، با افزایش لحظه‌ای اطلاعات در جهان، روز به روز رمزنگاری و رمزگذاری اطلاعات اهمیت دوچندانی پیدا می‌کند.برای مثال برقراری امنیت سیستم‌ها و شبکه‌های رایانه‌ای، ذخیره‌ی اطلاعات مهم و حساس و ... همگی مثال‌هایی هستند که بدون رمزنگاری و رمزگذاری ممکن نخواهند بود. بدون اغراق، بدون رمزگذاری و رمزنگاری، اینترنت به شکلی که امروز وجود دارد به هیچ عنوان وجود نمی‌داشت. راه‌های بسیار متفاوتی برای رمزنگاری و رمزگذاری وجود دارد، یکی از مهم‌ترینِ آنها، استفاده از توابع رمزگذاریِ بر پایه ی درهم‌سازی (*Hashing*) می باشد. درهم‌سازی به خودی خود پرکاربرد ترین داده‌ساختارِ استفاده شده در علوم رایانه‌ای است. برخی از توابع درهم‌سازی شامل ویژگی‌هایی هستند که آنهارا برای استفاده برای کاربردهایی چون رمزنگاری بسیار مناسب می‌کند. مهم‌ترین و پرکاربردترین این توابع، توابعی از خانواده‌ی *SHA* یا *Secure Hashing Algorithms* می‌باشند، توابعی چون MD-5 ، SHA-0 ، SHA-1 ، SHA-2 و SHA-3 که هر کدام خود شامل خانواده ای از توابع مخصوص کاربردهای خاص خود هستند.

توابع خانواده ی SHA توسط گروهی به نام *NIST* که کوتاه شده‌ی عبارت *National Institute of Standards and Technology* است، برگزیده می‌شوند. برای انتخاب هریک از ورژن های توابع این خانواده، ابتدا توابعی پیشنهاد شده، پس از آن تعدادی از آنها به عنوان فینالیست توسط NIST اعلام شده و در نهایت از بین فینالیست‌ها، یک تابع به عنوان ورژن جدید از توابع SHA معرفی می‌شود.

تابع مورد بررسی در این مقاله یکی از توابع فینالیست برای انتخاب SHA-3 می‌باشد که Skein نام دارد.

۲.۱ معرفی اجمالی الگوریتم

الگوریتم *Skein* یکی از خانواده های توابع درهم سازی است که براساس اندازه‌ی بلاک‌های داخلی سه نوع مختلف ۲۵۶ ، ۵۱۲ و ۱۰۲۴ بیتی دارد . الگوریتم مورد بررسی در این مقاله مربوط به اندازه‌ی داخلی ۵۱۲ بیتی آن یعنی skein512 می‌باشد. در بین این سه نوع کلی از توابع skein ، skein512 به عنوان تابع اصلی به کار می‌رود اما لازم به ذکر است که سرعت skein1024 دو برابر skein512 می‌باشد و skein256 زمانی به کار می رود که نیازمند حجم کمی از رم (حدود ۱۰۰ بایت) باشد .

درحالت کلی توابع skein توانایی درهم‌سازی ورودی به هر اندازه اندازه‌ای را دارد اما اندازه‌ی خروجی آن، معمولا یکی از حالت های ۲۵۶ یا ۵۱۲ یا ۱۰۲۴ بیتی است.

۳.۱ روند اجرایی الگوریتم

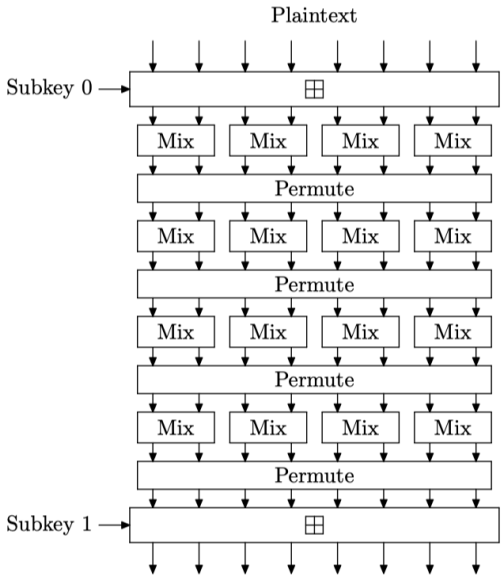
ایده ی اصلی توابع Skein استفاده از *Tweakable Block Ciphers* یا بلاک های رمزگذاری قابل تنظیم است . همه‌ی توابع skein از سه بخش کلی تشکیل شده اند:

- ***Threefish*** یا بلاک های رمزنگاری قابل تنظیم
- ***Unique Block Iteration (UBI)***
- ***Optional Argument System***

این سه بخش در کنار هم توابع درهم‌سازی skein را تشکیل می‌دهند. در ادامه به تفصیل به عملکرد هریک از این بخش ها خواهیم پرداخت.

۱.۳.۱ عملکرد Threefish

درالگوریتم های skein بسته به نوع تابع درهم‌سازی از بلاک‌های رمزگذاری استفاده می‌شود که به صورت زنجیره‌ای به یکدیگر متصل شده‌اند، اندازه‌ی این بلاک‌ها بسته به نوع الگوریتم ۲۵۶، ۵۱۲ یا ۱۰۲۴ بیت یا درواقع ۴، ۸ یا ۱۶ بسته‌ی ۶۴ بیتی از داده ها می‌باشد. بلاک های رمزنگاری هرکدام از ترکیب دو تابع غیر خطی به نام های درهم‌سازی (*Mix*) و جابه‌جایی (*Permutation*) تشکیل شده‌اند، هر بلاک رمزگذاری با بلاک‌های رمزگذاری دیگر سری شده و زنجیره ای از بلاک‌های رمزگذاری را تشکیل می‌دهند. علاوه براین بلاک‌ها، میان هر ۴ بلاک متوالی به مقادیر محاسبه شده تا آنجا مقادیر کلید هایی مربوط با آن دوره، افزوده می‌شود که به آنها *Subkey* گفته‌میشود. در تصویر زیر شمایی کلی از فرایند توضیح داده‌شده قابل مشاهده است:



شکل ۱.۱: شمای کلی از کارکرد بلاک‌های رمزگذاری قابل تنظیم

درادامه به توضیح هریک از این توابع می‌پردازیم.

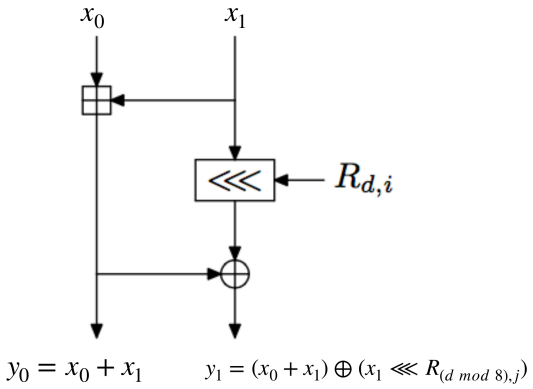
تابع درهم‌سازی (Mix)

این تابع غیر خطی دو بسته‌ی ۶۴ بیتی از داده‌ها را به عنوان ورودی دریافت کرده و دو بسته‌ی ۶۴ بیتی دیگر که حاصلی از ترکیب دو بسته‌ی ورودی هستند را در خروجی تحویل می‌دهد.اگر بسته های ورودی به این تابع به ترتیب ارزش، x_0 و x_1 باشند، بسته‌های خروجی این تابع از فورمول‌های زیر به دست می‌آیند:

$$y_0 = x_0 + x_1$$

$$y_1 = (x_0 + x_1) \oplus (x_1 \lll R_{(d \bmod 8),j})$$

و از لحاظ ساختار کلی، شمای حرکت داده در این تابع به شکل زیر خواهد بود:



شکل ۲.۱: شمای کلی از کارکرد تابع درهم‌سازی (Mix)

عدد d شمارنده‌ی بلاک‌های رمزگذاری می باشد که از صفر شروع شده است و عدد j معرف شمارنده‌ی توابع درهم‌سازی (Mix) داخل هر بلاک رمزگذاری است به صورتی که عدد j مربوط به تابعی که پرارزش‌ترین بسته‌های ورودی را دریافت می کند صفر و عدد j مربوط به تابعی که کم‌ارزش‌ترین بسته‌ها را به عنوان ورودی دریافت می‌کند برابر $N_W/2 - 1$ باشد، که در آن N_W تعداد بسته‌های موجود در بلاک‌های رمزگذاری است.

عدد $R_{(d \bmod 8),j}$ تعداد چرخش به چپ‌های لازم برای بسته‌ی ورودی دوم در تابع درهم‌سازی (Mix) jام در بلاک رمزگذاری dام را مشخص می کند که مقدار آن به ازای مقادیر مختلفی از j و d و N_W در جدول زیر قابل مشاهده است:

N_w		4		8				16							
j		0	1	0	1	2	3	0	1	2	3	4	5	6	7
	0	14	16	46	36	19	37	24	13	8	47	8	17	22	37
	1	52	57	33	27	14	42	38	19	10	55	49	18	23	52
	2	23	40	17	49	36	39	33	4	51	13	34	41	59	17
$d =$	3	5	37	44	9	54	56	5	20	48	41	47	28	16	25
	4	25	33	39	30	34	24	41	9	37	31	12	47	44	30
	5	46	12	13	50	10	17	16	34	56	51	4	53	42	41
	6	58	22	25	29	39	43	31	44	47	46	19	42	44	25
	7	32	32	8	35	56	22	9	48	35	52	23	31	37	20

شکل ۳.۱: جدول حاوی تعداد چرخش‌های صورت گرفته در تابع درهم‌سازی (Mix)

تابع جابه‌جایی (Permutation)

اگر فرض کنیم که خروجی های تابع درهم‌سازی (Mix) j در بلاک رمزگذاری d ام، $f_{d,\,۲j}$ و $f_{d,\,۲j+۱}$ باشند، خروجی نهایی بلاک رمزگذاری یا در واقع ورودی بلاک رمزگذاری بعدی، برابر خروجی تابع غیر خطی جابه‌جایی (Permutation) روی این مقادیر است که از رابطه ی زیر به دست می‌آید:

$$v_{d+۱,\,i} = f_{d,\,\pi(i)}$$

که v معرف بسته‌ی اطلاعاتی ۶۴ بیتی و عدد i شماره‌ی آن بسته در بلاک رمزگذاری مربوط و $\pi(i)$ یک تابع بوده که مقادیر آن در جدول زیر قابل مشاهده است:

	$i=$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$N_w =$ 4	0	3	2	1												
8	2	1	4	7	6	5	0	3								
16	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

شکل ۴.۱: جدول حاوی مقادیر $\pi(i)$ برای محاسبه‌ی خروجی تابع جابه‌جایی (Permutation)

عملیات افزودن مقادیر کلیدها (Subkeys)

علاوه بر دادگان ورودی (فرضا $p_{۰}$ تا $p_{N_W-۱}$ بسته های ۶۴ بیتی ورودی به کل بخش Threefish می‌باشند.) مقادیری به عنوان کلیدهای رمزگذاری ($k_{۰}$ تا $k_{N_W-۱}$ که خود بسته‌های ۶۴ بیتی اند) و دو بسته‌ی ۶۴ بیتی $t_{۰}$ و $t_{۱}$ به عنوان تنظیم (Tweak) نیز به بخش Threefish در این الگوریتم داده می‌شوند. این مقادیر اضافه پیش از هر ۴ بلاک رمزگذاری با مقادیر خروجی از بلاک رمزگذاری قبل ترکیب می‌شوند. درواقع ورودی بلاک رمزگذاری d ام از رابطه ی زیر به دست می‌آید:

$$e_{d,i} := \begin{cases} (v_{d,i} + k_{d/4,i}) \bmod 2^{64} & \text{if } d \bmod 4 = 0 \\ v_{d,i} & \text{otherwise} \end{cases}$$

که i در شماره‌ی بسته‌ی ۶۴ بیتی اطلاعات است و $v_{۰,i}$ ها همان بسته های ۶۴ بیتی ورودی به کل بخش Threefish یعنی $p_{۰}$ تا $p_{N_W-۱}$ می‌باشند و مقدار k ها نیز باتوجه به روابط زیر قابل مقایسه است:

$$\begin{aligned}
k_{s,i} &= k_{(s+i) \mod N_W+1} && i = 0,\ 1,\ 2,\ \ldots,\ N_W-4 \\
k_{s,5} &= k_{(s+5) \mod N_W+1} + t_{s \mod 3} \\
k_{s,6} &= k_{(s+6) \mod N_W+1} + t_{(s+1) \mod 3} \\
k_{s,7} &= k_{(s+7) \mod N_W+1} + s
\end{aligned}$$

که در این روابط مقادیر k_{N_W} و $t_{۲}$ از قرار زیر اند:

$$k_{N_W} = C_{۲۴۰} \oplus k_{۰} \oplus k_{۱} \oplus \ldots \oplus k_{N_W-۱}$$

$$t_{۲} = t_{۱} \oplus t_{۰}$$

که $C_{۲۴۰}$ عددی ثابت و برابر 0x1BD11BDAA9FC1A22 است و به آن جهت در فرمول وجود دارد که از ۰ نبودن تمامی بیت‌ها اطمینان حاصل شود.

جمع‌بندی

بخش‌هایی که توضیح داده شدند درکنار یک دیگر تشکیل یک Threefish را خواهند داد، بر اساس نوع و اندازه‌ی تابع skein تعداد بلاک‌های‌رمزگذاری ای که به یکدیگر باید متصل شوند متغیر است، این تعداد برای skein-512 برابر ۷۲ عدد و برای skein-1024 ۸۰ عدد می باشد.

۲.۳.۱ Unique Block Iteration (UBI)

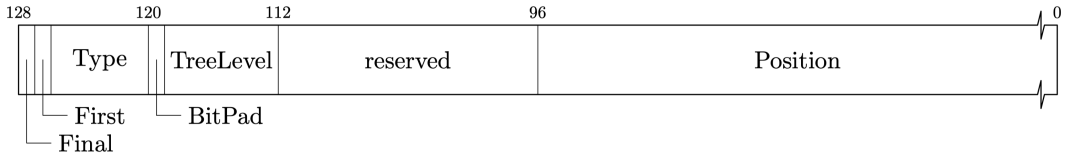
هدف اصلی UBI تولید خروجی با اندازه ی ثابت و مستقل از سایز ورودی تابع skein است.(برای مثال همواره سایز رشته ی هش شده برابر با ۶۴ بیت باشد.) UBI یک حالت زنجیره ای است که برروی بخش قبل یعنی Threefish Cipher Blocks پیاده می‌شود. تابع UBI سه ورودی دریافت می کند:

G مقداری آغازین به اندازه‌ی N_b بایت

M یک پیام با اندازه‌ای برابر نهایتا ۸ − ۲۹۹ بیت در ارائه شده به صورت بایت بایت.

T_s یک عدد صحیح با اندازه ی ۱۲۸ بیت که مقداری شروع کننده برای تنظیم (Tweak) است.

UBI هر رشته از اطلاعات (M) را با استفاده از مقدار یکتای تنظیم (Tweak) پردازش می کند . هر مقدار تنظیم (Tweak) از ۷ بخش تشکیل شده است که در کنار هم در تشکیل یک رشته می‌دهند. تصویر زیر نشان‌دهنده‌ی این مقادیر و اندازه و ترتیب قرارگیری‌شان می باشد:



شکل ۵.۱: نمودار نشان‌دهنده ی ساختار مقدار تنظیم (Tweak) در UBI ها

توضیحی مختصر از هریک از این مقادیر در جدول زیر قابل مشاهده است:

Name	Bits	Description
Position	0– 95	The number of bytes in the string processed so far (including this block)
reserved	96–111	Reserved for future use, must be zero
TreeLevel	112–118	Level in the hash tree, zero for non-tree computations.
BitPad	119	Set if this block contains the last byte of an input whose length was not an integral number of bytes. 0 otherwise.
Type	120–125	Type of the field (config, message, output, etc.)
First	126	Set for the first block of a UBI compression.
Final	127	Set for the last block of a UBI compression.

شکل ۶.۱: جدول معرفی فیلدهای مختلف موجود به عنوان تنظیم (Tweak) در UBI ها

اگر M یک رشته پیام باشد ، تا زمانی که پیام مضربی از اندازه ی بلاک باشد در قسمت bit pad صفر گذاشته می شود. p تعداد صفر ها است . طبق رابطه ی زیر $M"$ به k بلاک تقسیم می‌شود، که هر کدام از رشته ی اطلاعاتی N_b بیت دارند.

$$p := \begin{cases} N_b & \text{if } N_M = 0 \\ (-N_M) \bmod N_b & \text{otherwise} \end{cases}$$
$$M'' := M' \parallel 0^p$$

و در نهایت خروجی هر UBI از رابطه ای به فرم زیر محاسبه خواهد شد:

$$H_0 := G$$
$$H_{i+1} := E(H_i, \text{ToBytes}(T_s + \min(N_M, (i+1)N_b) + a_i 2^{126} + b_i(B2^{119} + 2^{127}), 16), M_i) \oplus M_i$$

۳.۳.۱ Optional Argument System

الگوریتم skein توانایی آنرا دارد که بدون محاسبات اضافی، ضمن محاسبات داخلی خود، کارهای جانبی دیگری نیز همچون تولید MAC و ... انجام دهد. در صورت نیاز می‌توان این بخش را به الگوریتم اضافه کرد تا با دریافت آرگومان های دلخواه به الگوریتم کمک کند تا کاربردهای ثانویه‌ای از خود نشان دهد، توضیح ساختار اجرایی دقیق این بخش به این مقاله نامربوط واز حوصله‌ی آن خارج است.

فصل ۲

پیاده‌سازی سخت‌افزاری با Verilog

۱.۲ مقدمه

دنیای نرم‌افزار و سخت‌افزار رایانه در نگاه کلی می‌توانند بسیار شبیه به هم باشند، برنامه های نرم‌افزاری، مقادیری را به عنوان ورودی دریافت کرده، سپس طی روند مشخصی محاسباتی روی آنها انجام داده و در نهایت مقادیری را به عنوان خروجی به کاربر خود تحویل می دهند، قطعات سخت‌افزاری نیز دارای port هایی برای ارتباط با دنیای خارجی و دریافت ورودی و تحویل خروجی‌های خود می‌باشند و از واحدهای مختلف پردازشی و عملیاتی مختلفی برای محاسبه ی خروجی‌های مناسب تشکیل شده اند. درعمل می‌توان سخت‌افزاری خاص برای اجرای بسیاری از روند های نرم‌افزاری طراحی و پیاده سازی کرد، ساخت سخت‌افزارِ خاصِ مربوط به یک الگوریتم می تواند کاربردهای بسیاری داشته باشد، برای مثال قطعه‌ای که بتواند داده های ورودی را رمزنگاری کند می‌تواند به صورت گسترده برای ذخیره ی اطلاعات به صورت سریع استفاده شود، سخت‌افزار های اختصاصیِ الگوریتم ها سریع و بهینه اند و میتوانند به اجرای هرچه سریع تر روندهای پیچیده‌ای که به الگوریتم مورد نظر وابستگی فراوان دارند کمک کلانی کنند.

همانند بسیاری از الگوریتم‌های رایانه‌ای، الگوریتم تابعِ Skein Hashing که در بخش قبل کلیتی از آن معرفی شد را می توان به صورت سخت افزاری پیاده‌سازی کرد، بدین صورت که قطعه‌ای طراحی و پیاده‌سازی کنیم که ورودی ای به اندازه ی دلخواه کاربر را دریافت و حاصل درهم‌سازی را به صورت خروجی ای به اندازه ی مورد نظر وی خروجی دهد. بر اساس نیاز و کاربرد کاربر از این قطعه، اندازه‌ی ورودی و خروجی را می توان ثابت و به مقدار دلخواه درنظر گرفت، سپس قطعه‌ای ثابت با پیاده سازیِ بهینه‌ای برای اندازه های مورد نظر طراحی کرد، یا این که قطعه‌ای برای ورودی و خروجی های با اندازه های متغیر طراحی و پیاده سازی کرد. همان طور که در بخش قبل توضیح داده شد، تابع درهم‌سازی Skein Hashing میتواند ورودی‌ای با اندازه ی دلخواه را دریافت کند و خروجی ای با اندازه‌ی دلخواه تحویل دهد، در این مقاله تمرکز روی پیاده سازی سخت‌افزاری حالتی از این تابع می باشد که اندازه ی بلاک های درونی تابع (حالت درونی تابع) **۵۱۲ بیت** و حاصل درهم‌سازی نیز به صورت خروجی ای به اندازه ی **۵۱۲ بیت** می باشد، این تابع در اصطلاح *Skein 512-512* نامیده می‌شود.

مراحل طراحی و پیاده‌سازی سخت‌افزاری یک قطعه معمولا به آن صورت است که برای اطمینان از کارکرد صحیح پیاده‌سازی، موازی با طراحی سخت‌افزاری قطعه، پیاده سازی دیگری از الگوریتم به نام مدل طلایی انجام می‌شود و پس از پایان طراحی‌ها، کارکرد قطعه با مدل طلایی مقایسه می شود تا قطعه‌ی نهایی مشکلی نداشته باشد. در بخش **مدل طلایی** به تفصیل درباره‌ی مدل طلایی استفاده شده در این پروژه توضیح داده شده است. در این بخش به پیاده‌سخت‌افزاری این الگوریتم به کمک زبانِ توصیف سخت‌افزارِ Verilog می پردازیم.

۲.۲ پیاده‌سازی

همان طور که توضیح داده شد، الگوریتم Skein از سه بخش اصلی تشکیل می‌شود:

- ***Threefish*** یا بلاک های رمزنگاری قابل تنظیم،
- ***Unique Block Iteration (UBI)*** که یک حالت زنجیره ای است که از Threefish به صورتی استفاده می کند که ورودیِ به اندازه‌ی دلخواه به خروجی‌ای به اندازه‌ای مشخص تبدیل شود.
- ***Optional Argument System*** که به الگوریتم توانایی پشتیبانی از بسیاری ویژگی های دلخواه را، بدون افزودن باری اضافه به پیاده سازی الگوریتم می‌دهد.

طراحی ای که در این پروژه به بررسی‌اش پرداخته شده است، یک طراحی بسیار ساده شده از الگوریتمِ Skein 512-512 می باشد. در این طراحی اندازه ی ورودیِ و خروجی اش ثابت و ۵۱۲ بیت می باشند، و اندازه ی حالت درونی تابع درهم‌سازی نیز دقیقا برابر اندازه‌ی ورودی و خروجی‌هاست، بنابراین در این طراحی اثری از پیاده‌سازی یک UBI پیچیده نیست. علاوه بر این، این پیاده‌سازی، پیاده‌سازی خام خود الگوریتم Skein 512-512 بوده و هیچ ویژگی اضافی ای را پشتیبانی نمی‌کند، بنابراین اثری از پیاده‌سازی Optional Argument System نیز در آن نیست. بنابراین طراحی، صرفا شامل بلاک های رمزنگاری قابل تنظیم بوده، داده‌ی ورودی به صورت مستقیم با این بلاک ها تزریق شده و خروجی الگوریتم نیز به صورت تقریبا مستقیم از آخرین بلاک دریافت می‌شود.

۱.۲.۲ بررسی درستی طراحی ارائه شده

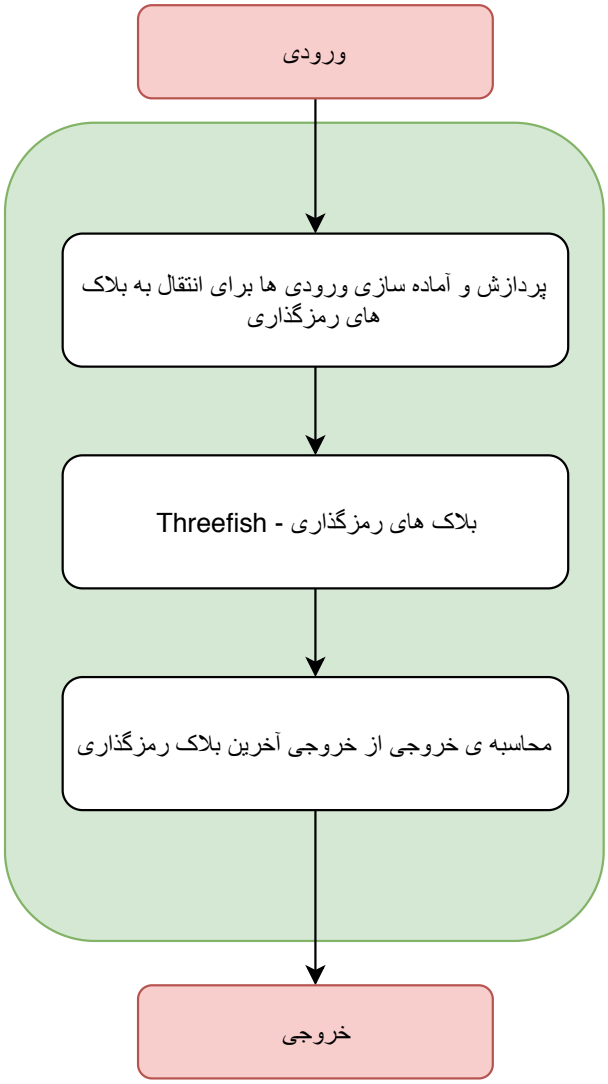
طراحی اولیه‌ای از الگوریتم که توسط تیم تدریس درس پیشنهاد شده بود (این فایل) ، شامل مشکلات منطقی فراوان بود، مشکلات موجود در هر خط از کد verilog به صورت یک comment در قالب

$$todo :< ValidCode >$$

مشخص شده اند، علاوه بر این طراحی تصحیح شده (این فایل) نیز در کنار مقاله پیوست شده است، تمامی مستند سازی های این مقاله، براساس پیاده سازی تصحیح شده ی کدهای اولیه می‌باشد.

۲.۲.۲ ساختار طراحی

ساختار کلی این الگوریتم شامل سه بخش کلی بوده که از وظایف ماژول اصلی این طراحی که skein512 نام گذاری شده است نیز می‌باشند این وظایف و بخش ها به صورت کلی در تصویر زیر قابل مشاهده اند:



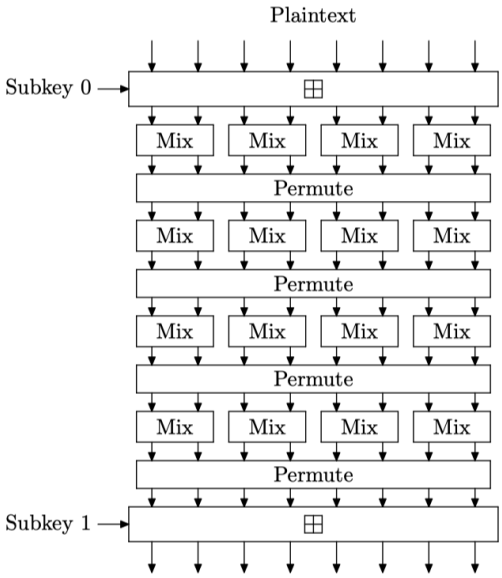
شکل ۱.۲: ساختار کلی پیاده‌سازی سخت‌افزاری

دو بخش پردازش ورودی و خروجی‌ها به صورت کامل در ماژول skein512 صورت می گیرند و به تفصیل درباره ی آنها توضیح داده خواهد شد، بخش بلاک های رمزگذاری از ماژول های skein_round و skein_round_1 تا skein_round_4 تشکیل شده و قرارگیری زنجیره‌ای آنها و محاسبه ی اولیه‌ی مقادیر کلیدهای رمزنگاری در ماژول skein512 صورت گرفته است.

۳.۲.۲ پیاده سازی بلاک‌های رمزگذاری

همان‌طور که در معرفی الگوریتم Skein Hashing در بخش اول توضیح داده شد، این الگوریتم برای تولید مقدار درهم‌سازی از بلاک های رمزگذاری که به صورت زنجیره ای یکی پس از دیگری قرار گرفته اند، استفاده می کند و این بلاک‌ها بخش عمده‌ای از طراحی سخت‌افزاری را دربر می‌گیرند.

شمای کلی حرکت داده داخل بلاک های رمزنگاری در این الگوریتم به شکل زیر می باشد:



شکل ۲.۲: شمایی از حرکت داده در بلاک های رمزنگاری

داده‌ی ورودی که به ۸ بخش ۶۴ بیتی تقسیم می‌شود، به بلاک‌های رمزنگاری تزریق شده و پس از ۴ مرحله ی متوالی از درهم‌سازی (Mix) و جابه‌جایی (Permutation) - که به هر یک از آنها یک $round$ می‌گوییم - با مقادیری به نام $Subkey$ که از کلیدهای رمزنگاری - که آنها نیز از ۸ بلاک ۶۴ بیتی تشکیل شده اند - به دست می‌آیند، جمع می‌شوند.

محاسبات دقیق Subkey ها و توابع غیرخطی درهم‌سازی (Mix) و جابه‌جایی (Permutation) هر round به تفصیل در بخش اول مقاله، توضیح داده شده است. آن چیزی که دراین میان حائز اهمیت است، استفاده‌ای هوشمندانه از نظم تکراری این توابع محاسباتی در پیاده‌سازی سخت‌افزاری مورد بررسی در این مقاله است.

پیاده سازی توابع جابه‌جایی (Permutation) هر round

تابع غیرخطیِ جابه‌جایی (Permutation) یک عملیات ثابت را روی مقادیر خروجی از توابع درهم‌سازی (Mix) انجام می‌دهد، این تابع در ماژول‌های skein_round_1 تا skein_round_4 به صورت توصیف رفتاری با کمک یک always block حساس به لبه‌ی بالارونده‌ی ساعت پیاده سازی شده است. مقادیر خروجی توابع درهم‌سازی (Mix) مربوط به هر ماژول - که جلوتر پیاده‌سازی آنها را بررسی خواهیم کرد -، به هنگام لبه‌ی بالارونده ی ساعت، جابه‌جا شده و به خروجی ماژول انتقال داده می‌شوند.

توصیف ارائه شده از جابه‌جایی (Permutation) در این always block ها در واقع معرف مجموعه ای از ۵۱۲ D-FlipFlop حساس به لبه‌ی بالارونده ی ساعت می باشد.

پیاده سازی توابع درهم‌سازی (Mix) هر round

برخلاف توابع جابه‌جایی (Permutation) که یک عملیات ثابت را در هر round اجرا می‌کنند، این توابع غیرخطی، بر اساس این که در کدام round قرار دارند، محاسبات خاص خود را خواهند داشت، همان طور که در بخش قبل توضیح داده شد، هر درهم‌سازی (Mix) شامل، یک جمع، یک گردش به چپ و یک xor می باشد. جمع و xor ها در همه‌ی round ها یکسان اند و به صورت یکتا پیاده می‌شوند.

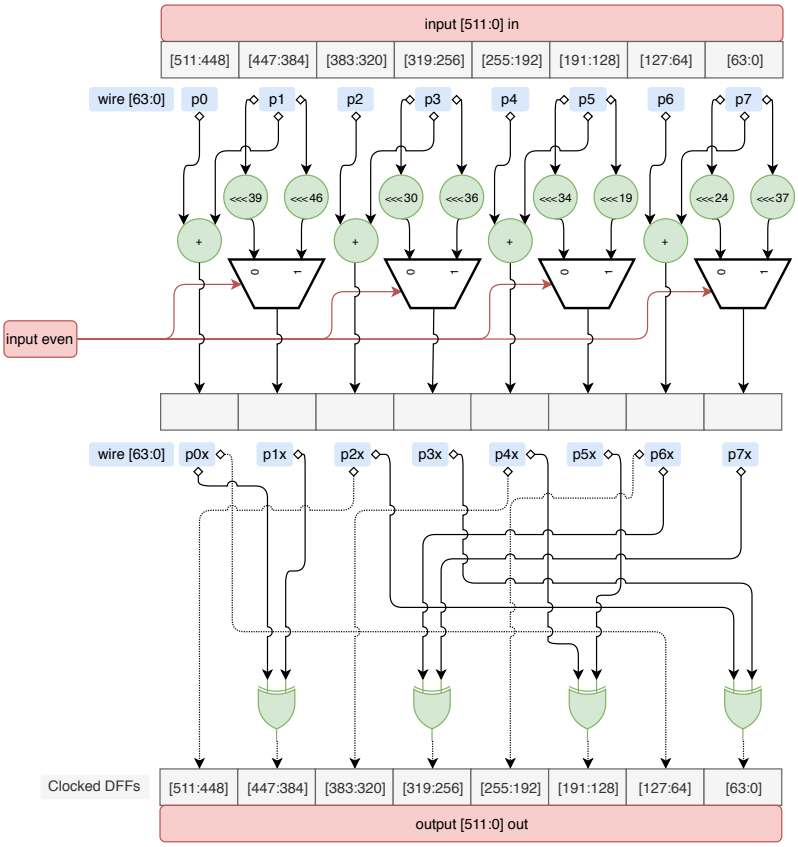
چیزی که در هر round متفاوت خواهد بود، تعداد گردش ها به چپ می‌باشد، با توجه به مقادیر ارائه شده و فورمول‌های توضیح داده شده در بخش قبل، این مقادیر هر ۸ round تکرار می شوند، مسئله ی دیگر آن است که قرار است هر ۴ round، subkey های مربوطه به مقادیر موجود در بلاک‌های رمزگذاری افزوده شوند، برای همین در پیاده‌سازی مورد بررسی در این پروژه، هر ۴ round در یک ماژول به نام skein_round در نظر گرفته شده است و پیاده سازی خود round ها در ماژول های skein_round_1 تا skein_round_4 تعریف شده اند. این بدین معنا است که برای دوره ی تناوب ۸ تایی توابع درهم‌سازی (Mix) هر round ، صرفا دانستن این که در ۴ round شماره ی فرد یا زوج قرار داریم برای ماژول های skein_round_1 تا skein_round_4 کافی می باشد. برای همین این ماژول‌ها یک ورودی تک بیتی به نام even دریافت می کنند و بر اساس آن تعداد گردش به چپ‌های مناسب را بر می گزینند.

پیاده‌سازی خود توابع درهم‌سازیِ (Mix) هر round، در دو بخش در ماژول های skein_round_1 تا skein_round_4 تعریف شده است. در بخش اول عملیات‌های جمع و گردش به چپ‌ها به توصیفی ساختاری و به کمک یک سری continuous assignment در این ماژول‌ها مشخص شده اند، لازم به ذکر است که در verilog عملگر گردش به چپ وجود ندارد، برای همین برای پیاده سازی گردش به چپ یک vector، از ترکیب part selection و concatenation استفاده شده است. در بخش دوم عملیات‌های xor هم‌زمان با عملیات‌های جابه‌جایی (Permutation) در always block مربوط به آنها در این ماژول‌ها معرفی شده است.

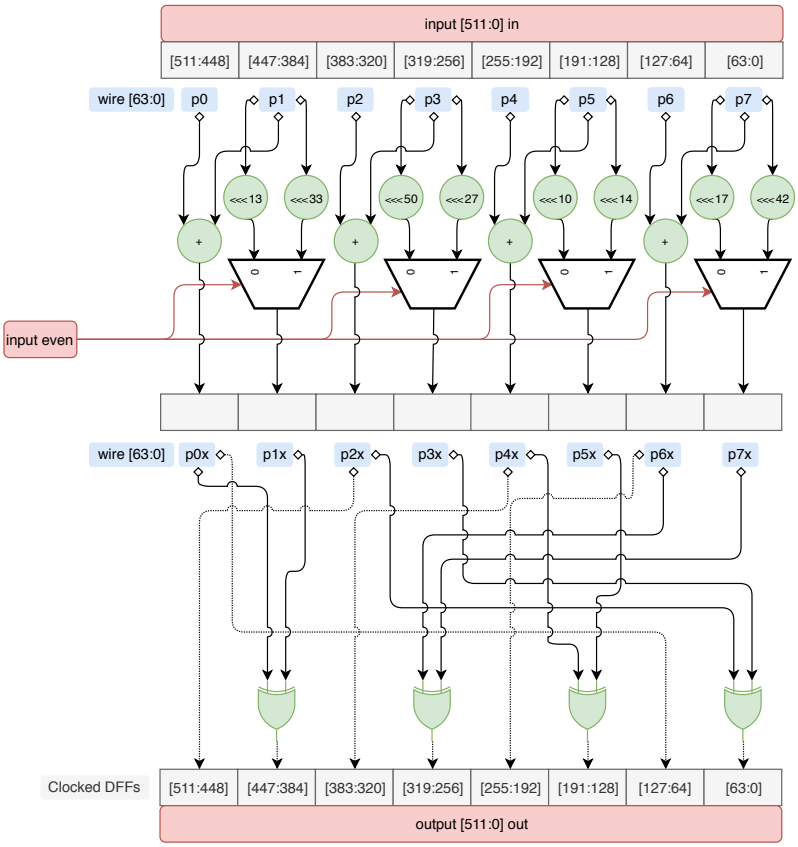
توصیف ارائه شده از توابع درهم‌سازی (Mix) در عمل معرف مجموعه ای از مدارهای ترکیبی (combinational) می باشد و منجر به تولید گیت‌های منطقی مستقل از سیگنال ساعت خواهد شد.

پیاده سازی round ها

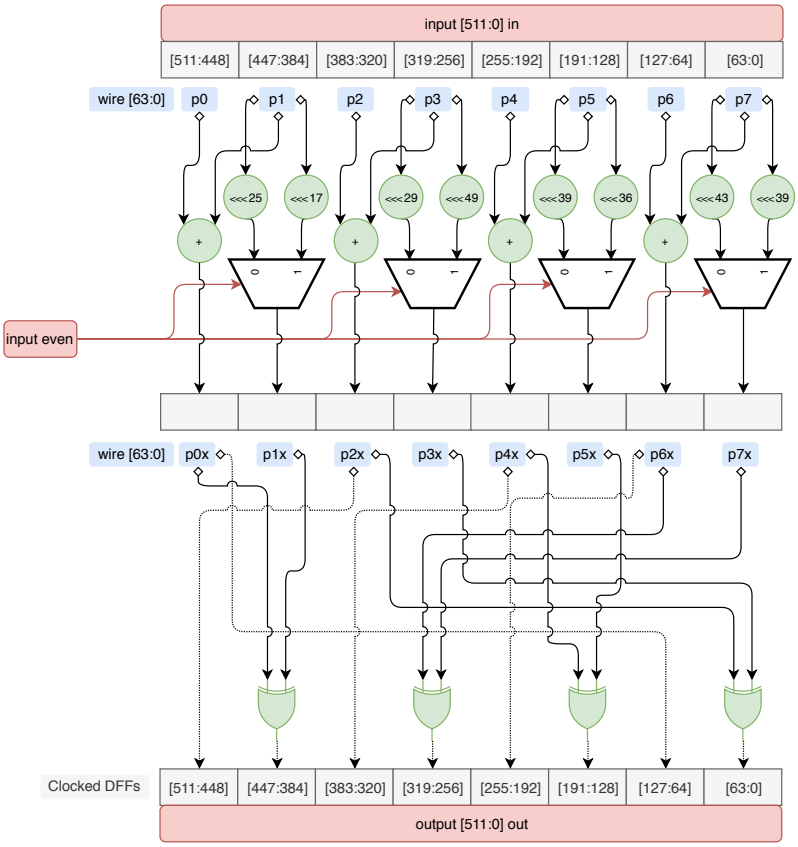
همان طور که بار ها گفته شد round شامل درهم‌سازی (Mix) و جابه‌جاییِ (Permutation) روی ورودی‌هایش می‌باشد که به تفصیل به پیاده سازی این دو تابع غیرخطی در طراحی مورد بررسی این پروژه پرداختیم، تنها مسئله ی مجهول شیوه‌ی اتصال این دو بخش در ماژول های skein_round_1 تا skein_round_4 و تشکیل واحد های محاسباتی round های الگوریتم می‌باشد که بلاک دیاگرام های تهیه شده در تصاویر دو صفحه‌ی بعدی به تفصیل این مسئله و هم‌چنین شمای کلی حرکت داده داخل این ماژول‌ها را توضیح می دهند:



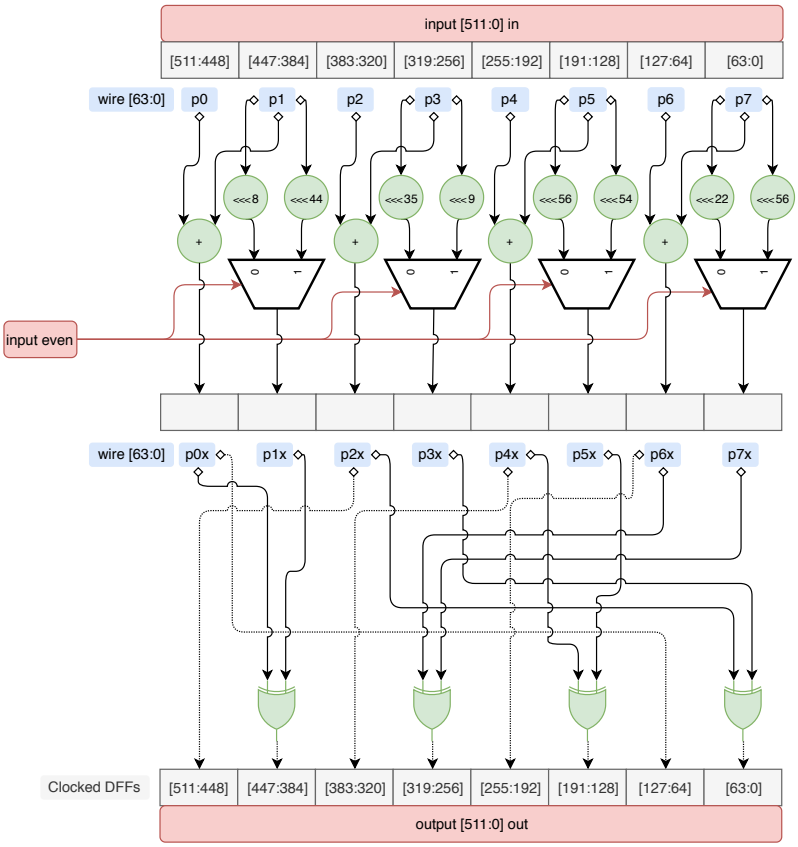
شکل ۳.۲: بلاک دیاگرام شماتیک و نحوه‌ی جریان داده در ماژول skein__round__1



شکل ۴.۲: بلاک دیاگرام شماتیک و نحوه‌ی جریان داده در ماژول skein__round__2



شکل ۵.۲: بلاک دیاگرام شماتیک و نحوه‌ی جریان داده در ماژول skein__round__3

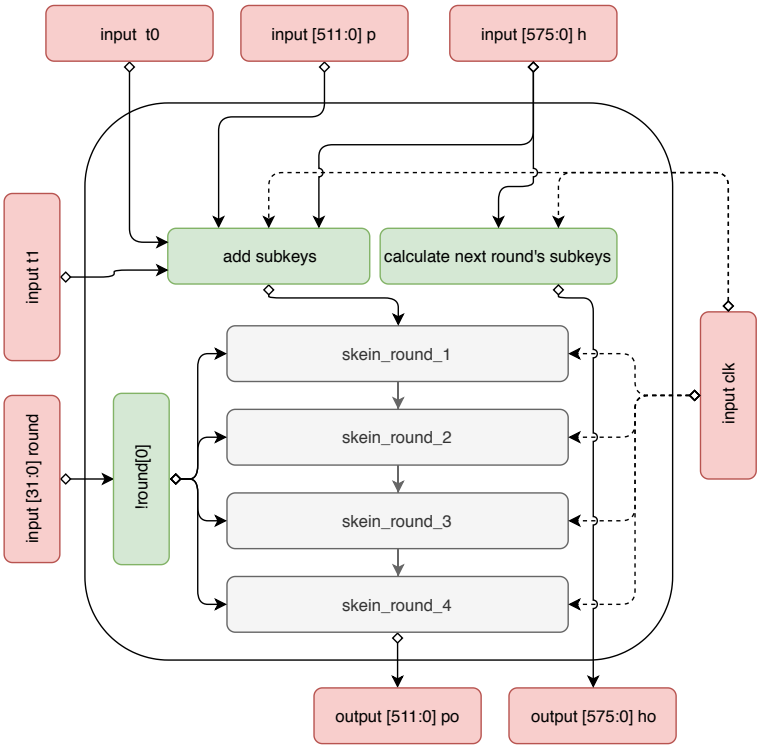


شکل ۶.۲: بلاک دیاگرام شماتیک و نحوه‌ی جریان داده در ماژول skein__round__4

محاسبه و افزودن Subkey ها

همان‌طور که اشاره شد، قبل از هر ۴ round مقادیری به نام Subkey به مقادیر محاسبه شده در بلاک‌های رمزگذاری افزوده می‌شود، طبق توضیحات بخش اول، مقادیر Subkey ها باتوجه به شماره‌ی round تنظیم می‌شوند، اگر به فورمول محاسبه‌ی Subkey ها توجه کنید متوجه دو تناوب خواهید شد، یکی برای خود مقادیر Subkey ها که پس از هربار محاسبه، انگار که یک خانه جابه‌جا شوند، دیگر آنکه پس از هر ۳ round جای مقادیر تنظیم tweak (t_0 ، t_1 ، t_2) یک دور کامل می‌چرخند.

ماژول skein_round شامل ۴ round متوالی از بلاک‌های رمزگذاری و واحد افزودن مقادیر Subkey ها به مقادیر محاسبه شده در بلاک‌های رمزگذاری قبلی و همچنین پیاده‌سازی یکی از این دو دوره‌ی تناوب می‌باشد. به صورت خلاصه تصویر زیر معرف شمای کلی و نحوه‌ی جریان داده ها در هر نمونه از ماژول‌های skein_round می‌باشد:



شکل ۷.۲: شمایی از نحوه‌ی حرکت داده و ساختار هر نمونه از ماژول skein_round

ورودی input [575:0] h به ترتیب از سمت باارزش‌ترین بیت، شامل Subkey های مربوط به این round یعنی:

$$Subkey_{round,\ ۰}, Subkey_{round,\ ۱}, Subkey_{round,\ ۲}, ..., Subkey_{round,\ ۸}$$

و ورودی های input t0 و input t1 به ترتیب حاوی مقادیر تنظیم (tweak) مربوط به این round ، یعنی:

$$t_۰ = t_{round\ mod\ ۳}$$

$$t_۱ = t_{(round+۱)\ mod\ ۳}$$

می باشد. بنابراین این ورودی ها بدون هیچ پیش‌پردازشی آماده‌ی تزریق به بخش افزایش به مقادیر رمزگذاری شده در round های قبلی می‌باشند. بخش افزاینده‌ی مقادیر Subkey ها به مقدار رمزگذاری شده از بلاک(ها)ی قبلی ، در این ماژول، با توجه به فورمول محاسباتی توضیح داده شده در بخش قبل، محاسبات مربوط به این round را از روی ورودی های خود انجام می‌دهد. پیاده‌سازی این بخش در کد با توصیفی رفتاری به کمک یک always block حساس به لبه‌ی بالارونده‌ی ساعت مشخص شده است. این توضیحات

بخش اول توصیفات موجود در این always block بوده و در عمل معرف ترکیبی از یک مداری ترکیبی (برای محاسبات جمع Subkeyهاو مقادیر رمزگذاری شده‌ی p) و مداری ترتیبی (برای انتقال حاصل عملیات‌های جمع به ورودی skein__round__1) می‌باشد.

بخش دیگری که در این ماژول پیاده‌سازی شده است، محاسبه‌ی Subkey های مربوط به round بعدی براساس Subkey های این round می‌باشد. اگر به فورمول های محاسبه‌ی Subkey ها توجه کنید، واضح است که پس از هر round انگار Subkey ها یک گردش به چپ دارند. دقیقا همین ایده در این ماژول به کمک توصیفی رفتاری در بخش دوم کد های always block حساس به لبه‌ی بالارونده‌ی ساعت، پیاده شده‌است. این توصیف در واقع معرف یک مدار ترکیبی برای محاسبه ی حاصل xor . $Subkey_{round,\,0}$ تا $Subkey_{round,\,7}$ و یک مدار ترتیبی برای انتقال مقادیر یک بلاک چرخش به چپ و حاصل xor Subkey ها به خروجی می باشد.

پیاده‌سازی و مقداردهی تناوبی مقادیر تنظیم (tweak) به هنگام نمونه‌گیری ماژول‌های skein__round در ماژول اصلی یعنی skein512 انجام شده است.

اتصال round ها به یکدیگر و پیاده‌سازی بلاک های رمزگذاری به صورت زنجیره‌ای

در پیاده‌سازی الگوریتم Skein 512-512 برای محاسبه‌ی مقدار درهم‌سازی نهایی باید ۷۲ round بلاک‌های رمزگذاری پشت‌سرهم به صورت یک زنجیره تکرار شوند، پیاده‌سازی این مسئله در ماژول skein512 که بالاترین ماژول طراحی مورد بررسیِ این پروژه است، صورت می‌گیرد.

این‌کار به صورت ترکیبی از توصیف های ساختاری و رفتاری در این ماژول انجام شده است، کنارهم قرارگرفتن بلاک‌های رمزگذاری به توصیفی ساختاری با نمونه گیری از ۱۸ ماژول skein__round انجام شده است. همان‌طور که در بخش قبل توضیح داده شد، محاسبات مربوط به Subkey ها، دارای دو تناوب هستند، یکی از این تناوب ها در محاسبات داخل خود ماژول های skein__round صورت می‌گیرد که به تفصیل درباره‌ی آن توضیح داده شد. تناوب دومِ محاسباتی مربوط به انتخاب تنظیم (tweak)مناسب بین سه تنظیم t_0 ، t_1 ، t_2 می‌باشد. این تناوب هنگام نمونه‌گیری ماژول های skein__round در ماژول skein512 صورت گرفته است و چرخش ۳ تا ۳ تای مقادیر تنظیم اختصاص داده شده به port های ماژول های skein__round ، به وضوح قابل مشاهده است.

در بخش دیگر، اتصالات این ماژول هایی که نمونه‌گیری می‌شوند توصیف شده اند. در یک always block حساس به لبه‌ی بالارونده‌ی ساعت، خروجی هریک از ماژول های skein__round به ورودی ماژول skein__round بعدی خود متصل شده است. این پیاده سازی در عمل مداری کاملا ترتیبی است و موجب قرارگیری یک سری D-FlipFlop حساس به لبه‌ی بالارونده‌ی ساعت بینِ ماژول های skein__round می‌شود که سرِ هر سیگنال بالارونده‌ی ساعت، خروجی هر skein__round را به ورودی skein__round بعدی منتقل می کند.

۴.۲.۲ پیاده‌سازی بخش پردازش ورودی اولیه و خروجی نهایی

همان‌طور که در توضیحات اخیر اشاره شد، ماژول اصلی طراحی، skein512 علاوه بر دربرداشتن کل بلاک‌های رمزگذاری، ورودی ها را برای تزریق به این بلاک ها پیش‌پردازش کرده و خروجی مناسب را از خروجی آخرین بلاک رمزگذاری تولید می‌کند. پیاده‌سازی این بخش بسیار واضح و سرراست است، از روی مقادیر nonce و data مقدار ورودی اولیه به بلاک های رمزگذاری محاسبه می شود که پیاده سازی این بخش عمدتا به صورت توصیفی رفتاری از مدار ترکیبی در یک always block صورت گرفته است. این always block مدار skein512 دارای دو حالت کلی است که با یک فلیپ فلاپ به نام phase مشخص شده است، مقدار کنونی phase phase__q و مقدار بعدی آن phase__d به کمک این معرفه‌ی حالت، مدار بین هر پالس ساعت دو عملکرد متفاوت تزریق کلیدها و تنظیمات و داده ها را به بلاک های رمزگذاری انجام می‌دهد.

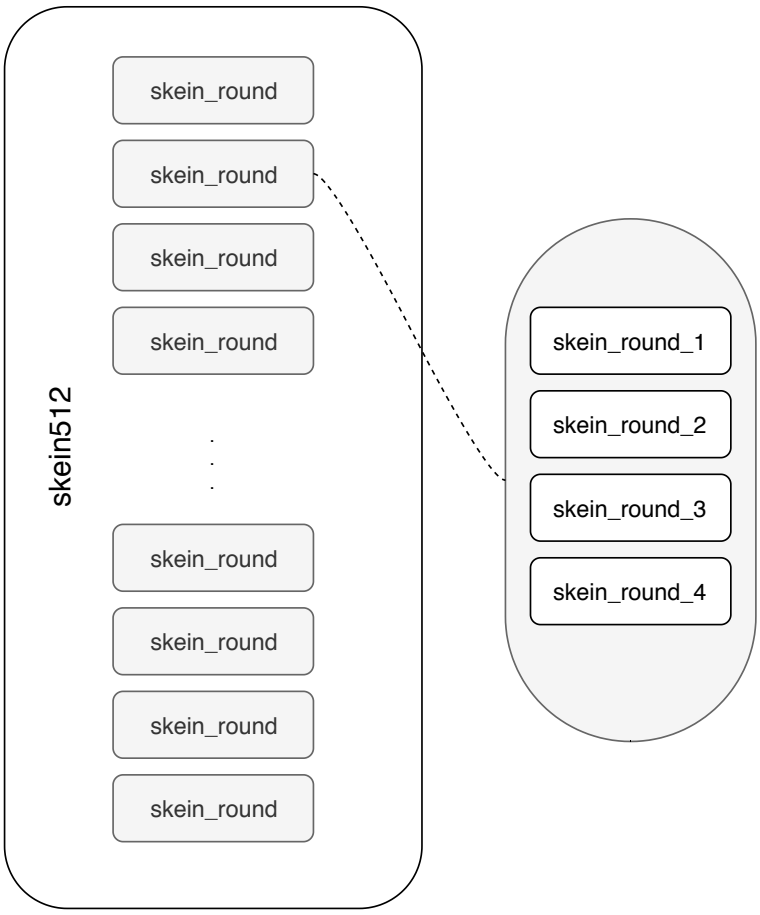
علاوه بر محاسبه ی ورودی و کلیدها و تنظیمات مناسب - از روی ورودی های ماژول - برای بلاک های رمزگذاری قابل تنظیم، ماژول skein512 از این بلاک‌های رمزگذاری خروجی ای به عنوان مقدار درهم‌سازی شده از ورودی ها به کاربر خواهد داد، پیاده‌سازی

محاسبات مربوط به استخراج خروجی نهایی همانند مجاسبات مربوط به پیش‌پردازش ورودی ها سرراست و ساده است، پس از محاسبه و افزودن یک سری دیگر از Subkey ها (کلیدهای طبقه‌ی شماره‌ی ۱۸ یا درواقع ۱۹ ام) به مقادیر خروجی از بلاک‌های رمزگذاری که توصیف آنها در بخش دوم always block معرف مدار ترکیبی ماژول آمده است، به کمک توصیف ساختاری و با استفاده از part selection و continuous assignment بایت‌های مقدار محاسبه‌شده پس از افزودن سری ۱۹ام کلیدها، جایگشت کاملا تازه ای به خود گرفته و به عنوان خروجی الگوریتم مشخص می شوند.

بنابراین بخش محاسبه‌ی خروجی نهایی ماژول، شامل یک بخش محاسبه و افزاینده ی کلیدهای سری ۱۹ام به مقادیر محاسبه‌شده در بلاک‌های رمزگذاری و بخشی برای جابه‌جایی مقدار محاسبه شده می‌باشد، این پیاده‌سازی ها در عمل کاملا معرف مدارهایی ترکیبی می‌باشد.

۵.۲.۲ جمع‌بندی

بنابر توضیحات ارائه شده در این بخش، ساختار کلی و دقیق طراحی سخت‌افزاری الگوریتم مشخص شد. از دید ساختار درختی، این طراحی از ۶ ماژول تشکیل تشکیل می‌شود که ارتباط کلی آنها در تصویر زیر قابل مشاهده‌است:



شکل ۸.۲: نموداری از ساختار درختی و روابط ماژول ها با یکدیگر در طراحی

۳.۲ شبیه‌سازی

در مراحل طراحی قطعات سخت‌افزاری، پیش از تولید نهایی قطعات، به کمک نرم‌افزارهای شبیه‌ساز، طراحی انجام شده شبیه‌سازی می شود. برای انجام شبیه‌سازی لازم است نحوه ی ورودی و خروجی گرفتن از طراحی، مشخص شود. این‌کار به کمک طراحی جداگانه‌ای به نامِ *Test Bench* صورت می‌گیرد. برای شبیه‌سازی از یک تست بنچ تغیر یافته از همان تست‌بنچ پیشنهادی تیم تدریس درس

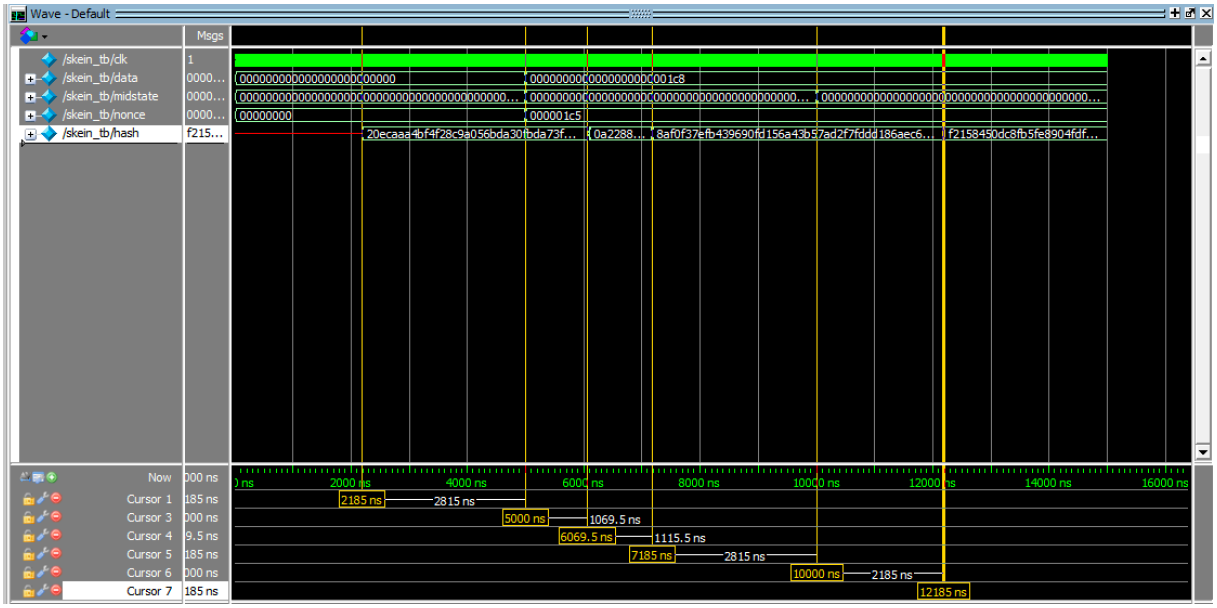
استفاده شد (این فایل) مدت زمان شبیه‌سازی در این تست‌بنچ ۱۵۰۰۰ نانو ثانیه (۱۵۰۰ کلاک) درنظر گرفته شده است. برای شبیه‌سازی از نرم‌افزار *ModelSim* استفاده شده است.

۱.۳.۲ توضیح تست‌بنچ

تست‌بنچ طراحی شده ابتدا ورودی‌های صفر به قطعه‌ی skein512 می‌دهد، پس از ۵۰۰۰ نانو ثانیه یک‌سری ورودی و دوباره پس از ۵۰۰۰ نانو ثانیه یک‌سری ورودی دیگر به قطعه می‌دهد. یکی از نتایجی که از این شبیه‌سازی به دست می‌آید، میزان زمانی است که طول می‌کشد تا خروجی مناسب توسط قطعه تولید شود، دیگر آن‌که خود خروجی داده شده معرف درستی یا نادرستی طراحی انجام شده است.

۲.۳.۲ نتایج شبیه‌سازی

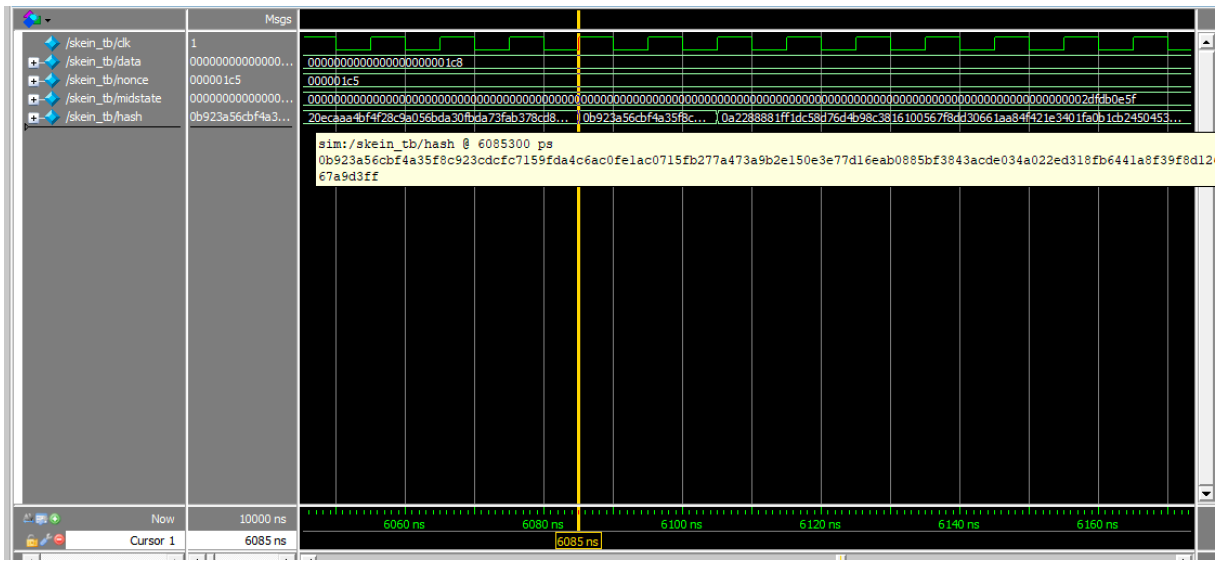
نتایج کلی این شبیه‌سازی در تصاویر زیر قابل مشاهده‌اند:



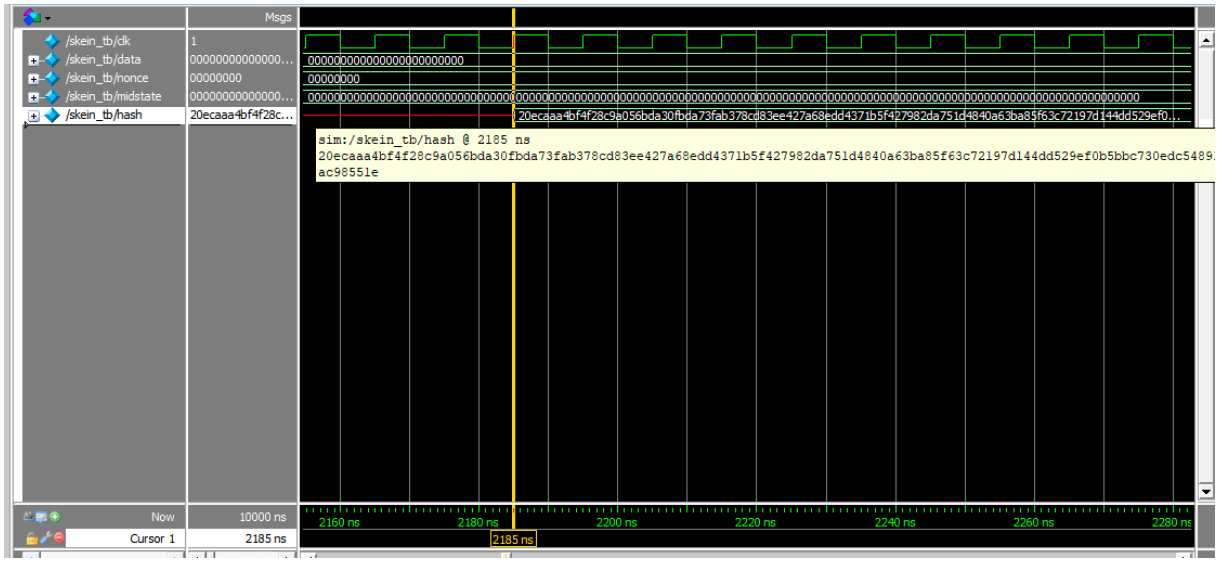
شکل ۹.۲: شکل موج مربوط به کل زمان شبیه‌سازی همراه با زمان‌های مهم

همان‌طور که مشخص است، هربار که ورودی قطعه تغیر می‌کند، تقریبا ۲۸۱۵ نانو ثانیه طول می‌کشد که خروجی قطعه به حالت پایدار و نهایی خود برسد و قبل از این زمان خروجی قطعه ممکن است چندباری تغیر کند و مقدار درهم‌سازی نهایی نباشد، بنابراین در عمل این قطعه پس از **۲۱۸ پالس ساعت** جواب تولید خواهد کرد و درواقع ۲۱۸ پالس ساعت تاخیر دارد.

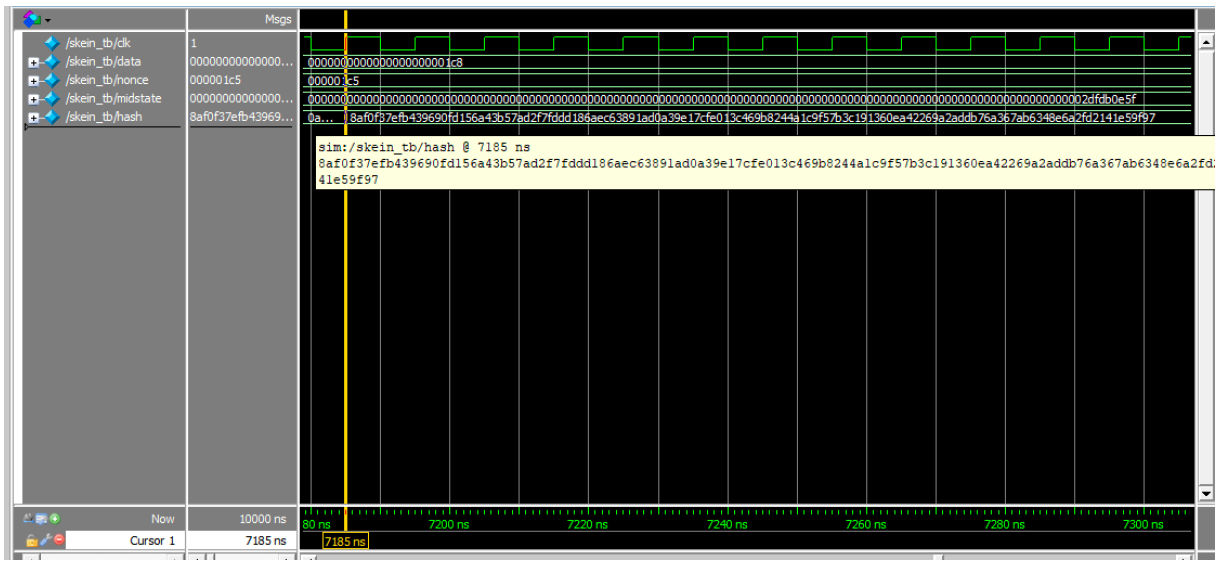
علت این مقدار تاخیر کاملا واضح است، طراحی انجام شده شامل ۷۲ round بوده و هر یک از این round ها دقیقا پس از ۴ کلاک خروجی خود را به بلاک بعدی انتقال می‌دهند. بنابراین ۲۱۸ = ۴ × ۷۲ کلاک طول خواهد کشید که جواب نهایی قطعه تولید شود، که این همان مقدار تاخیر مشاهده‌شده در شبیه‌سازی است.



شکل ۱۰.۲: تولید خروجی غلط پیش از گذشت زمان ۲۱۸ پالس ساعت از لحظه‌ی تغیر ورودی



شکل ۱۱.۲: پاسخ تولید شده برای ورودی‌های صفر، پس از گذشت ۲۱۸ پالس ساعت



شکل ۱۲.۲: پاسخ تولید شده برای ورودی‌های داده شده پس از ۵۰۰۰ نانو ثانیه، پس از گذشت ۲۱۸ پالس ساعت

شکل ۱۳.۲: پاسخ تولید شده برای ورودی‌های داده شده پس از ۱۰۰۰۰ نانو ثانیه، پس از گذشت ۲۱۸ پالس ساعت

۴.۲ سنتز

عمل سنتز به فرایند تبدیل طراحی سخت‌افزاری انجام شده با زبان‌های توصیف سخت‌افزار به لیستی از گیت‌های منطقی و ماژول هایی واقعی‌ای از رم و رام ها و غیره برای تولید مدارات و قطعات ASIC و یا قرارگیری روی سخت‌افزارهای از پیش‌آماده و قابل برنامه‌ریزی چون FPGAها گفته‌می‌شود. این‌کار توسط ابزاری نرم‌افزاری به نام Synthesis tool انجام پذیراست.

برای سنتز این پروژه برای محیط مقصد xilinx ، از ISE Design suite استفاده شد. درپیاده‌سازی تصحیح شده از الگوریتم، ماژول اصلی دارای ۱۱۵۳ port ورودی و خروجی می‌باشد، با توجه به این‌که هر قطعه‌ی FPGA ای توان پشتیبانی از این تعداد I/O port را ندارد،برای سنتز، پس از بررسی چند قطعه و با توجه به محدودیت های licensing برخی از FPGA هایی که توان پیاده‌سازی این طراحی را داشتند، نهایتا از قطعه‌ی XC6VLX760 از خانواده‌ی Virtex6 برای سنتز استفاده شد.

مراحل سنتز این پیاده‌سازی دچار هیچ پیغام خطایی نشد که لزومی به ارائه‌ی راه‌کار برای رفع آن باشد و تمامی فایل های خروجی غیر حجیم این سنتز از این آدرس قابل دسترسی می‌باشند.

این فایل شامل گزارشی کلی درباره ی این سنتز، اعم از درصد استفاده از سخت‌افزارهای مختلف موجود در FPGA و غیره می باشد.

این فایل شامل گزارشی از پورت‌های استفاده شده در FPGA برای اتصال ورودی و خروجی های قطعه‌ی طراحی شده است.

این فایل شامل گزارشی از تنظیمات محیط انجام سنتز در ابزار سنتز است.

این فایل شامل گزارشی از مراحل placement و format می باشد.

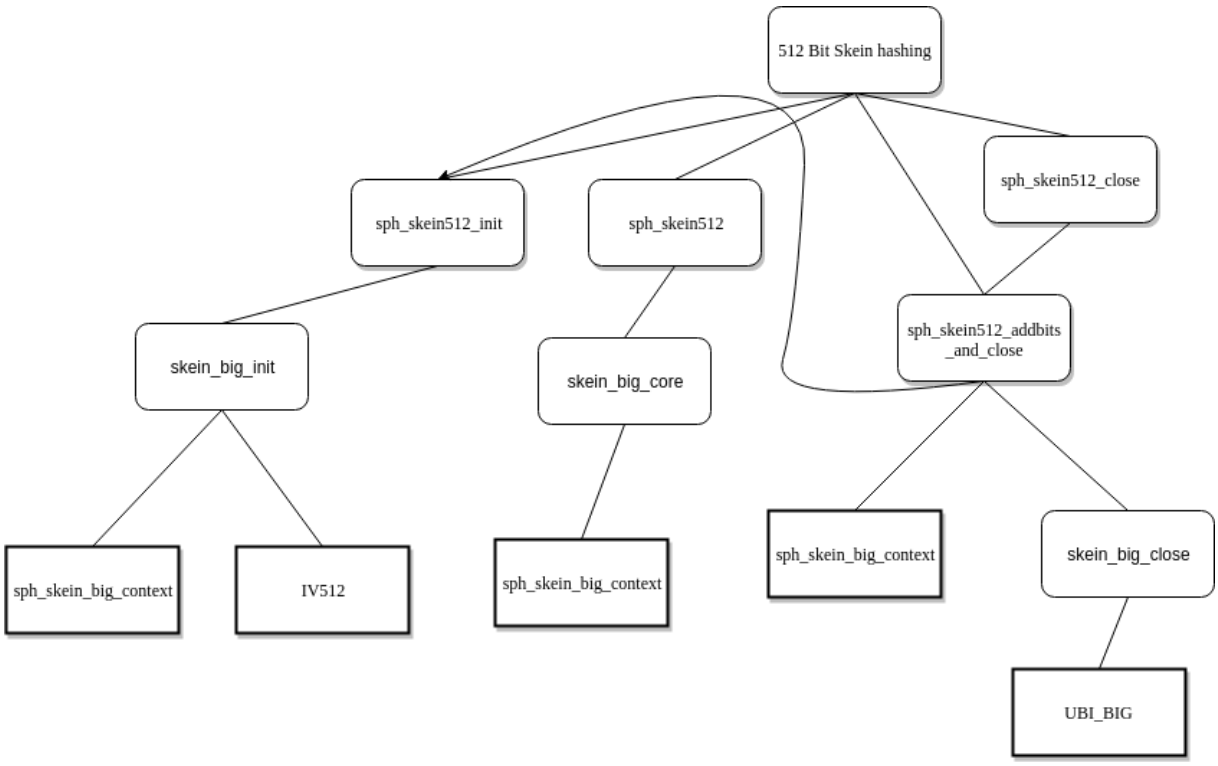
فصل ۳

مدل طلایی

۱.۳ مقدمه

برای انجام یک پروژه، موازی با پیاده‌سازی آن در زبانی مانند وریلاگ، تیم دیگری همان پروژه را در زبانی معمولا سطح بالاتر پیاده‌سازی می‌کنند. این برنامه‌ی پیاده‌سازی شده مدل طلایی نام دارد. از مدل طلایی برای صحت سنجی نتیجه‌ی به دست آمده‌ی مدل اصلی است. خروجی این دو برنامه یکسان است، اما نحوه‌ی پیاده‌سازی این دو لزوما مشابه نیست.

در مدل طلایی ۴ نوع متفاوت از Skein hash آورده شده‌است (۲۲۴ و ۲۵۶ و ۳۸۴ و ۵۱۲ بیت) که همانطور که در مدل طراحی شده با verilog نیز تنها نوع استاندارد (۵۱۲ بیت)آن پیاده‌سازی شده است ، در مدل طلایی نیز تنها توضیحات و مستندات این نوع ارائه خواهد شد.



۲.۳ پیاده‌سازی الگوریتم

در شکل بالا تمامی توابع و ساختارهای مورد نیاز و سلسله مراتب آن‌ها برای نوع ۵۱۲ بیتی الگوریتم آورده شده است ، برای توضیح نحوه‌ی اجرای الگوریتم با شروع از sph-skein-big-context سلسله اجرای برنامه توضیح داده خواهد شد.

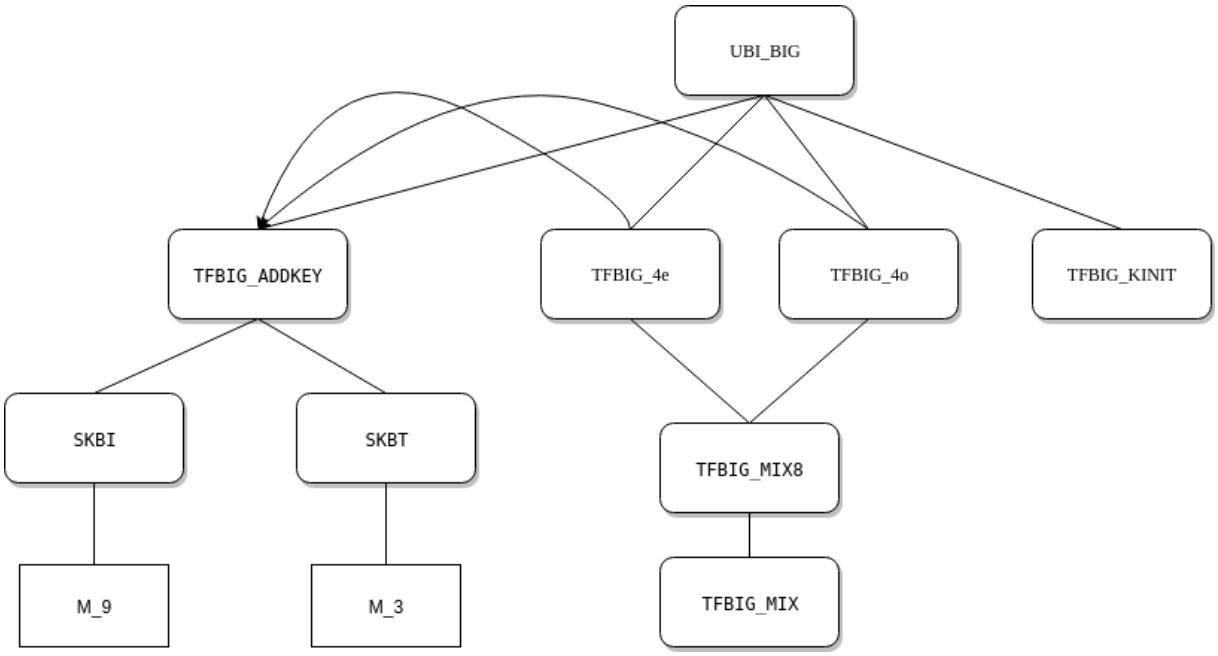
در این برنامه برای ذخیره و استفاده از هش ، از ساختاری به نام sph-skein-big-context استفاده شده است و هدف برنامه اجرای الگوریتم و بدست آوردن درهم‌سازی مورد نظر است.

برای اجرای الگوریتم هش ۵۱۲ بیتی ، در سلسله‌ی اجرا از توابع زیر استفاده شده است :

در ابتدا برنامه با ذخیره‌ی مقادیر از پیش تعیین شده IV512 در ساختار معرفی شده شروع به کار می‌کند ، و این کار توسط تابع sph-skein512-init انجام میگردد.

سپس با در نظر گرفتن ورودی و سایز این ورودی، اجرای الگوریتم هش توسط تابع sph-skein512 شروع می‌شود و این تابع شروع به درهم‌سازی داده‌ی ورودی می‌کند، به این صورت که ابتدا ۵۱۲ بیت ابتدایی را با استفاده از UBI-BIG هش می‌کند و در ادامه ۵۱۲ بیت بعدی را هش می‌کند تا بیت‌های نهایی، که مقادیر آن‌را در بافر ذخیره می‌کند . مسئولیت درهم‌سازی این بیت های نهایی بر عهده‌ی تابع sph-skein512-close است، که این تابع در صورت وجود بیت اضافه در ورودی، با اضافه کردن آن‌ها به دیتای ذخیره‌شده در بافر شروع به درهم‌سازی این بیت‌های نهایی می‌کند (که این کار را با کمک ماکروی UBI-BIG انجام می‌دهد). در انتها نیز مقادیر ساختار هش را به همان مقادیر اولیه تغییر می‌دهد.

حال برای فهم درست از توابع مورد استفاده لازم است نحوه‌ی پیاده‌سازی UBI-BIG توضیح داده‌شود. تمامی سلسله مراتب طراحی آن در شکل زیر آورده شده‌است.



کار این ماکرو درهم‌سازی بلوکی از دیتاست که به عنوان ورودی میگیرد، که این‌کار را با استفاده از نتیجه‌ی درهم‌سازی قبلی و ورودی جدید انجام می‌دهد.

۳.۳ ساختارها

ساختارها شامل struct ها، ماکروها، ثابت‌ها و جنس‌های تعریف شده است.

sph-skein-big-context ۱.۳.۳

این ساختار مورد نظر برای ذخیره و استفاده از هش است (شامل مقادیری از هش قبلی و مقادیر جدید محاسبه شده).
این ساختار شامل یک آرایه‌ی ۶۴ بیتی از کاراکترها و هشت عدد ۶۴ بیتی که برای ذخیره‌ی ۵۱۲ بیت هش استفاده می‌شوند و هم‌چنین شامل دو عدد با نام‌های ptr, bcount است که ptr به آخرین خانه‌ی پر شده‌ی بافر از دیتا و bcount تعداد مضارب صحیح ۵۱۲ کوچکتر از طول دیتا است.

IV512 ۲.۳.۳

این ساختار شامل مقادیر اولیه‌ی هش است. یک عدد ۵۱۲ بیتی را برای خوانا بودن در مبنای ۱۶ و در ۸ بلاک ۱۶ رقمی نگاه می‌دارد.

UBI-BIG ۳.۳.۳

در این تابع روی دیتای ذخیره شده در بافر، عملیات درهم‌سازی انجام شده‌است. این درهم‌سازی برمبنای مقادیر قبلی موجود در $h_۰$ تا $h_۷$ (که در سری قبلی صدا شدن این تابع مشخص شده‌اند)، دیتا و ورودی‌های extra و etype انجام شده‌است. در ابتدا سه sph-u64با اسامی $t_۰, t_۱, t_۲$ تعریف شده‌اند. sph-u64 جنسی تعریف شده برای متغیرهای ۶۴ بیتی بدون علامت است. با شروع از buf هر هشت عنصر که هر کدام یک بایت هستند توسط دیکودر در sph-dec64le-aligned به ۶۴ بیت پشت سر هم تبدیل شده و سپس به $m_۰$تا $m_۷$ داده شده‌اند. مقدارهای $m_۰$ تا $m_۷$ در $p_۰$ تا $p_۷$ ریخته شده‌اند. $m_۰$ تا $m_۷$ تا اخر تابع بدون تغییر باقی مانده و مقدارهای اولیه‌ی بخش‌های دیتا هستند. سپس مقدار extra پس از cast به sph-u64با مقدار bcount با ۶ بیت شیفت به چپ جمع شده و به $t_۰$ داده شده‌است. سپس مقدار bcount ، ۵۸ بیت به راست شیفت داده شده و etype هم ۵۵ بیت به چپ شیفت داده شده‌است. این مقادیر با هم جمع شده‌اند و جمعشان به $t_۱$ داده شده‌است. مقدار شیفت داده شدن‌ها از روی شکل زیر قابل توجیه‌اند:

Name	Bits	Description
Position	0– 95	The number of bytes in the string processed so far (including this block)
reserved	96–111	Reserved for future use, must be zero
TreeLevel	112–118	Level in the hash tree, zero for non-tree computations.
BitPad	119	Set if this block contains the last byte of an input whose length was not an integral number of bytes. 0 otherwise.
Type	120–125	Type of the field (config, message, output, etc.)
First	126	Set for the first block of a UBI compression.
Final	127	Set for the last block of a UBI compression.

در این جدول TreeLevel همان bcount است که در تابع skein-big-core در هربار صدا کردن UBI-BIG مقدار آن یک واحد افزوده می‌شود. etype برای رد کردن بخش position و هم‌چنین مشخص کردن بیت first است و extra برای تعیین بیت final و bitpad استفاده شده‌است. پنج بیت خالی هم برای Typeقرار داده شده و هم‌چنین بخش reserved هم صفر است.
سپس تابع TFBIG-KINIT صدا شده تا مقادیر $t_۲$ و $h_۸$ برمبنای بقیه ورودی های تابع یعنی $t_۰, t_۱$ و $h_۰$ تا $h_۷$ تعیین شوند. سپس برای اعداد زوج بین ۰ تا ۱۷TFBIG-4e و برای فردها TFBIG-4o صدا شده‌است. به این ترتیب میکس در ۱۸ سری چهار تایی اجرا شده که هر کدام ۴ round دارند و هر ۸ سری صدا شدن مشابه است و در هر یک از این ۱۸ سری دانستن زوج و فرد بودن سری کافیست. هم‌چنین هر چهار بار یعنی در ابتدای هر TFBIG-4eیاTFBIG-4o یک بار TFBIG-ADDKEYصدا شده تا بر مبنای شماره‌ی سری که در این‌جا با s نمایش داده شده و همین‌طور در pi و باقی مانده گرفتن از جمعشان کلید جدید مشخص

شده و با pi جمع شود. این‌جا از h. تا h_7 که حاصل سری قبلی اجرای UBI-BIG است و همین‌طور tweak ها استفاده شده تا مقدار جدید pi تعیین و در سری بعد استفاده شود. سپس برای بار هجدهم TFBIG-ADDKEY صدا شده‌است. در نهایت hi از xor گرفتن pi و mi به دست آمده‌است، پس hi نشان‌دهنده‌ی بیت‌های تغییر یافته‌ی pi در طول تابع است.

TFBIG-4o و TFBIG-4e ۴.۳.۳

این توابع برای تغییر مقادیر p. تا p_7 طراحی شده‌است. همان‌طور که پیش‌تر توضیح داده شد، ۷۲ بار تابع درهم‌سازی صدا شده، و هر ۸ سلسله از این ۷۲ مرحله یکسان است، هم‌چنین در انتهای هر ۴ مرحله مقدار کلید تغییر داده شده‌است.

این تابع یک ورودی s دارد. تابع TFBIG-ADDKEY با p. تا p_7 و h و t و s به ترتیب به عنوان w. تا w_7 و k و t و s صدا شده‌است. h و t برای concat و ساختن کلید در تابع TFBIG-ADDKEY استفاده شده‌اند.

سپس TFBIG-MIX8 چهار بار برای ترتیب‌های متفاوتی از p. تا p_7 با اعداد متفاوت به عنوان rc صدا شده‌است. ترتیب صدا شدن p. تا p_7 برای تعداد بلاک ۸ به صورت جدول‌های زیر است، که برای هر راند از ۰ تا ۳، بر حسب راند قبل ترتیب‌ها چهار عدد جابه‌جا شده‌اند، و تفاوت حالت‌های زوج و فرد، در اعداد استفاده شده است. در جدول‌ها N_w تعداد cipher block‌هاست که در این کد ۸ است. همین‌طور j همان شماره‌ی راند در ماژول‌های وریلاگ است.

N_w	4		8				16							
j	0	1	0	1	2	3	0	1	2	3	4	5	6	7
0	14	16	46	36	19	37	24	13	8	47	8	17	22	37
1	52	57	33	27	14	42	38	19	10	55	49	18	23	52
2	23	40	17	49	36	39	33	4	51	13	34	41	59	17
$d=$ 3	5	37	44	9	54	56	5	20	48	41	47	28	16	25
4	25	33	39	30	34	24	41	9	37	31	12	47	44	30
5	46	12	13	50	10	17	16	34	56	51	4	53	42	41
6	58	22	25	29	39	43	31	44	47	46	19	42	44	25
7	32	32	8	35	56	22	9	48	35	52	23	31	37	20

	$i=$															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	0	3	2	1												
$N_w=$ 8	2	1	4	7	6	5	0	3								
16	0	9	2	13	6	11	4	15	10	7	12	3	14	5	8	1

TFBIG-ADDKEY ۵.۳.۳

در این تابع طبق فرمول‌های زیر مقادیر ورودی تغییر داده شده‌است و برای اینکار از تابع‌های SKBI و SKBT استفاده شده‌است. متغیرهای h. تا h_8 در تولید کلید استفاده می‌شوند، برای محاسبه‌ی اندیس آن‌ها (از ۰ تا ۸) از SKBI استفاده شده است. متغیرهای دیگری که در تولید کلید استفاده شده‌اند t. تا t_2 هستند که برای تولید اندیس آن‌ها از تابع SKBT استفاده شده است.

$$
\begin{aligned}
k_{s,i} &= k_{(s+i) \mod 9} && i = 0, 1, 2, ..., 4 \\
k_{s,5} &= k_{(s+5) \mod 9} + t_{s \mod 3} \\
k_{s,6} &= k_{(s+6) \mod 9} + t_{(s+1) \mod 3} \\
k_{s,7} &= k_{(s+7) \mod 9} + s
\end{aligned}
$$

دقت شود که تمامی این محاسبات برای نوع ۵۱۲ بیتی الگوریتم است.

SKBI ۶.۳.۳

تابع SKBI برای محاسبه‌ی اندیس کلید استفاده شده‌است.

در الگوریتم برای تولید k. تا k_8 از این ماکرو استفاده شده است. k و s و i به این ماکرو داده شده و سپس k به M9-s-i متصل می‌شود. M9-s-i باقی‌مانده‌ی $s + i$ بر ۹ تعریف شده‌است.

SKBT ۷.۳.۳

برای تولید $t_.$ تا $t_۲$ از این ماکرو استفاده شده است، t و s و i به این ماکرو داده شده و سپس t به M3-s-i متصل شده است. M3-s-i باقی‌مانده‌ی $s + i$ بر ۳ تعریف شده است.

TFBIG-MIX8 ۸.۳.۳

همان‌طور که در مقدمه گفته شده‌است، هر سری از هشت سری، چهار round دارد، پس طراحی این تابع برای ساده‌سازی استفاده‌ی متداول از TFBIG-MIX است. به صورت متداول در کد به چهار سری استفاده از TFBIG-MIX به‌صورت پشت سر هم نیاز است.

TFBIG-MIX ۹.۳.۳

وظیفه‌ی این تابع، درهم سازی بلاک‌های ورودی طبق فرمول‌های زیر است.

$$y_. = (x_. + x_۱) \mod ۲^{۶۴}$$
$$y_۱ = (x_۱ <<< R_{(d \mod ۸),j}) \oplus y_.$$

که مقادیر R در جدول زیر آمده است :

N_w		4		8				16							
j		0	1	0	1	2	3	0	1	2	3	4	5	6	7
	0	14	16	46	36	19	37	24	13	8	47	8	17	22	37
	1	52	57	33	27	14	42	38	19	10	55	49	18	23	52
	2	23	40	17	49	36	39	33	4	51	13	34	41	59	17
$d =$	3	5	37	44	9	54	56	5	20	48	41	47	28	16	25
	4	25	33	39	30	34	24	41	9	37	31	12	47	44	30
	5	46	12	13	50	10	17	16	34	56	51	4	53	42	41
	6	58	22	25	29	39	43	31	44	47	46	19	42	44	25
	7	32	32	8	35	56	22	9	48	35	52	23	31	37	20

TFBIG-KINIT ۱۰.۳.۳

در این تابع با ورودی‌های $t_.$ تا $t_۲$ و $h_.$ تا $h_۸$ مقادیر زیر محاسبه شده‌است:

$$k_۸ = C \oplus k_. \oplus k_۱ \oplus ... \oplus k_۷$$

$$t_۲ = t_۱ \oplus t_.$$

که مقدار ثابت C به آن جهت در فرمول وجود دارد که از ۰ نبودن تمامی بیت‌ها اطمینان حاصل شود.

DECL-STATE-BIG ۱۱.۳.۳

در این ماکرو متغیرهای $h_.$ تا $h_۷$ و bcount از جنس sph-u64 (متغیر ۶۴بیتی بدون علامت) تعریف شده‌اند.

READ-STATE-BIG ۱۲.۳.۳

این تابع برای خواندن اطلاعات از ورودی و ذخیره‌ی آن‌ها بر روی متغیرهاست. به طور دقیق‌تر به این تابع بلاک sc به عنوان ورودی داده شده‌است. در آن، متغیرهای $h_.$ تا $h_۷$ و bcount استراکت به ترتیب به متغیرهای $h_.$ تا $h_۷$ و bcount کد مقداردهی شده‌اند.

WRITE-STATE-BIG ۱۳.۳.۳

این تابع برای ذخیره‌ی اطلاعات بر روی struct است.

به این تابع ورودی struct با نام sc داده شده‌است. متغیرهای h_0 تا h_7 و همین‌طور bcount کد، در h_0 تا h_7 و bcount ساختار ذخیره شده‌اند.

توابع ۴.۳

sph-skein512-init ۱.۴.۳

این تابع مسئولیت مقداردهی اولیه‌ی ساختار هش را بر عهده دارد، که برای آن تابع skein-big-init را با ورودی اولیه‌ی IV512 اجرا می‌کند.

skein-big-init ۲.۴.۳

این تابع دو ورودی می‌پذیرد که یکی از آن‌ها آدرس یک ساختار هش است و دیگری مقدار اولیه است. در این تابع مقادیر متناظر ساختار داده شده برابر مقادیر اولیه قرار گرفته‌اند. که مقادیر اولیه در حالت ۵۱۲ بیتی در ساختار IV512 ذخیره شده است.

sph-skein512 ۳.۴.۳

در این تابع، ماکروی skein-big-core صدا شده‌است. ورودی‌های این تابع که بدون انجام هیچ پردازشی به skein-big-core پاس داده شده‌اند، cc که همان ساختار هش است، data که داده‌ی ورودی است و len که طول data است، هستند.

skein-big-core ۴.۴.۳

در این تابع همه‌ی بلاک‌های دیتا به‌جز بلاک اخر در دسته‌های ۵۱۲ تایی هش شده‌اند. مقدار bcount هم برای استفاده‌ی ثانویه تعیین شده و هم‌چنین آخرین بلاک دیتا در بافر ریخته شده‌است.

به این تابع ورودی‌های sc که ساختار هش است، data که داده‌ی ورودی برای هش است و len که طول data است پاس داده شده‌اند. در ابتدای تابع با صدا شدن DECL-STATE-BIG متغیرهای لازم تعریف شده‌اند. سپس در یک if بررسی شده‌است که برای دیتا در بافر فضای کافی هست یا خیر:

- در صورتی که فضا باشد، کل دیتا از جایی که پوینتر به آن اشاره کرده‌است دخیره شده و سپس پوینتر که به پایان دیتای ذخیره شده اشاره دارد، به اندازه‌ی طول دیتا به جلو جابه‌جا شده‌است. در خود struct هم مقدار آن update شده و سپس از تابع خارج شده‌است.
- در غیر این صورت، ابتدا READ-STATE-BIG صدا شده‌است. متغیر first یک متغیر هشت بیتی با مقدار ۰ یا ۱۲۸ است. اگر این تابع به طور متدوال از skein-hash صدا شود first برابر با ۱۲۸ می‌شود. سپس در یک لوپ ابتدا در صورت پر بودن بافر از دیتا (به این معنی که پوینتر برابر با سایز بافر شده‌باشد)، اول bcount یکی زیاد شده، سپس UBI-BIG با ورودی‌های $first + ۹۶$ به عنوان etype و ۰ به عنوان extra صدا شده‌است. پس از آن first و ptr هر دو صفر گذاشته شده‌اند تا برای سری بعد پر شدن دیتا، بافر از ابتدا overwrite شود. سپس شرط پر بودن بافر تمام شده و به اندازه‌ی مینیموم مقداری که در بافر جا هست با طول دیتای باقی‌مانده، دیتا در بافر ذخیره شده‌است. پوینتر و دیتا با مقدار این مینیموم جمع و len منهای آن شده تا مقدار دیتای باقی‌مانده را نشان دهد. سپس حلقه تا زمانی که هنوز دیتایی باقی مانده‌باشد تکرار می‌شود. درواقع در این حلقه هر سری به تعداد بزرگترین مضرب سایز بافر که اکیدا کوچک‌تر از len دیتا است تکرار شده و هر سری روی همان طول از دیتا UBI-BIG صدا شده‌است. bcount همین تعداد بار را نشان می‌دهد. آخرین بلاک دیتا که کوچک‌تر مساوی سایز بافر است، در بافر ذخیره شده و ptr به آخر آن اشاره کرده‌است. در آخر WRITE-STATE-BIG صدا و بعد مقدار فعلی پوینتر هم در struct ذخیره شده‌است.

نحوه ذخیره‌سازی tweak به شکل زیر است.

Name	Bits	Description
Position	0– 95	The number of bytes in the string processed so far (including this block)
reserved	96–111	Reserved for future use, must be zero
TreeLevel	112–118	Level in the hash tree, zero for non-tree computations.
BitPad	119	Set if this block contains the last byte of an input whose length was not an integral number of bytes. 0 otherwise.
Type	120–125	Type of the field (config, message, output, etc.)
First	126	Set for the first block of a UBI compression.
Final	127	Set for the last block of a UBI compression.

دلیل جمع کردن first با ۹۶، رد کردن بخش position است. حالت اولیه first هم به این علت با چک کردن bcount مقداردهی شده که مقدار بخش first باید برای سری اول گرفتن بلوک دیتا برابر با ۱ باشد.

در این تابع ممکن است سایز دیتا دقیقا مضربی از سایز بلاک (۵۱۲) باشد، برای این حالت باید مقدار بیت final یک شود، اما این تابع از این که در حال پردازش اخرین بخش دیتا هست یا نه باخبر نیست و درنتیجه در آخر ممکن است بافر شامل یک بلاک کامل از دیتا باشد.

skein-hash ۵.۴.۳

در این تابع ابتدا هش از جنس آرایه‌ای ۶۴ تایی از کاراکترهای بدون علامت (uint8-t) و سپس متغیری با نام ctx، ساختاری از جنس sph-skein-big-context تعریف شده‌است. این هش ۵۱۲ در ۵۱۲ است و به همین دلیل حاصل نهایی هم ۶۴ بایت درنظر گرفته شده است. سپس آدرس ctx به تابع sph-skein512-init پاس داده شده‌است. در این تابع مقدارهای اولیه در struct ذخیره شده‌اند. سپس تابع sph-skein512 صدا شده که در آن تمام بلاک‌های دیتا به جز بلاک آخر هش شده و بلاک اخر هم در بافر ذخیره شده‌است. سپس تابع sph-skein512-close صدا شده و به آن struct و ادرس شروع hash داده شده‌اند. در این مرحله بیت‌های اضافی اضافه شده‌اند. بلاک آخر هش شده و در dst ذخیره شده‌است. در نهایت ۳۲ بایت از hash در output ریخته شده‌است.

sph-skein512-close ۶.۴.۳

در این تابع sph-skein-addbits-and-close صدا شده و به آن ساختار cc ، ادرس dst و همین طور صفر به عنوان ub که بیت‌های اضافه است و n که تعداد این بیت‌های اضافه است داده شده‌اند.

sph-skein-addbits-and-close ۷.۴.۳

در این تابع skein-big-close با مقدار ۶۴ برای out-len و sph-skein512-init صدا شده‌اند. کار درهم‌سازی در skein-big-close تمام و سپس دوباره h. تا $h_۷$ از مقدارهای اولیه پر شده‌اند.

skein-big-close ۸.۴.۳

به این تابع ورودی‌های sc به عنوان ساختار هش، ub به عنوان بیت‌های اضافه، n به عنوان تعداد بیت‌های اضافه، dst برای ذخیره هش نهایی و out-len به عنوان طول دیتا پاس داده شده‌اند. با توجه به هشت بیتی بودن بلوک‌ها در حداکثر مقدار n هشت است. در نتیجه در صورت غیر صفر بودن n ،با شیفت دادن ۱۲۸ به اندازه‌ی n، متغیری به نام z با n بیت صفر در سمت راست و سپس یک بیت ۱ ساخته شده‌است. سپس با and کردن ub با z-، n بیت سمت راست ub صفر شده و این مقدار به x داده شده است. (زیرا

z به صورت مکمل دو منفی شده‌است.) سپس بیت n+1م ub با or گرفته‌شدن با z ، ۱ شده‌است. سپس skein-big-core برای طول ۱ صدا شده‌است. (زیرا طول x یک بایت است.) شرط بررسی n اینجا به پایان رسیده‌است. سپس read-state-big صدا و بعد از آن باقی‌مانده‌ی فضای بافر با ۰ پر شده‌است. سپس دو دفعه UBI-BIG صدا شده‌است. در دفعه‌ی اول صدا شدن تابع، etype برابر جمع (۲۵۶ + ۹۶) با ۱۲۸ ،در صورت ۰ بودن bcount است و ۹۶ + ۱۲۸ در صورت غیر صفر بودن آن. همین‌طور در صورت غیر صفر بودن n عدد یک هم با etype جمع شده‌است. extra هم برابر با مقدار ptr گذاشته شده‌است، که در زمان صدا کردن تابع به اخرین جایی که دیتا در بافر هست و بیت‌های بعدی آن با ۰ پر شده‌اند، اشاره دارد. جمع شدن ۲۵۶ با etype برای یک گذاشتن BitPad و جمع شدن ۹۶ برای اسکیپ کردن بخش position است. پیش از صدا شدن تابع برای بار دوم کل بافر با ۰ پر و bcount هم برابر با ۰ گذاشته شده‌است. بار دوم UBI-BIG برای etype با مقدار ۵۱۲ و extra با مقدار ۸ صدا شده‌است. در ۵۱۰ فقط بیت دوم از راست یک است، که بیت final است و ۸ بایت بافر با ۰ پر شده‌است. درنتیجه extra هم ۸ است. در نهایت h_0 تا h_7 توسط sph-enc64le-aligned بایت بایت شده و در بافر ذخیره شده‌است، و سپس بافر به تعداد بایت out-len (در این‌جا ۶۴ بایت) در dst ذخیره شده‌است.

۵.۳ نحوه‌ی استفاده از مدل طلایی

برای اجرای کد مدل طلایی، تابع skeinhash صدا شده‌است. ورودی این تابع، یک متغیر از جنس constant void* است و آدرس ذخیره‌ی خروجی توابع هش هم همراه با ورودی به تابع پاس داده شده‌است. برای این کار یک اسکریپت main.c و یک اسکریپت skeinhash.h به کد اضافه شده‌است.

اسکریپت skeinhash.h به این دلیل اضافه شده‌است که بتوان از تابع skeinhash در اسکریپت main.c استفاده کرد.

در اسکریپت main.c متغیر ورودی، یعنی دیتای تهیه شده برای تست عملیات هش با نام inputو آدرس ذخیره‌ی output تعریف شده‌اند. در تابع skeinhash متغیری که نشان‌دهنده‌ی طول input است، ۸۰ بایت در نظر گرفته شده، با این حال کد مدل طلایی برای هر طولی پاسخ‌گو است. متغیر ورودی در main.c برای طول ۸۰ از جنس رشته (آرایه‌ای ۸۰تایی از char) تعریف شده، تا مقداردهی آن بایت به بایت انجام شود. خروجی خود تابع‌های داخلی هش، ۶۴ بایت یا ۵۱۲ بیت است، اما در نهایت ۳۲ بایت اول این خروجی در output ریخته شده‌اند، در نتیجه در main.c این متغیر ۳۲ بایتی تعریف شده‌است. در نهایت این خروجی در مبنای ۱۶ چاپ شده‌است.

فصل ۴

نتیجه‌گیری

در بخش اول مقاله مقدمه‌ای در رابطه با Skein hashing تهیه شده‌است. در این بخش توضیح کاملی درباره‌ی الگوریتم، توابع داخلی و فرمول‌های استفاده شده در این هش داده شده‌است.

در ادامه، بخش دو و سه‌ی مقاله در رابطه با مستندسازی کدها آورده شده‌اند. در این مستندسازی‌ها علاوه بر توضیح دقیق کد، به توضیح علت طراحی هر قطعه کد با توجه به استفاده‌ی آن در الگوریتم Skein hashing نیز پرداخته شده‌است.

در بخش دوم مقاله، به مستندسازی بخش پیاده‌سازی سخت‌افزاری با وریلاگ پرداخته شده‌است. در زیربخش اول این قسمت، اشتباهات منطقی یا سینتکسی موجود در کد و راه حل ارائه شده برای هر یک از آن‌ها و سپس مستندسازی قسمت به قسمت ماژول‌های وریلاگ آورده شده‌است. سپس در زیربخش جمع‌بندی، نمودار درختی این ماژول‌ها برای توصیف ساختار کلی طراحی سخت‌افزاری طراحی شده‌است. در انتها زیربخش‌های شبیه‌سازی و توضیحات مربوط به تست‌بنچ قرار گرفته‌اند. در این دو قسمت اطلاعات مربوط به شبیه‌سازی نظیر کلاک‌ها، تصویر موج‌ها و توصیفی از ساختار تست‌بنچ آورده شده‌اند.

بخش سه مربوط به مستندسازی مدل طلایی است. در این بخش ابتدا توضیح مختصری در رابطه با دلیل استفاده از مدل طلایی و نموداری درختی از ساختار کلی مدل طلایی، سپس ساختارها و توابع و پس از آن روند اجرا و استفاده از کد مدل طلایی توصیف شده‌اند.

هم‌چنین بخشی هم مربوط به سنتز و اطلاعات مربوط به آن نوشته شده‌است.

در این پروژه لزوم استفاده از مدل طلایی برای مقایسه و اطمینان حاصل کردن از صحت کد سطح پایین‌تر در کنار طراحی تست، نحوه‌ی مستندسازی و لزوم استفاده از آن برای قابل فهم کردن و درنتیجه ارتقا دادن کد، نحوه‌ی شبیه‌سازی و سنتز و اصلاح یک برنامه در مقیاسی مشابه با صنعت آموخته‌شد.

همچنین منبع استفاده شده در این پروژه را اینجا ببینید.

فصل ۵

منابع

The Skein Hash Function Family (version 1.3 , Oct 2010)