LSH 소스코드 사용 매뉴얼

ver 1.0

2016년 6월

NSR 국가보안기술연구소

1. 해시함수 LSH

LSH(Lightweight Secure Hash)는 다음의 입출력 규격을 가지는 해시함수다.

- 워드 길이: 32 또는 64 비트
- 출력 길이: 1 ~ 8w 길이의 정수 (비트)

LSH는 w 비트 워드 단위로 동작하며, n 비트 출력 값을 가지는 해시함수 LSH-8w-n 으로 구성된 해시함수 군이다. 여기에서 w는 32 또는 64이고, n은 1과 8w 사이의 정수이다. 규격서에 따르면 LSH-256-224, LSH-256-256, LSH-512-224, LSH-512-256, LSH-512-384, LSH-512-512의 6개 규격을 제시하고 있다.

2. HMAC

HMAC(Hash-based Message Authentication Code)는 인증 값(tag)을 생성 또는 검증하는 기능을 제공한다. HMAC에서 인증 값 생성 과정은 키와 메시지를 입력으로, 인증 값을 출력으로 가진다. HMAC 입력 메시지는 임의의 길이를 가진다(길이 0도 가능함).

3. 소스코드 구성

LSH 소스코드는 다음과 같이 구성되어 있다.

- LSH의 SIMD 고속 구현 코드
- LSH의 C 레퍼런스 코드
- LSH의 Java 소스코드
- LSH의 Python 소스코드

각 소스코드 별 적용 환경은 다음 표와 같다.

<table>
<tr><th></th><th>OS</th><th>CPU</th><th>SIMD 가속</th><th>비고</th></tr>
<tr><td rowspan="4">SIMD 고속 구현 코드</td><td>Windows</td><td>Intel/AMD</td><td>적용</td><td>Visual Studio, MinGW 및 cygwin 환경에서 컴파일 가능</td></tr>
<tr><td rowspan="3">Linux 등 POSIX 계열</td><td>Intel/AMD</td><td>적용</td><td rowspan="3">크로스 컴파일 지원</td></tr>
<tr><td>ARM(NEON 지원)</td><td>적용</td></tr>
<tr><td>기타</td><td>미적용</td></tr>
<tr><td>C 레퍼런스 코드</td><td>OS 의존성 없음</td><td>CPU 의존성 없음</td><td>미적용</td><td>C 컴파일러를 제공하는 모든 환경에서 사용 가능</td></tr>
<tr><td>Java 코드</td><td colspan="2">Java 1.5 이상</td><td>미적용</td><td></td></tr>
<tr><td>Python 코드</td><td colspan="2">Python 2.7 이상 또는 3.2 이상</td><td>미적용</td><td></td></tr>
</table>

SIMD 고속 구현 코드는 다음과 같은 명령어 셋을 지원하는 CPU에서 동작한다.

- x86_x64: SSE2, SSSE3, AVX2, XOP
- ARM: NEON

CPU와 컴파일러에 따라서 적용되는 SIMD 가속이 다르며, 이와 관련한 세부사항은 5장과 6장에서 다룬다.

LSH 계산을 위해 init, update, final, digest의 4개의 인터페이스를 제공하며, 이는 C, Java, Python 코드 모두에 공통적으로 적용된다.

HMAC 계산을 위해 init, update, final, digest의 4개의 인터페이스를 제공하며, 이는 C, Java, Python 코드 모두에 공통적으로 적용된다.

4. LSH C 소스코드

4.1. 서비스

LSH의 C 소스코드는 다음의 서비스를 제공한다.

- 해시함수 LSH-256, LSH-512
- LSH 기반의 메시지 인증 코드(HMAC)

4.2. C 레퍼런스 코드의 구성

LSH의 C 레퍼런스 소스코드는 다음과 같이 구성되어 있다.

파일명	내용
LICENSE	오픈소스 라이선스 파일 (LSH 소스코드는 MIT 라이선스를 따름)[1]
include/lsh.h	LSH 함수 정의
include/lsh_def.h	LSH를 위한 기본 데이터 형 정의
src/hmac.h	HMAC 함수 정의
src/lsh_local.h	기본 매크로 정의
src/lsh.c	LSH 래퍼 함수 (wrapper function) 모음
src/lsh256.h	LSH256 구현
src/lsh256.c	
src/lsh512.h	LSH512 구현
src/lsh512.c	
src/hmac.c	HMAC 구현
test/	테스트벡터 코드

4.3. SIMD 고속 구현 코드의 구성

LSH의 SIMD 고속 구현 코드는 다음과 같이 구성되어 있다. x86, x86_64 아키텍처의 코드는 x86_x64 폴더, ARM 아키텍처의 코드는 arm 폴더에 있으며, 파일 구성에 일부 차이가 존재한다.

파일명	내용
Makefile	GNU Makefile
LICENSE	오픈소스 라이선스 파일 (LSH 소스코드는 MIT 라이선스를 따름)[1]
include/lsh.h	LSH 함수 정의
include/lsh_def.h	LSH를 위한 기본 데이터 형 정의
include/hmac.h	HMAC 함수 정의
src/lsh_local.h	기본 매크로 정의
src/lsh.c	LSH 래퍼 함수 (wrapper function) 모음

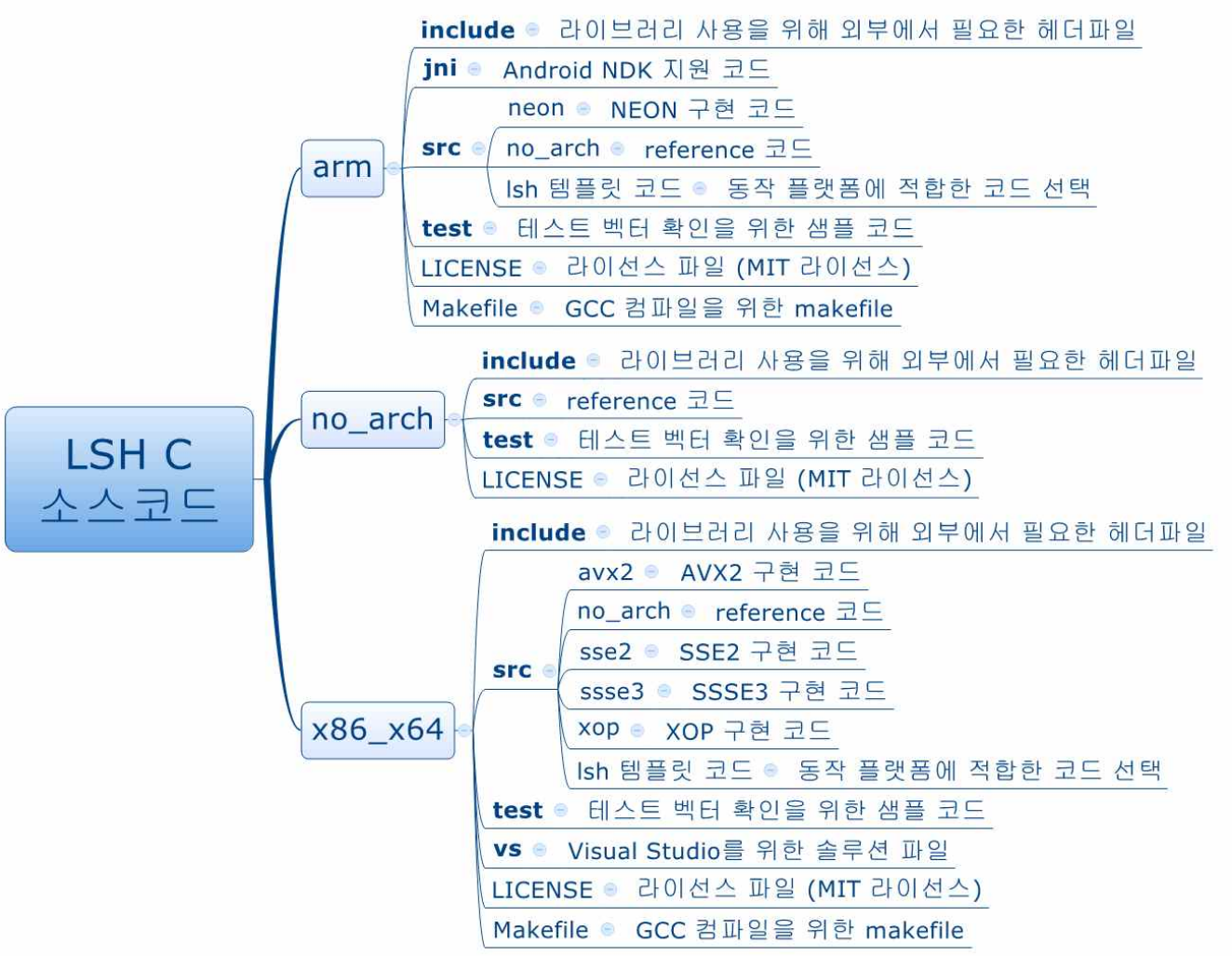
파일명	내용
src/hmac.c	HMAC 구현
src/no_arch/lsh256.h src/no_arch/lsh256.c	레퍼런스 LSH256 구현
src/no_arch/lsh512.h src/no_arch/lsh512.c	레퍼런스 LSH512 구현
src/sse2/lsh256.h src/sse2/lsh256.c	SSE2 LSH256 고속 구현
src/sse2/lsh512.h src/sse2/lsh512.c	SSE2 LSH512 고속 구현
src/ssse3/lsh256.h src/ssse3/lsh256.c	SSSE3 LSH256 고속 구현
src/ssse3/lsh512.h src/ssse3/lsh512.c	SSSE3 LSH512 고속 구현
src/xop/lsh256.h src/xop/lsh256.c	XOP LSH256 고속 구현
src/xop/lsh512.h src/xop/lsh512.c	XOP LSH512 고속 구현
src/avx2/lsh256.h src/avx2/lsh256.c	AVX2 LSH256 고속 구현
src/avx2/lsh512.h src/avx2/lsh512.c	AVX2 LSH512 고속 구현
src/neon/lsh256.h src/neon/lsh256.c	NEON LSH256 고속 구현
src/neon/lsh512.h src/neon/lsh512.c	NEON LSH512 고속 구현
jni/	Java Native Interface 구현(arm 전용)
test/	테스트벡터, 속도테스트 코드
vs/	Visual Studio 용 솔루션 파일(x86_x64 전용)

1. 오픈소스 라이선스 관련 정보는 한국저작권위원회 홈페이지를 참고하면 된다. (https://www.olis.or.kr)

4.4. LSH C 소스코드의 구성

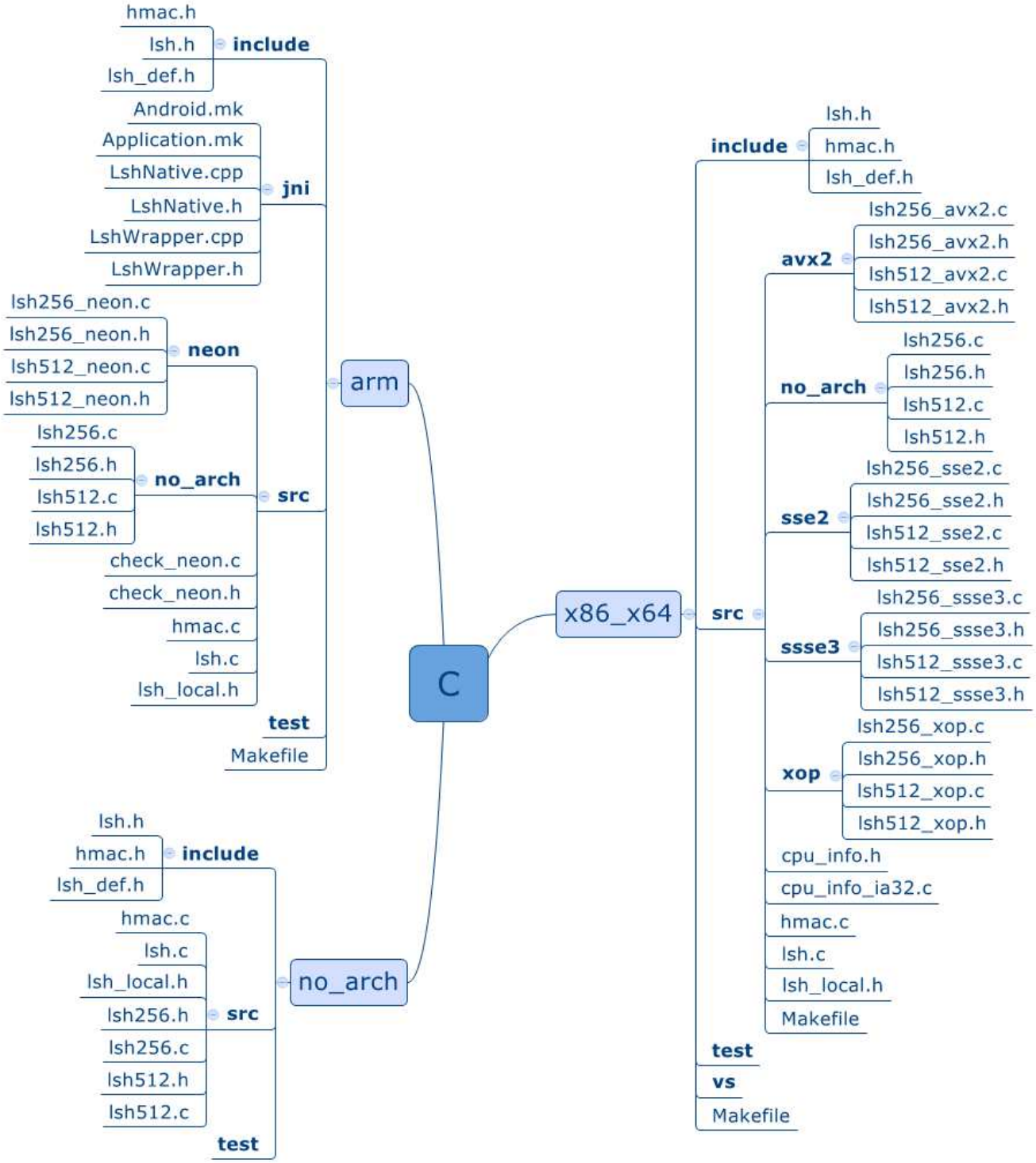
4.4.1. 소스코드 폴더 구조

LSH C 소스코드는 [그림 1]과 같은 폴더 구조로 구성되어 있다. arm 폴더는 ARM 머신에서 동작하는 레퍼런스 코드 및 neon을 적용한 고속 구현코드를 포함하며, gcc 빌드 할 수 있도록 Makefile을 제공한다. no_arch 폴더는 다양한 환경에 적용할 수 있도록 레퍼런스 C 코드만을 포함하고 있으며, 빌드는 각 환경에 맞게 개발자가 구성할 수 있도록 별도의 Makefile을 제공하지 않는다. x86_x64 폴더는 Intel, AMD의 x86 (amd64) 아키텍처의 머신에서 동작하는 레퍼런스 코드 및 SIMD를 적용한 고속구현 코드를 포함하며, gcc 빌드를 위한 Makefile 및 Visual Studio 빌드를 위한 솔루션 파일을 제공한다.



[그림 1] LSH C 소스코드 폴더 설명

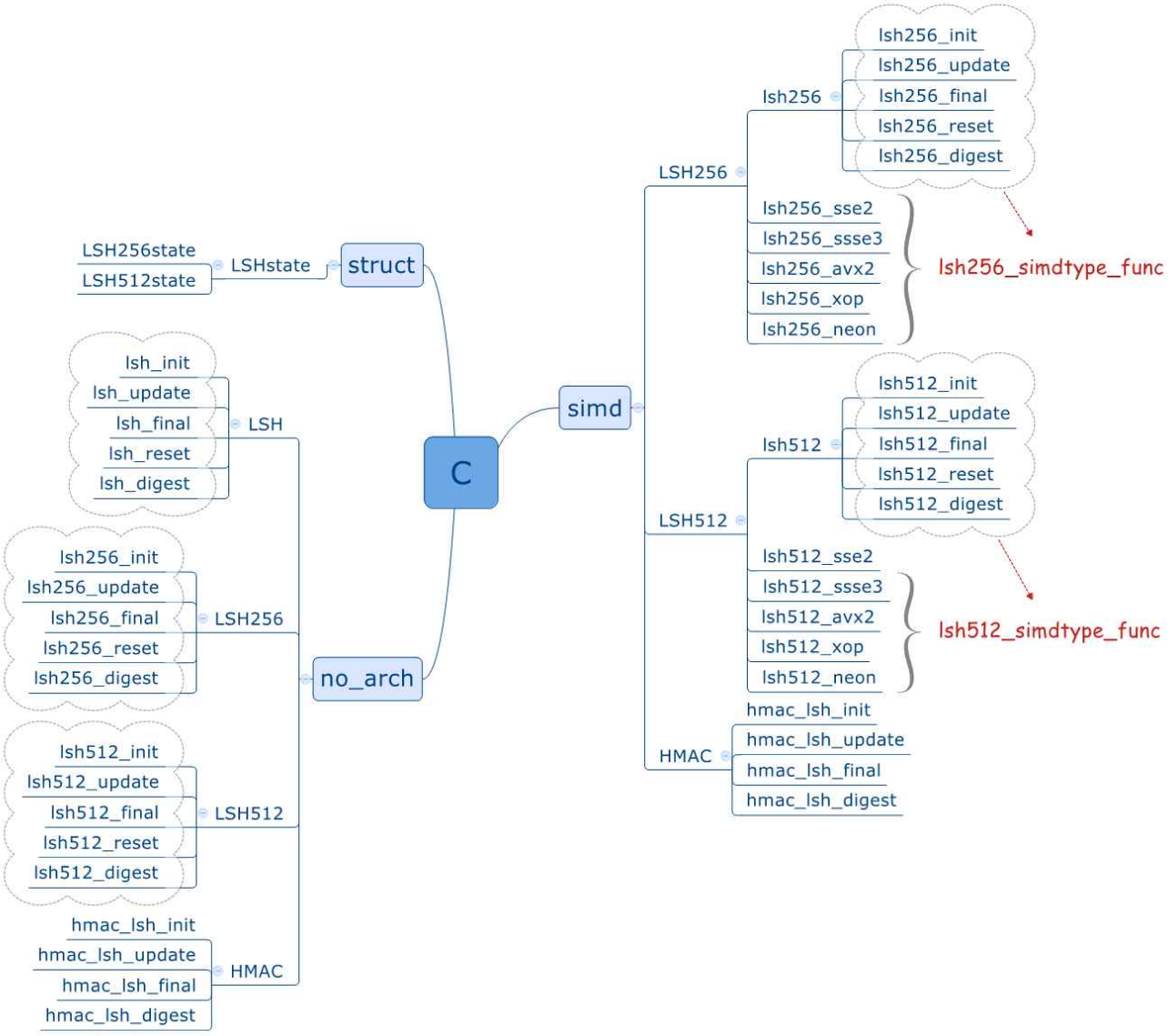
LSH의 C 소스코드는 [그림 2]와 같이 공통으로 사용하는 헤더파일은 include 폴더에, 공통으로 사용되는 소스코드는 src 아래, 각 SIMD 구현별 소스코드는 src 폴더 밑의 SIMD 이름 폴더에 존재하며, 개발자의 환경에 따라 필요한 소스코드만 발췌하여 사용하기 편리한 구조로 되어있다.



[그림 2] LSH C 소스코드 폴더 구조

4.4.2. LSH C 인터페이스 설계 원리

LSH C 소스코드는 LSHstate 구조체에 해시 중간 계산 값을 저장하여 사용하며, 해시 계산은 init, update, final의 단계를 순서대로 수행한다. 해시 계산의 세 단계를 한 번에 수행하기 위한 함수로 digest 함수를 제공하며, LSHstate 구조체를 재사용하기 위해 초기화 하는 reset 함수를 제공한다. init, update, final, digest, reset이 LSH를 구성하는 함수의 기본 집합이다. LSH256 계산과 LSH512 계산을 위한 함수 집합이 별도로 존재하며, SIMD 고속코드별로 각각 LSH256을 위한 함수집합, LSH512를 위한 함수집합이 별도로 구현되어 있다. HMAC 계산도 init, update, final 의 구조를 따르며, 한 번에 수행하기 위한 digest 함수를 제공한다.



[그림 3] LSH C 인터페이스

4.5. LSH C 소스코드의 인터페이스

본 절에서는 LSH C 소스코드가 제공하는 인터페이스를 다룬다.
해시 계산 및 HMAC계산에 있어 init, update, final의 구조를 따르고 있으며, 각 함수들의 입력으로 내부 상태를 저장하기 위한 구조체를 사용한다. LSH_MAKE_TYPE 매크로를 이용하여 계산하고자 하는 LSH의 파라미터를 조정할 수 있다.

4.5.1. LSH_MAKE_TYPE 매크로

LSH 알고리즘 명세 값을 생성한다.

문법

```
#define LSH_MAKE_TYPE(is_lsh512, hash_bit_len)
```

매개 변수

is_lsh512

LSH256을 사용하려면 0, LSH512를 사용하려면 양수를 입력한다.

hash_bit_len

출력 해시의 길이를 비트 단위로 입력한다. LSH256인 경우 256이 최댓값이며, LSH512인 경우 512가 최댓값이다.

설명

lsh_type 타입의 변수가 생성되며, 이 값은 lsh 인터페이스 전반에 사용된다. 이외에도 사전 설정된 lsh_type 값으로 LSH_TYPE_224, LSH_TYPE_256, LSH_TYPE_384, LSH_TYPE_512를 사용할 수 있다.

4.5.2. lsh_init

해시 계산을 위해 내부 상태 구조체를 초기화한다.

문법

```
lsh_err lsh_init(
        union LSH_Context *ctx,
        const lsh_type algtype
);
```

매개 변수

ctx [in/out]

LSH 해시 계산을 위해 필요한 내부 상태를 저장할 구조체

algtype [in]

LSH_MAKE_TYPE 매크로에 의해 만들어진 LSH 알고리즘 명세 값

반환 값

정상 수행한 경우 LSH_SUCCESS(=0)를, 에러가 발생한 경우 에러 코드를 반환한다.

설명

algtype에 명시된 값에 따라 LSH 내부 상태를 저장할 구조체를 초기화한다. LSH-256-224, LSH-256-256, LSH-512-224, LSH-512-256, LSH-512-384, LSH-512-512는 사전 계산된 초기상태 값을 이용하며, 이외의 algtype이 지정된 경우에는 초기상태 값을 새로 계산한다.

4.5.3. lsh_update

메시지를 입력받아 해시 내부 상태를 업데이트한다.

문법

```
lsh_err lsh_update(
        union LSH_Context *ctx,
        const lsh_u8 *data,
        size_t databitlen
);
```

매개 변수

ctx [in/out]

LSH 해시 계산을 위해 필요한 내부 상태를 저장할 구조체

data [in]

해시 값을 계산할 메시지

databitlen [in]

메시지의 길이 (비트단위)

반환 값

정상 수행한 경우 LSH_SUCCESS(=0)를, 에러가 발생한 경우 에러 코드를 반환한다.

설명

ctx 구조체의 버퍼에 남아 있는 메시지와 입력받은 메시지를 연접하여 블록단위로 내부 상태를 갱신하고, 남는 메시지는 ctx 구조체의 버퍼에 저장한다.

주의사항

databitlen은 lsh_update를 마지막으로 호출하는 경우 외에는 항상 8의 배수여야 한다.

4.5.4. lsh_final

최종 해시 값을 계산한다.

문법

```
lsh_err lsh_final(
        union LSH_Context *ctx,
        lsh_u8 *hashval
);
```

매개 변수

ctx [in/out]

LSH 해시 계산을 위해 필요한 내부 상태를 저장할 구조체

hashval [out]

해시 값을 저장할 버퍼

반환 값

정상 수행한 경우 LSH_SUCCESS(=0)를, 에러가 발생한 경우 에러 코드를 반환한다.

설명

ctx의 최종 내부 상태를 갱신하여 해시 값을 계산한다. ctx 구조체의 algtype 값의 출력 비트 길이가 8의 배수가 아닌 경우, 해시 값의 마지막 바이트는 상위 비트부터 채워진다.

4.5.5. lsh_digest

init, update, final 단계를 한 번에 수행하여 해시 값을 계산한다.

문법

```
lsh_err lsh_digest(
        const lsh_type algtype,
        const lsh_u8 *data,
        size_t databitlen,
        lsh_u8 *hashval
);
```

매개 변수

algtype [in]

LSH_MAKE_TYPE 매크로에 의해 만들어진 LSH 알고리즘 명세 값

data [in]

해시 값을 계산할 메시지

databitlen [in]

메시지의 길이 (비트단위)

hashval [out]

해시 값을 저장할 버퍼

반환 값

정상 수행한 경우 LSH_SUCCESS(=0)를, 에러가 발생한 경우 에러 코드를 반환한다.

설명

algtype에 명시된 LSH명세에 따라 LSH해시를 계산한다. algtype 값의 출력 비트 길이가 8의 배수가 아닌 경우, 해시 값의 마지막 바이트는 상위 비트부터 채워진다.

4.5.6. hmac_lsh_init

HMAC 계산을 위해 내부 상태 구조체를 초기화한다.

문법

```
lsh_err hmac_lsh_init(
        struct HMAC_LSH_Context *ctx,
        const lsh_type algtype,
        const lsh_u8 *key,
        size_t keybytelen
);
```

매개 변수

ctx [in/out]

HMAC LSH 계산을 위해 필요한 내부 상태를 저장할 구조체

algtype [in]

LSH_MAKE_TYPE 매크로에 의해 만들어진 LSH 알고리즘 명세 값

key [in]

HMAC 연산에 사용할 키

keybytelen [in]

키의 길이(바이트단위)

반환 값

정상 수행한 경우 LSH_SUCCESS(=0)를, 에러가 발생한 경우 에러 코드를 반환한다.

설명

algtype에 명시된 LSH명세에 따라 HMAC LSH 내부 상태를 저장할 구조체를 초기화한다. LSH-256-224, LSH-256-256, LSH-512-224, LSH-512-256, LSH-512-384, LSH-512-512는 사전 계산된 초기상태 값을 이용하며, 이외의 algtype이 지정된 경우에는 초기상태 값을 새로 계산한다.

4.5.7. hmac_lsh_update

메시지를 입력받아 HMAC 내부 상태를 업데이트한다.

문법

```
lsh_err hmac_lsh_update(
        struct HMAC_LSH_Context *ctx,
        const lsh_u8 *data,
        size_t databytelen
);
```

매개 변수

ctx [in/out]

HMAC LSH 계산을 위해 필요한 내부 상태를 저장할 구조체

data [in]

HMAC 값을 계산할 메시지

databytelen [in]

메시지의 길이 (바이트단위)

반환 값

정상 수행한 경우 LSH_SUCCESS(=0)를, 에러가 발생한 경우 에러 코드를 반환한다.

설명

ctx 구조체의 버퍼에 남아 있는 메시지와 입력받은 메시지를 연접하여 블록단위로 내부 상태를 갱신하고, 남는 메시지는 ctx 구조체의 버퍼에 저장한다.

4.5.8. hmac_lsh_final

최종 HMAC 값을 계산한다.

문법

```
lsh_err hmac_lsh_final(
        struct HMAC_LSH_Context *ctx,
        lsh_u8 *digest
);
```

매개 변수

ctx [in/out]

LSH 해시 계산을 위해 필요한 내부 상태를 저장할 구조체

digest [out]

HMAC 값을 저장할 버퍼

반환 값

정상 수행한 경우 LSH_SUCCESS(=0)를, 에러가 발생한 경우 에러 코드를 반환한다.

설명

ctx 구조체의 버퍼에 남아있는 메시지를 이용하여 최종 내부 상태를 갱신하고, HMAC 값을 계산한다.

4.5.9. hmac_lsh_digest

init, update, final 단계를 한 번에 수행하여 HMAC 값을 계산한다.

문법

```
lsh_err hmac_lsh_digest(
        const lsh_type algtype,
        const lsh_u8 *key,
        size_t keybytelen,
        const lsh_u8 *data,
        size_t databytelen,
        lsh_u8 *digest
);
```

매개 변수

algtype [in]
    LSH_MAKE_TYPE 매크로에 의해 만들어진 LSH 알고리즘 명세 값

key [in]
    HMAC 연산에 사용할 키

keybytelen [in]
    키의 길이(바이트단위)

data [in]
    HMAC 값을 계산할 메시지

databytelen [in]
    메시지의 길이 (바이트단위)

digest [out]
    HMAC 값을 저장할 버퍼

반환 값

정상 수행한 경우 LSH_SUCCESS(=0)를, 에러가 발생한 경우 에러 코드를 반환한다.

설명

algtype에 명시된 LSH명세에 따라 HMAC LSH값을 계산한다.

4.6. LSH 레퍼런스 C 소스코드의 컴파일 방법

LSH의 레퍼런스 C 소스코드는 운영환경, 컴파일 환경에 제한 없이 동작할 수 있도록 구현되어 있다. 또한 C99 표준, 혹은 C99에 준하는 기능을 지원하는 컴파일러에서 컴파일 할 수 있다.
다양한 환경을 고려하였으므로 구동 환경에 적합한 별도의 컴파일·빌드 방법을 이용할 수 있다.

기본 컴파일러인 cc를 이용하여 정적 라이브러리를 빌드하는 예는 다음과 같다.

```
cc -c -O2 src/*.c
ar r liblsh src/*.o
```

4.7. LSH SIMD C 소스코드의 컴파일 방법

LSH의 C 소스코드는 기본적으로 실행 환경과 독립적으로 수행될 수 있도록 구현되어 있다. 그리고 추가적으로 실행 환경의 자원을 활용할 수 있을 경우 그 자원을 활용해 고속화 할 수 있도록 구현되어 있다. LSH C 소스코드에는 다음의 기술을 활용해 LSH를 고속화 할 수 있는 기법이 적용되어있다.

기술	분류	적용 부분
SSE2	SIMD	LSH 해시 압축 함수 계산
SSSE3		
AVX2		
XOP		
NEON		

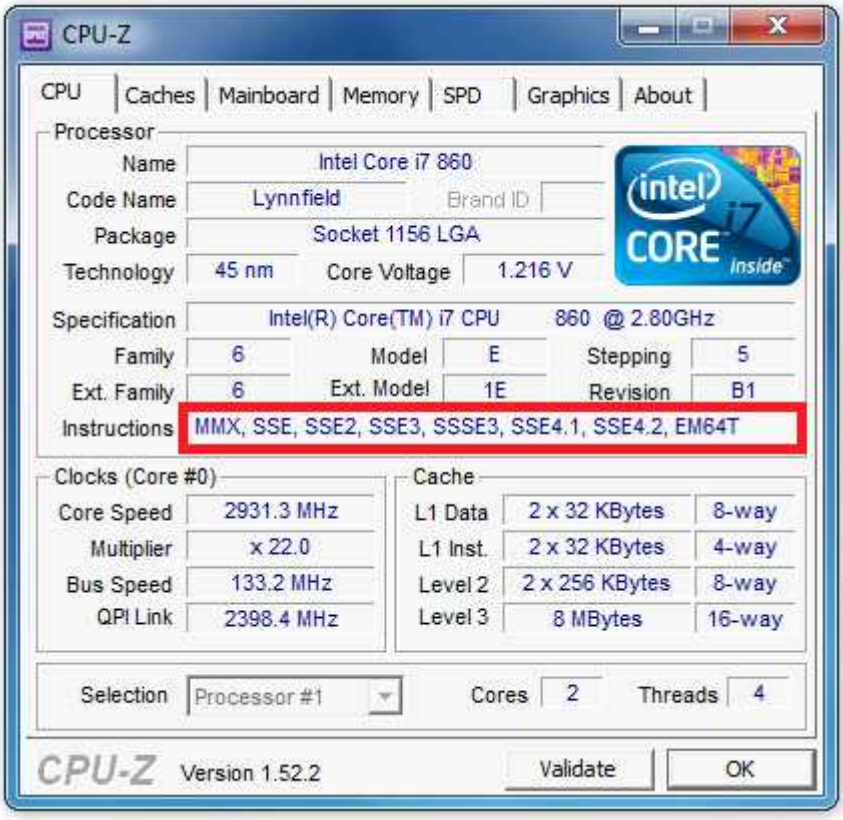
이 기능은 컴파일 시 각 SIMD를 활용한 코드를 탑재하고, 실행 환경에 따라 CPU에서 지원하는 SIMD를 실행하도록 구성되어 있다. 아래에는 대표적인 개발도구인 Visual Studio와 GCC를 이용할 때의 설정방법을 설명한다.

4.7.1. Visual Studio 설정

Visual Studio는 버전에 따라 적용할 수 있는 기술의 종류가 다르다. 또한 이 기술들은 CPU에서 지원해야 적용 가능하다. 다음은 기술별 컴파일 가능한 Visual Studio 버전과 CPU 아키텍처이다. CPU가 지원한다 하더라도 Visual Studio에서 지원하지 않는 기술일 경우, 컴파일 시 해당 SIMD를 사용한 코드가 처리되지 않으므로 주의한다.

기술	Visual Studio 버전	CPU 아키텍처	
		Intel	AMD
SSE2	Visual Studio 6.0 sp5 이상	Pentium 4 이상	Opteron 이상
SSSE3	Visual Studio 2008 이상	Conroe 이상	Bobcat 이상
AVX2	Visual Studio 2013 sp1 이상	Haswell 이상	Excavator 이상
XOP	Visual Studio 2010 sp1 이상	미지원	Bulldozer 이상

실행 환경에서 지원하는지는 CPU-Z (http://www.cpuid.com/softwares/cpu-z.html)를 통해서도 확인할 수 있다.



Visual Studio 솔루션 파일 사용

Visual Studio에서 컴파일하기 위해 vs 폴더에 Visual Studio용 솔루션 파일을 제공한다. 솔루션 파일을 이용하면 정적 라이브러리, DLL, 테스트 벡터 프로그램이 생성되며, 생성된 프로그램은 프로그램을 실행하는 환경이 제공하는 SIMD를 선택하여 사용한다.

만약 Visual Studio 2013 SP1 이상을 가지고 있지 않다면, 프로젝트의 플랫폼 도구 집합을 v120에서 해당 비주얼 스튜디오에 맞는 값으로 변경해주어야 하며, 이렇게 해결되지 않을 경우 .vcproj 파일을 열어 ToolsVersion="12.0" 으로 설정된 부분을 ToolsVersion="4.0" 으로 변경하여 시도해볼 수 있다.

별도 프로젝트 사용

제공된 Visual Studio용 솔루션 파일을 이용하지 않는 경우, SSE2, XOP, AVX2를 사용한 파일을 컴파일하기 위해 코드 생성 옵션을 주어야 한다. SIMD를 적용한 C 파일(lsh256_*.c, lsh512_*.c)의 속성 -> C/C++ -> 코드 생성으로 들어가 "고급 명령 집합 사용"을 다음과 같이 설정한다.

파일명	기술	고급 명령 집합 사용
lsh256_sse2.c lsh512_sse2.c	SSE2	스트리밍 SIMD 확장 2(/arch:SSE2)
lsh256_ssse3.c lsh512_ssse3.c	SSSE3	스트리밍 SIMD 확장 2(/arch:SSE2)
lsh256_xop.c lsh512_xop.c	XOP	스트리밍 SIMD 확장 2(/arch:SSE2)
lsh256_avx2.c, lsh512_avx2.c	AVX2	고급 벡터 확장 2 (/arch:AVX2)

이후 실행코드와 함께 컴파일 시 SSE2, SSSE3, XOP, AVX2 SIMD 연산을 지원하는 코드로 컴파일 된다.

만약 일부 SIMD 구현을 제외하고자 한다면, 해당 코드를 프로젝트에서 제외하고, lsh_def.h 의 LSH_NO_**** 항목을 주석 해제하여 컴파일할 수 있다.

4.7.2. GCC 설정

GCC도 버전에 따라 적용할 수 있는 기술이 다르다. 다음은 각 기술에 대한 GCC 버전과 필요한 옵션을 정리한 것이다.

기술	GCC 버전	옵션
SSE2	3.0 이상	-msse2
SSSE3	4.3 이상	-mssse3
AVX2	4.7 이상	-mavx2
XOP	4.5 이상	-mxop
NEON	4.6 이상	-mfloat-abi=softfp -mfpu=neon

또한 해당 기술은 CPU가 지원해야 적용할 수 있으므로 실행 환경의 CPU가 다음 기술을 지원하는지 확인해야 한다.

기술	CPU 아키텍처		
	Intel	AMD	ARM
SSE2	Pentium 4 이상	Opteron 이상	미지원
SSSE3	Conroe 이상	Bobcat 이상	미지원
AVX2	Haswell 이상	Excavator 이상	미지원
XOP	미지원	Bulldozer 이상	미지원
NEON	미지원	미지원	Cortex-A series

제공된 Makefile 사용

GCC에서 컴파일하기 위해 GNU Makefile을 제공한다. GNU Makefile을 이용하면 지정된 cc, ar, as, ranlib을 이용하여 컴파일을 시도한다. GNU Makefile은 POSIX 환경(MinGW, Cygwin, Linux 등)과 x86, x86_64, ARM 환경을 지원하며, 컴파일 시 동적 라이브러리 파일(DLL, SO)과 테스트 벡터를 생성한다. 정적 라이브러리 파일은 “make SHARED=FALSE” 명령을 통해 생성 가능하다. 환경에 따라 GNU make임을 명시하여 make 대신 gmake를 사용하거나, gcc48 등과 같이 CC 변수에 gcc의 버전을 명시해야 할 수도 있다.

사용 예제는 다음과 같다.
- 리눅스 환경에서 동적 라이브러리 파일(so) 생성 및 설치
make install
- 리눅스 환경에서 정적 라이브러리 파일(a) 생성
make SHARED=false
- 리눅스 환경에서 ARM 크로스 컴파일러를 이용한 정적 라이브러리 파일(a) 생성
(사용하는 크로스 컴파일러에 따라 각 변수의 이름이 바뀔 수 있음)
make CCPREFIX=arm-linux-gnueabi-
- FreeBSD 환경에서 gcc 4.8 버전을 사용하여 동적 라이브러리 파일(so) 생성 및 설치
gmake CC=gcc48 install

- Solaris 환경에서 OpenCSW로 설치한 gcc 4.8 버전을 사용하여 동적 라이브러리 파일(so) 생성 및 설치
export PATH=/opt/csw/bin:$PATH
gmake install

정적 라이브러리 만들기

직접 정적 라이브러리를 만들기 위해서는 2단계를 수행하여야 한다. 먼저 소스코드를 원하는 옵션을 적용하여 컴파일하고, 다음으로는 이를 하나의 정적 라이브러리로 통합한다.

1단계를 수행하기 위해 포함시키고자 하는 소스코드와 기능을 선택한 후 각 C 파일을 컴파일 한다.
컴파일 명령어는

gcc -c [옵션] [파일 이름]

이다.
-c 옵션은 컴파일은 하되 링킹은 하지 않도록 하는 옵션으로 반드시 포함해야 한다. 이 외에 필요한 옵션이 있다면 [옵션]에 포함시킬 수 있다.
다음은 소스코드를 이용하여 SSE2만을 지원하는 LSH에 대한 정적 라이브러리를 컴파일 하는 예제이다.

```
gcc -c -O2 lsh.c cpu_info_ia32.c hmac.c no_arch/*.c -DNO_AVX2 -DNO_XOP -DNO_SSSE3
gcc -c -fPIC -O2 -msse2 sse2/*.c
```

모든 소스코드를 사용하지 않는 경우 각 파일을 위의 방법으로 컴파일하거나 Makefile을 이용하여 컴파일 한다.

2단계로 컴파일한 각 오브젝트를 하나의 정적 라이브러리로 통합한다. 이를 위해 ar 이라는 툴을 사용한다. ar 명령어는

ar r [옵션] [정적 라이브러리 이름] [컴파일한 오브젝트]

이다. r 옵션은 정적 라이브러리가 없을 경우 새로 생성하라는 옵션으로 반드시 포함해야 한다. 그 외에 필요한 옵션을 추가해도 된다.

동적 라이브러리 만들기

직접 동적 라이브러리를 만들기 위해서는 정적 라이브러리를 만들 때와 같이 2단계를 수행하여야 한다.

1단계를 수행하기 위해 포함시키고자 하는 소스코드와 기능을 선택한 후 각 C 파일을 컴파일 한다.
컴파일 명령어는

gcc -c -fPIC [옵션] [파일 이름]

이다. -c 옵션은 컴파일은 하되 링킹은 하지 않도록 하는 옵션으로 반드시 포함하여야 한다. 또 fPIC 옵션은 Position Independent Code, 즉 위치와 무관하게 동작하는 코드를 생성하여 각 프로그램이 동시에 사용할 수 있도록 해주는 옵션으로 반드시 포함하여야 한다. 다음은 소스코드를 이용하여 AVX2, XOP, SSE2, SSSE3를 지원하는 LSH에 대한 동적 라이브러리를 컴파일하는 예제이다.

```
gcc -c -fPIC -O2 lsh.c cpu_info_ia32.c hmac.c no_arch/*.c
gcc -c -fPIC -O2 -mavx2 -msse2 avx2/*.c
gcc -c -fPIC -O2 -msse2 -mssse3 -mxop xop/*.c
```

```
gcc -c -fPIC -O2 -msse2 sse2/*.c
gcc -c -fPIC -O2 -msse2 -mssse3 ssse3/*.c
```

2단계로 컴파일한 각 오브젝트를 하나의 동적 라이브러리로 통합한다. 명령어는

gcc -shared -o [동적 라이브러리 이름] [컴파일한 오브젝트]

이다.

4.7.3. Android NDK 빌드

LSH C 소스코드의 arm 폴더 하위의 jni 폴더에 Android NDK로 빌드할 수 있는 소스코드 및 Makefile이 포함되어 있다. arm\jni 폴더에서 ndk-build 명령어를 수행하면 libs 폴더가 생성되면서 armeabi-v7a, arm64-v8a 폴더에 각각 32비트, 64비트 버전으로 컴파일된 라이브러리 파일이 생성된다. (NDK 설치 방법 및 빌드 환경 설정 방법은 Android 홈페이지 https://developer.android.com/ndk/index.html 을 참고한다. 생성된 라이브러리 파일 및 prebuilt/c_android_ndk/libs 폴더 밑의 LshNative.jar 파일을 Android 앱 프로젝트에 추가하여 사용하면 된다.

5. Java 소스코드

5.1. 서비스

LSH의 Java 소스코드는 다음의 서비스를 제공한다.

- 해시함수 LSH-256, LSH-512
- LSH 기반의 메시지 인증 코드(HMAC)

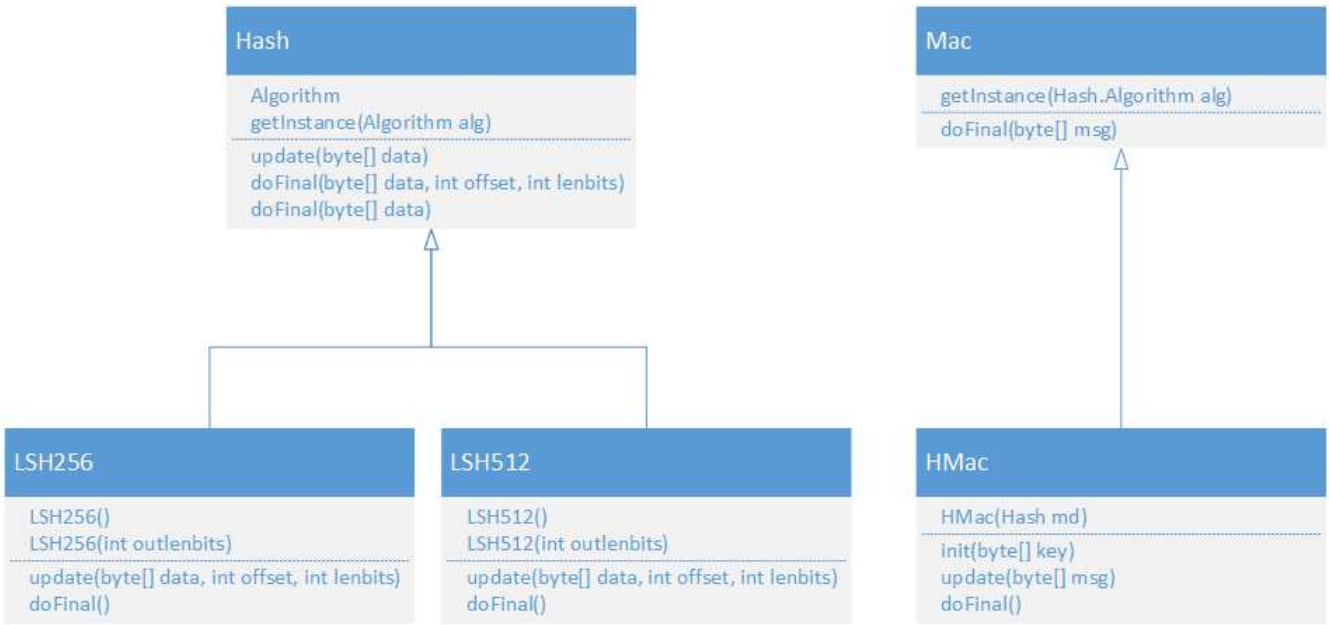
5.2. 소스코드의 구성

LSH의 Java 소스코드는 다음과 같이 구성되어 있다.

파일명	내용
Hash.java	LSH에 사용되는 공통 함수를 정의한 추상 클래스
LSH256.java	LSH256 구현 클래스
LSH512.java	LSH512 구현 클래스
Mac.java	MAC 계산에 사용되는 함수 인터페이스
HMac.java	HMAC 한 블록 암/복호 구현
PackLE.java	int, long <-> byte 변환 함수 모음

5.3. Java 소스코드 구조

LSH Java 소스코드는 객체지향 설계기법을 따라 [그림 5]와 같은 클래스 계층 구조를 하고 있다. Hash 클래스에 공통 인터페이스가 정의되어있고, 워드 길이의 차이에 따른 상세 구현은 LSH256, LSH512 클래스에 구현되어있다. HMAC도 마찬가지로 Mac 클래스에 공통 인터페이스가 정의되어 있으며 HMac을 위한 세부 내용이 HMac 클래스에 구현되어있다.



[그림 5] LSH Java 소스코드 클래스 다이어그램

5.4. Java 소스코드의 컴파일 방법

LSH, HMAC이 포함된 라이브러리를 컴파일하기 위해 LSH 폴더의 build.bat (build.sh) 파일을 이용할 수 있다. 해당 스크립트를 수행하고 나면 lsh.jar 파일이 생성된다.

```
$ build.sh
```

테스트 코드를 실행하기 위해서 LSH_testcode 폴더에 lsh.jar 파일을 복사해두고, build.bat (build.sh) 파일을 이용하여 컴파일 할 수 있다.

```
$ build.sh
```

테스트 코드 실행은 bin 폴더에 들어가 run.bat (run.sh) 파일을 실행하면 된다.

```
$ run.sh
```

5.5. 소스코드 사용법

LSH의 Java 소스코드를 사용하는 방법은 공통적으로 객체 생성(내부적으로 init 수행), 메시지 추가(update), 정리(doFinal)의 3단계를 거친다. Hash 클래스의 getInstance 메서드를 사용하거나, LSH256, LSH512 클래스의 객체를 만들어 사용한다.

5.5.1. 해시 값 계산

코드

```
// 객체 생성
Hash lsh = Hash.getInstance(Hash.Algorithm.LSH256_256);

// 해시 값 계산
lsh.update(data1);
lsh.update(data2);
byte[] hash = lsh.doFinal();
```

코드설명

Hash 클래스의 getInstance 함수를 통해 LSH256 혹은 LSH512 객체를 얻고, 해시를 계산할 메시지를 update 메서드를 이용하여 입력으로 넣다가, doFinal 메서드를 사용하여 최종 해시 값을 계산한다.

public static Hash getInstance(Algorithm algorithm)

매개 변수

Algorithm algorithm [in]

LSH256_224, LSH256_256, LSH512_224, LSH512_256, LSH512_384, LSH_512_512 의 6개 알고리즘을 선택할 수 있다.

반환 값

LSH256 혹은 LSH512 객체

설명

algorithm에 정의된 알고리즘 명세에 맞게 LSH객체를 초기화하여 반환한다.

public static byte[] digest(Algorithm algorithm, byte[] data)

매개 변수

Algorithm algorithm [in]

LSH256_224, LSH256_256, LSH512_224, LSH512_256, LSH512_384, LSH_512_512 의 6개 알고리즘을 선택할 수 있다.

byte[] data [in]

해시 값을 계산할 메시지

반환 값

algorithm을 사용하여 계산한 data에 대한 해시 값

설명

algorithm에 맞는 해시 객체를 생성하고, data에 대한 해시 값을 계산하여 반환한다.

public static byte[] digest(Algorithm algorithm, byte[] data, int offset, int lenbits)

매개 변수

Algorithm algorithm [in]

LSH256_224, LSH256_256, LSH512_224, LSH512_256, LSH512_384, LSH_512_512 의 6개 알고리즘을 선택할 수 있다.

byte[] data [in]

해시 값을 계산할 메시지

int offset [in]

메시지 시작 위치

int lenbits [in]

메시지 길이, 비트 단위

반환 값

algorithm을 사용하여 계산한 data의 일부에 대한 해시 값

설명

algorithm에 맞는 해시 객체를 생성하고, data의 일부에 대한 해시 값을 계산하여 반환한다.

public void update(byte[] msg)

매개 변수

byte[] msg [in]
    해시 값을 계산할 메시지

반환 값

없음

설명

버퍼에 남아있는 메시지와 입력 메시지를 연접하여 블록단위로 내부 상태를 갱신하고, 남는 메시지는 버퍼에 저장한다.

public void update(byte[] msg, int offset, int lenbits)

매개 변수

byte[] msg [in]
    해시 값을 계산할 메시지

int offset [in]
    메시지 시작 위치, 바이트 단위

int lenbits [in]
    메시지 길이, 비트 단위

반환 값

없음

설명

버퍼에 남아있는 메시지와 입력 메시지의 일부를 연접하여 블록단위로 내부 상태를 갱신하고, 남는 메시지는 버퍼에 저장한다.

public byte[] doFinal(byte[] msg)

매개 변수

byte[] msg [in]
    해시 값을 계산할 메시지

반환 값

해시 값

설명

버퍼에 남아있는 메시지와 입력 메시지를 연접하여 내부 상태를 갱신하고, 최종 해시 값을 계산한다.

public byte[] doFinal(byte[] msg, int offset, int lenbits)

매개 변수

byte[] msg [in]
    해시 값을 계산할 메시지

int offset [in]
    메시지 시작 위치 (바이트단위)

int lenbits [in]
    메시지 길이, 비트 단위

반환 값

해시 값

설명

버퍼에 남아있는 메시지와 입력 메시지의 일부를 연접하여 내부 상태를 갱신하고, 최종 해시 값을 계산한다.

public byte[] doFinal()

매개 변수

없음

반환 값

해시 값

설명

버퍼에 남아있는 메시지를 이용하여 내부 상태를 갱신하고, 최종 해시 값을 계산한다.

5.5.2. HMAC

코드

```
// 객체 생성
Mac mac = new HMac(Hash.getInstance(LSH256_256));

// mac 생성
mac.init(key);
mac.update(data);
byte[] hmac = mac.doFinal();
```

코드설명

HMac 객체를 초기화할 때, HMAC 계산에 사용할 LSH 객체를 입력해주어야 하며, init 메서드를 이용하여 key를 주입하고, update 메서드를 이용하여 HMAC 계산에 사용되는 메시지를 입력하며, doFinal 메서드를 이용하여 최종 HMAC 값을 계산한다.

public void init(byte[] key)

매개 변수

byte[] key [in]

HMAC 생성에 사용될 키

반환 값

없음

설명

HMAC 계산을 위한 키를 입력받아 초기화한다.

public void update(byte[] msg)

매개 변수

byte[] msg [in]

HMAC을 계산할 메시지

반환 값

없음

설명

버퍼에 남아 있는 메시지와 입력받은 메시지를 연접하여 블록단위로 HMAC 계산을 수행하고, 남는 메시지를 버퍼에 저장한다.

public byte[] doFinal(byte[] msg)

매개 변수

byte[] msg [in]

HMAC 값을 계산할 메시지

반환 값

HMAC 값

설명

버퍼에 남아있는 메시지와 새로 입력받은 메시지를 연접하여 내부 상태를 갱신하고, HMAC 값을 계산한다.

public byte[] doFinal()

매개 변수

없음

반환 값

HMAC 값

설명

버퍼에 남아있는 메시지를 이용하여 내부 상태를 갱신하고, HMAC 값을 계산한다.

public static byte[] digest(Hash.Algorithm algorithm, byte[] key, byte[] msg)

매개 변수

Hash.Algorithm algorithm [in]

HMAC 값 계산에 사용할 Hash 알고리즘

byte[] key [in]

HMAC 키

byte[] msg [in]

HMAC 값을 계산할 메시지

반환 값

HMAC 값

설명

Hash 객체, 키, 메시지를 이용하여 HMAC 값을 계산한다.

6. Python 소스코드

6.1. 서비스

LSH의 Python 소스코드는 Python 2.7 이상, 또는 Python 3.2 이상의 환경에서 다음의 서비스를 제공한다.

- 해시함수 LSH-256, LSH-512
- LSH 기반의 메시지 인증 코드(HMAC)

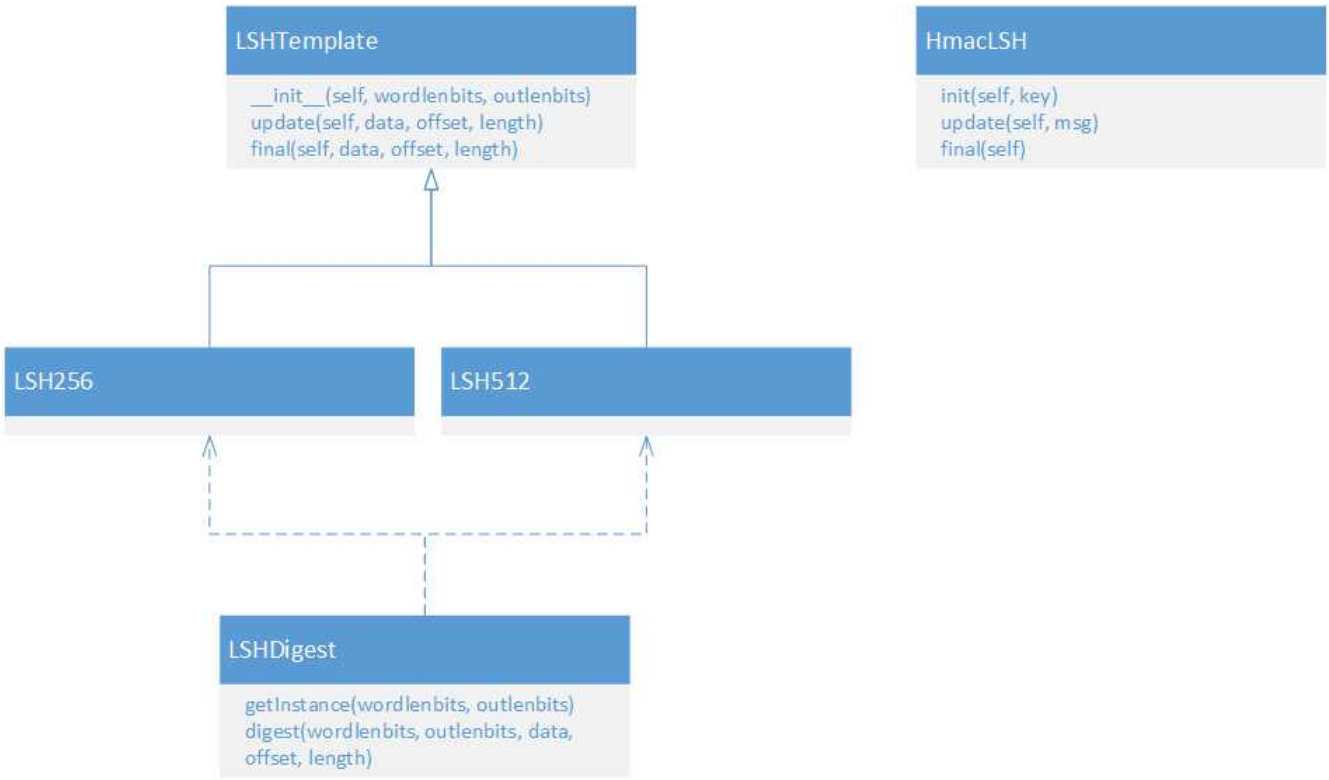
6.2. 소스코드의 구성

LSH의 Python 소스코드는 다음과 같이 구성되어 있다.

파일명	내용
setup.py	모듈 설치를 위한 파일
lsh/__init__.py	모듈 import를 위한 파일
lsh/lsh_digest.py	LSH 해시함수 wrapper 클래스
lsh/lsh_template.py	LSH 공통 구현
lsh/lsh256.py	LSH256 구현
lsh/lsh512.py	LSH512 구현
lsh/hmac_lsh.py	LSH기반 HMAC구현

6.3. 소스코드의 구조

LSH Python의 소스코드는 객체지향 설계기법에 따라 [그림 6]과 같이 구성되어있다. 공통적인 인터페이스는 모두 LSHTemplate 클래스에 정의하여 구현하였으며, LSH256, LSH512 클래스에 각각 세부 구현별로 달라지는 부분이 정의되어있다. LSHDigest 클래스는 LSH256이나 LSH512 객체를 생성하기 위해 사용하는 일종의 팩토리 클래스이며, HMAC 계산은 HmacLSH 클래스에 구현되어있다.



[그림 6] LSH Python 소스코드의 클래스 다이어그램

6.4. 소스코드 컴파일

소스코드 상위 폴더의 설치 스크립트(윈도우:setup.bat, 리눅스:setup2.sh/setup3.sh)를 실행하면 LSH 소스코드가 컴파일 된 후 패키지를 설치한다. 패키지 설치 후 테스트를 위해 vs 폴더 안에 있는 lshvs.py 혹은 hmacvs.py 파일을 python 명령어로 실행하면 된다.

```
$ ./setup2.sh
$ cd vs
$ python2 lshvs.py
$ python2 hmacvs.py
```

```
$ ./setup3.sh
$ cd vs
$ python3 lshvs.py
$ python3 hmacvs.py
```

6.5. 소스코드 사용법

LSH의 Python 소스코드를 사용하는 방법은 공통적으로 객체 생성(내부적으로 init 수행), 메시지 추가(update), 정리(final)의 3단계를 거친다. 메시지를 여러 개로 나누어 입력하더라도 online 처리할 수 있도록 하여 대용량 데이터의 메시지 인증코드 생성을 가능하게 한다. 지원하는 메시지의 최소 단위는 바이트이다.

입출력 시 사용하는 키, 메시지 등의 기본 자료형은 **bytearray**이며, 해당 자료형 대신에 **Buffer Protocol**, **str**(Python 2), **bytes**(Python 3), **memoryview**와 같은 bytes-like-object, 256 미만의 정수로 이루어진 list를 입력값으로 사용 할 수 있다.

6.5.1. 해시 값 계산

코드

```
from lsh import LSHDigest

lsh = LSHDigest.getInstance(256, 256);
lsh.update(msg);
hash = lsh.final();
```

코드설명

LSHDigest 클래스의 getInstance 함수를 이용하여 LSH 객체를 생성하고, 해시 값을 계산할 메시지를 update 함수를 통해 입력한 후, 최종 해시 값을 final 함수를 사용하여 계산한다.

LSHDigest.getInstance

매개 변수

int wordlenbits

비트단위 워드 길이, 256, 512만 선택할 수 있으며, 그 외의 값을 입력하면 ValueError가 발생한다.

int outlenbits

비트단위 출력 길이, 1 <= outlenbits <= 워드길이 사이의 값만 유효하며, 그 외의 값을 입력하면 ValueError가 발생한다.

반환 값

LSH256 혹은 LSH512 객체

wordlenbits와 outlenbits 값에 따라 초기화된 LSH256 혹은 LSH512 객체를 반환한다.

설명

LSH256 혹은 LSH512 객체를 생성하고 초기화하여 반환한다.

LSHDigest.digest

매개 변수

int wordlenbits

비트단위 워드 길이, 256, 512만 선택할 수 있으며, 그 외의 값을 입력하면 ValueError가 발생한다.

int outlenbits

비트단위 출력 길이, 1 <= outlenbits <= 워드길이 사이의 값만 유효하며, 그 외의 값을 입력하면 ValueError가 발생한다.

bytearray msg

해시 값을 계산할 메시지

int offset

메시지 시작 위치, 기본 값은 0

int lenbits

비트단위 메시지 길이, 기본 값은 전체 메시지 길이

반환 값

bytearray

최종 계산된 해시 값

설명

LSH256 혹은 LSH512 객체를 생성하고, 메시지를 이용하여 내부 상태를 갱신한 뒤, 해시 값을 계산하여 반환한다.

LSH256.update / LSH512.update

매개 변수

bytearray msg

해시 값을 계산할 메시지

int offset

메시지 시작 위치, 기본 값은 0

int lenbits

비트단위 메시지 길이, 기본 값은 전체 메시지 길이

반환 값

없음

설명

버퍼에 남아있는 메시지와 입력 메시지를 연접하여 내부 상태를 갱신하고, 최종 해시 값을 계산한다.

LSH256.final / LSH512.final

매개 변수

bytearray msg

해시 값을 계산할 메시지, 기본 값은 None

int offset

메시지 시작 위치, 기본 값은 0

int lenbits

비트단위 메시지 길이, 기본 값은 전체 메시지 길이

반환 값

bytearray

최종 계산된 해시 값

설명

버퍼에 남아있는 메시지를 이용하여 내부 상태를 갱신하고, 최종 해시 값을 계산한다.

6.5.2. HMAC

코드

```
from lsh import HmacLSH

hmac = HmacLSH(256, 256)
hmac.init(key)
hmac.update(msg)
result = hmac.final()
```

코드설명

HMAC 클래스 생성자에 워드길이와 출력해시길이를 이용하여 객체를 초기화한 후, init 함수를 이용하여 key를 입력한다. update 함수를 통해 HMAC을 계산하고자 하는 메시지를 입력한 후, final 함수를 이용하여 최종 HMAC 값을 계산한다.

HmacLSH

매개 변수

int wordlenbits 워드 길이 (비트단위)

int outlenbits 출력 해시 길이 (비트단위)

반환 값

HmacLSH 객체

update, final을 메소드로 제공하는 HMAC 계산을 위한 객체를 반환한다.

설명

HMAC 계산을 위한 객체를 생성한다.

HmacLSH.init

매개 변수

bytearray key

HMAC 계산에 필요한 키

반환 값

없음

설명

HMAC 계산에 필요한 키를 입력받아 내부 상태를 초기화한다.

HmacLSH.update

매개 변수

bytearray msg

HMAC 계산을 하고자 하는 메시지

반환 값

없음

설명

버퍼에 남아있는 메시지와 입력받은 메시지를 연접하여 내부 상태를 갱신하고, 남는 메시지는 버퍼에 저장한다.

HmacLSH.final

매개 변수

없음

반환 값

bytearray

최종 계산된 HMAC 값

설명

버퍼에 남아있는 메시지를 이용하여 내부 상태를 갱신하고, 남는 메시지는 버퍼에 저장한다.

HmacLSH.digest

매개 변수

int wordlenbits

비트단위 워드길이, 256, 512만 선택할 수 있으며, 그 외의 값을 입력하면 ValueError가 발생한다.

int outlenbits

비트단위 출력길이, 1 <= outlenbits <= 워드길이 사이의 값만 유효하며, 그 외의 값을 입력하면 ValueError가 발생한다.

bytearray key

HMAC 키

bytearray msg

HMAC 값을 계산할 메시지

반환 값

bytearray

키와 메시지를 이용하여 계산한 HMAC 값

설명

HmacLSH 객체를 생성하고, 키와 메시지를 이용하여 HMAC 값을 계산한다.

코드 관련 문의: cryptoalg@nsr.re.kr