



MYONGJI
UNIVERSITY

Threshold EdDSA 구현 과제

최종 보고 및 코드 시연

정보보안연구실

February 5, 2020

Table of contents

- 1 연구 범위 및 내용**
 - 연구 범위
 - 연구 내용
- 2 1차 과제 대비 변경**
 - ECDSA vs EdDSA
 - 최적화, 보안 위협, 기타 수정
- 3 API 사용 및 주의사항**
 - API 사용
 - 주의사항
- 4 Unit Tests**
- 5 참고문헌**

연구 목표

본 과제의 목표는 EdDSA 기반의 Threshold Signature Scheme의 구현임

1 주요 구현 범위

- 1 Secret Sharing Scheme
- 2 Verifiable Secret Sharing Scheme
- 3 Generating a Random Secret
- 4 EdDSA
- 5 A (t, n) EdDSA Threshold Signature Scheme

2 산출물

- 1 Threshold EdDSA 소스 코드
- 2 구현 산출물 관련 Unit Test
- 3 발표자료

연구 내용

- Threshold Signature Scheme 설계 원리 분석
 - 1 Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates [1]
 - 2 Fast Multiparty Threshold ECDSA with Fast Trustless Setup [2]

- 목표 설계를 위한 기반 기술 분석 및 구현
- Threshold EdDSA Signature Scheme 정의 및 구현

Table of contents

- 1** 연구 범위 및 내용
 - 연구 범위
 - 연구 내용
- 2** 1차 과제 대비 변경
 - ECDSA vs EdDSA
 - 최적화, 보안 위협, 기타 수정
- 3** API 사용 및 주의사항
 - API 사용
 - 주의사항
- 4** Unit Tests
- 5** 참고문헌

ECDSA

- ECDSA (Elliptic Curve Digital Signature Algorithm)은 타원곡선을 이용한 전자서명 알고리즘임
- 2018년 Rasario Gennaro and Steven Goldfeder [2]에 의해 Threshold ECDSA가 개발됨
- 2019년 Penta Security와 고려대학교의 산학과제를 통해 Threshold ECDSA의 C 언어 코드가 개발됨
- 본 과제는 Threshold ECDSA과제 산출물을 참조하여 Threshold EdDSA를 개발하는 것을 목표로 함

ECDSA 구조

ECDSA

□ Key Generation

- ❖ x : Private Key
- ❖ Y : Public Key, $Y = xG$

□ Signature Generation

- ❖ $m = h(M)$
- ❖ Secret random number k^{-1}
- ❖ $r = x$ -coordinate of ($R = k^{-1}G$)
- ❖ $s = k(m + rx)$
- ❖ Signature : (r, s)

□ Signature Verification

- ❖ $m = h(M)$
- ❖ $R' = s^{-1}mG + s^{-1}rY$
- ❖ Check $r = x$ -coordinate of R'

ECDSA Sign부와 Threshold ECDSA

- ECDSA의 서명 과정은 다음과 같음
 - $r = k^{-1}G$
 - $s = k(m + rx)$
- 간략히 설명하면 ¹, r 과 s 의 생성에 사용되는 값 중 k 와 x 를 MPC의 입력으로 사용될 partial data로 볼 수 있음
- Threshold ECDSA의 구현을 위해
 - partial data k_i 로부터 k^{-1} 과 km 을 구해야 함
 - k_i 와 x_i 를 모두 이용하여 krx 를 구해야 함
- 모든 과정이 Gennaro [2]에 기재되어 있으며 1차 과제 [3]를 통해 C 코드 산출물을 얻음
- Threshold EdDSA도 [2]를 간단히 수정하여 구현할 수 있을 것으로 기대하였으나, 1차 과제 [3]를 분석한 결과 1차 과제의 이론적 배경 [2]을 그대로 사용할 수 없음을 확인함

¹ mapping 과정 등을 이해의 편의를 위해 생략함

EdDSA

- 공개키 서명 알고리즘 EdDSA (Edwards-curve Digital Signature Algorithm) [4]은 Schnorr 서명 알고리즘의 변형으로 Twisted Edwards curves에 기반함. EdDSA의 목적은 보안 강도 희생 없이 빠른 속도를 제공하는 것임
- 본 과제에서는 EdDSA 구현 중 SHA-512와 Curve25519를 사용한 Ed25519를 사용함. 특별한 설명이 없는 경우 본 문서의 EdDSA는 Ed25519를 지칭함
 - Ed25519는 EdDSA의 저자인 Daniel J. Bernstein이 직접 x86-64 Nehalem/Westmere 프로세서에 대해 최적화를 수행함

EdDSA 구조

EdDSA

□ Key Generation

- ❖ u : Secret Value
- ❖ x : Private Key, $x = h_L(u)$
- ❖ Y : Public Key, $Y = xG$

□ Signature Generation

- ❖ $k = h(h_H(u), M)$
- ❖ $R = kG$
- ❖ $m = h(R, Y, M)$
- ❖ $s = r + xm$
- ❖ Signature : (R, s)

□ Signature Verification

- ❖ $m = h(R, Y, M)$
- ❖ Check $R = sG - h(R, Y, M)Y$

EdDSA Sign부와 Threshold EdDSA

- EdDSA의 서명 과정은 다음과 같음
 - $R = kG$
 - $s = r + xm$, where $m = h(R, Y, M)$
- k, x 에 대한 partial data k_i, x_i 에 대해
- ECDSA의 경우와 달리 Threshold EdDSA의 구현에는 다음 연산이 필요함
 - partial data k_i 로부터 r 을 구해야 함
 - partial data x_i 로부터 xm 을 구해야 함
- 상기한 차이점으로 인해 Gennaro [2]를 통해 Threshold EdDSA를 구현할 수 없음을 확인함
- 본 과제에서는 Threshold Schnorr 서명을 위한 Stinson [1]을 수정하여 Threshold EdDSA를 구현함

Gennaro Scheme

Gennaro Scheme [2]은 Threshold ECDSA를 구현하기 위해 사용됨
구조는 다음과 같음²

- 키 생성 과정에서 각 플레이어가 Feldman VSS를 통해 partial secret key x_i 를 얻음
- 서명 과정에서 각 플레이어가 k_i, γ_i 를 생성
- 서명 과정에서 각 플레이어가 두 번의 MtA³ 연산을 수행함
 - 첫 번째 MtA([2], Phase 1)의 생성물을 통해 $R = k^{-1}G$ 의 연산 가능
 - 두 번째 MtA([2], Phase 2)의 생성물 σ_i 를 통해 kx 의 연산 가능
- $s_i = mk_i + r\sigma_i$ 를 생성함. s_i 의 합이 서명 s 와 같음

²1차 과제 범위에 포함되어 있으므로 간략히 소개함

³Multiply-to-additive partial data conversion subprotocols

Stinson Scheme

Stinson Scheme [1]은 Threshold Schnorr를 구현하기 위해 사용됨⁴
 본 과제에서는 Threshold EdDSA를 구현하기 위한 기초 이론으로 사용함

Stinson Scheme은 Pedersen VSS scheme (Verifiable Secret Sharing scheme)을 이용하여 분산 방식으로 Random Secret을 공유함⁵
 본 발표자료에서는 Stinson Random Secret Sharing을 다음과 같이 표기함

$$(\alpha_1, \dots, \alpha_n) \leftrightarrow_{(t,n)} (x | Y = xG, evidence_Y) \quad (1)$$

이는 Random Secret 공유에 참여하는 각 플레이어 P_i 의 partial secret이 α_i 이며, 이를 (t, n) -threshold를 통해 구할 수 있는 secret이 x , 그리고 xG , 즉 secret에 해당하는 공개값이 Y 임을 의미함
 $evidence_Y$ 는 각 α_i 의 정합성 확인을 위해 사용하는 값임

⁴EdDSA가 아닌 Schnorr임. 따라서 Stinson [1]또한 Threshold EdDSA 구현에 바로 적용될 수 없음

⁵[1, 2.4 Generating a Random Secret]

Stinson Scheme

- 키 생성 과정:
 - $(\alpha_1, \dots, \alpha_n) \leftrightarrow_{(t,n)} (x|Y, evidence_Y)$
- 서명 생성 과정:
 - $(\beta_1, \dots, \beta_n) \leftrightarrow_{(t,n)} (e|V, evidence_V)$
 - 각 플레이어가 $\gamma_i = \beta_i + h(m||V)\alpha_i$ 를 연산
 - $evidence_Y$ 와 $evidence_V$ 를 통해 γ_i 의 정합성 확인
 - 정합성이 확인된 경우 γ_i 를 통해 σ 연산
 - 서명 값 (σ, V) 출력

검증 과정은 일반 Schnorr와 동일함

(참고: Schnorr 서명값은 랜덤한 e 에 대해 $V = eG$, $\sigma = e + h(m||V)x$ 일 때 (σ, V) 쌍이며, 검증은 $\sigma G = V + h(m||V)Y$ 를 통해 수행됨)

과제 구현

본 과제는 주로 Stinson [1]을 참고하여 Threshold EdDSA를 개발함
주요 수정점은 다음과 같음

- Schnorr 관련 수식을 EdDSA 관련 수식으로 수정
- EdDSA 관련 수식으로 수정할 때 Threshold EdDSA가 deterministic 서명값을 출력할 수 있도록 함
 - EdDSA의 특성상 Stinson을 사용하더라도 기존과 완전히 동일하게 작동하는(테스트 벡터가 같은) Threshold EdDSA를 구현할 수 없음
 - 본 과제에서는 가능한 한 원본 EdDSA와 특성을 맞추려고 하였음
- Stinson Random Secret Sharing의 구현은 Pedersen VSS 대신 Gennaro [2]와 동일하게 Feldman VSS를 사용함⁶

⁶Feldman VSS의 구현에 Gennaro [2]의 내용을 참조함

과제 구현 - Detail I

■ 키 생성 과정

- 1 각 플레이어 P_i 가 랜덤하게 u_i 를 선택함
- 2 $l_i = h_L(u_i)$ 를 연산한 뒤, l_i 를 통해 Stinson Random Secret Sharing을 수행
- 3 $(\alpha_1, \dots, \alpha_n) \leftrightarrow_{(t,n)} (x|Y, evidence_Y)$, where $x = \sum_{i=0}^n l_i$

■ 서명 생성 과정

- 1 각 플레이어 P_i 가 $pre_i = h_H(u_i)$ and $l_i = h_L(u_i)$ 를 연산
- 2 각 플레이어가 $k_i = h(pre_i, M)$ 를 연산 후 k_i 를 통해 Stinson Random Secret Sharing을 수행
- 3 $(\beta_1, \dots, \beta_n) \leftrightarrow_{(t,n)} (k|R, evidence_R)$
- 4 각 플레이어가 $m = h(R, Y, M)$ 를 연산. 이 값은 모든 플레이어에게 동일함
- 5 각 플레이어가 $s_i = \beta_i + m\alpha_i$ 를 연산

과제 구현 - Detail II

- 6 $evidence_Y$ 와 $evidence_R$ 을 통해 각 s_i 의 정합성 확인
- 7 정합성이 확인된 경우 s_i 를 통해 s 연산
- 8 서명 값 (R, s) 출력

검증 과정은 일반 EdDSA와 동일함

과제 구현 - Detail III

- 초기 과제 협의사항
 - 키 생성 과정에서 각 플레이어가 랜덤값 x 를 선택한 뒤 x 를 통해 Random Secret 공유
 - 서명 과정에서 각 플레이어가 랜덤값 k 를 선택한 뒤 k 를 통해 Random Secret 공유
- 과제 협의와의 변경점
 - 상기 두 부분은 EdDSA와 형태를 맞추기 위해 u_i 를 선택 후 h_L 과 h_H 를 통해 l_i, pre_i 를 생성한 뒤 Random Secret 공유를 진행하는 것으로 수정됨

과제 구현 - 최적화

VSS의 구현에 1차 과제 [3]의 산출물을 참조하였으나, 일부 코드에 성능 저하 요소가 있었음

- VSS 과정 중 partial data와 evidence를 생성, 검증하는 부분에서, 1차 과제 [3]는 타원곡선상 포인트 P 를 저장할 때 P 를 Affine⁷한 뒤 x 좌표와 y 좌표를 저장하는 방식을 사용함
- 이는 대개의 타원곡선 라이브러리에 존재하는 compress/decompress 함수 쌍을 통해 최적화할 수 있음
- VSS 이외에도 불필요한 Affine 등을 제거하여 최적화를 수행함

⁷타원곡선상 포인트 P 의 좌표 중 z 를 1로 수정

과제 구현 - 최적화

<pre> // secret s * generator G, sG -> P //secret s x, y -> data[0], data[1] pcis_ED25519_Mult_Mint(&state, &P, secret, &ED25519_B); MINT_SetByEC25519IntArray(scheme->data, P.x); MINT_SetByEC25519IntArray(scheme->data+1, P.y); //coefficients -> scheme's data array for (l=0; l<t-1; l++) { pcis_ED25519_Mult_Mint(&state, &P, coeff_a+l, &ED25519_B); MINT_SetByEC25519IntArray(scheme->data+2*(l+1), P.x); MINT_SetByEC25519IntArray(scheme->data+2*(l+1)+1, P.y); } </pre>	<pre> 87 // secret s * generator G, sG -> P 88 //secret s x, y -> data[0], data[1] 89 pcis_ED25519_Mult_Mint(&state, &P, secret, &ED25519_B); 90 + pcis_ED25519_Compress(&P, scheme->data[0]); 91 92 //coefficients -> scheme's data array 93 for (l=0; l<t-1; l++) 94 { 95 pcis_ED25519_Mult_Mint(&state, &P, coeff_a+l, &ED25519_B); 96 + pcis_ED25519_Compress(&P, scheme->data[l+1]); 97 } </pre>
---	---

VSS Share⁸/Evidence 생성 코드 최적화

⁸코드 내의 share는 발표자료의 partial data를 의미함

과제 구현 - 최적화

<pre> 127 VSS_verify(VSS_scheme *scheme, int index, MINT *share) 128 129 130 int i,k,t; 131 132 MINT tmp1 = {0,}; 133 MINT tmp2 = {0,}; 134 135 PC12_ED25519_Point P; 136 PC12_ED25519_Point S; 137 PC12_ED25519_Point p,tmp1; 138 PC12_ED25519_Point p,tmp2; 139 140 MINT_WriteInEC25519IntArray(p,tmp1,x,scheme->data); 141 MINT_WriteInEC25519IntArray(p,tmp1,y,scheme->data+1); 142 curve25519_Copy(p,tmp1,z,ED25519_ORDER); 143 pois_ED25519_MakeAffine(4p,tmp1); // This will safely fill x 144 pois_ED25519_Point_Copy(4p,4p,tmp1); // P = A * B = secret 145 146 //tmp1 = index 147 MINT_SetByInteger(4tmp1, index); 148 t = scheme->len; 149 150 for (i=0; i<t-1; i++) // P = ... A_0 = [A_1 index^i] i=0, ..., t-1 151 { 152 MINT_SetByInteger(4tmp2, 1); 153 //tmp2 = index^(i+1) 154 for (k=0; k<i-1; k++) 155 MINT_Mult(4tmp2, 4tmp2, 4tmp1); 156 MINT_ModClassic(4tmp2, sed25519_order); 157 } 158 //P_i tmp1 = A_i(i+1) 159 MINT_WriteInEC25519IntArray(p,tmp1,x,scheme->data+2*(i+1)); 160 MINT_WriteInEC25519IntArray(p,tmp1,y,scheme->data+2*(i+1)+1); 161 curve25519_Copy(p,tmp1,z,ED25519_ORDER); 162 pois_ED25519_MakeAffine(4p,tmp1); // This will safely fill x 163 164 //P_i tmp2 = A_i(i+1) * index^(i+1) 165 pois_ED25519_Mult_Mint(4state, 4p,tmp2, 4tmp2, 4p,tmp1); 166 pois_ED25519_Add(4state, 4p, 4p, 4p,tmp2); 167 168 pois_ED25519_MakeAffine(4p); 169 170 //S = share * Q 171 pois_ED25519_Mult_Mint(4state, 4s, share, 4ED25519_Q); </pre>	<pre> 127 VSS_verify(VSS_scheme *scheme, int index, MINT *share) 128 129 130 int i,k,t; 131 132 MINT tmp1 = {0,}; 133 MINT tmp2 = {0,}; 134 135 PC12_ED25519_Point P; 136 PC12_ED25519_Point S; 137 PC12_ED25519_Point p,tmp1; 138 PC12_ED25519_Point p,tmp2; 139 140 pois_ED25519_Decompress(4P,scheme->data[0]); 141 142 //tmp1 = index 143 MINT_SetByInteger(4tmp1, index); 144 t = scheme->len; 145 146 for (i=0; i<t-1; i++) // P = ... A_0 = [A_1 index^i] i=0, ..., 147 { 148 MINT_SetByInteger(4tmp2, 1); 149 //tmp2 = index^(i+1) 150 for (k=0; k<i-1; k++) 151 MINT_Mult(4tmp2, 4tmp2, 4tmp1); 152 MINT_ModClassic(4tmp2, sed25519_order); 153 } 154 //P_i tmp1 = A_i(i+1) 155 pois_ED25519_Decompress(4p,tmp1,scheme->data[1+i]); 156 157 //P_i tmp2 = A_i(i+1) * index^(i+1) 158 pois_ED25519_Mult_Mint(4state, 4p,tmp2, 4tmp2, 4p,tmp1); 159 pois_ED25519_Add(4state, 4p, 4p, 4p,tmp2); 160 161 } </pre>
--	--

VSS Share/Evidence 검증 코드 최적화

과제 구현 - 보안 위협

1차 과제 [3]에는 몇몇 부분에 보안 위협이 존재하였음

- Threshold ECDSA의 연산과정에서 비밀키 x 에 해당하는 partial keys는 합쳐지지 말아야 함
- 그러나 Sign_TEST PHASE 0에서 partial keys를 합쳐 x 를 생성하는 부분이 존재함
- 잠재적인 보안 위협을 제거하였음

과제 구현 - 보안 위협

```
printf("-----Sign_TEST_Start-----\n");
printf("-----PHASE0-----\n");
for (int i = 0; i < t; i++) // PCS pks copy
{
    MINT_Copy(vec_pk->data[i]->n, keys_vec[ind_set.data[i] - 1]->ek->n);
    MINT_Copy(vec_pk->data[i]->g, keys_vec[ind_set.data[i] - 1]->ek->g);
    MINT_Copy(vec_pk->data[i]->n2, keys_vec[ind_set.data[i] - 1]->ek->n2);
}

MINT_SetByInteger(x, 0);
for (int i = 0; i < t; i++)
{
    VSS_map_share_to_new( lag_coeff[i], ind_set.data[i], &ind_set, param);
    MINT_Mult( w[i] , lag_coeff[i] , keys_vec[ind_set.data[i] - 1]->x_i );
    MINT_ModClassic(w[i], ec->order);
    MINT_Add_mod( x, x, w[i], ec->order);
}
```

Threshold ECDSA Sign_TEST PHASE 0

과제 구현 - 의존성

- Threshold ECDSA는 krx 를 연산할 필요가 있어 GMP(GNU Multiple Precision Arithmetic Library) 의존성이 추가됨
- Threshold EdDSA는 해당 연산이 필요하지 않기 때문에 GMP 의존성이 삭제됨
- 구체적으로 `gmp >= 6.1.2`에 대한 의존성이 필요하였으나 삭제되었음

MINT의존성도 삭제할 수 있을 것으로 보이나 현 산출물은 MINT에 의존하고 있음. 최적화가 필요할 경우 MINT 의존성을 삭제할 경우 도움이 되리라 예상됨

과제 구현 - 모듈화, Makefile

1차 과제 [3]는 Threshold ECDSA를 main.c 파일 내에 작성함

- main.c 내부에서 사용되는 VSS, MtA 등은 별도 파일에 모듈화가 되어 있으나,
- Threshold ECDSA 자체는 모듈화 되어 있지 않았음
- 본 과제에서는 Threshold EdDSA의 Key Generation, Signature Generation, Verification을 모듈화하였음 (API 사용에서 설명)

또한 main.c를 빌드하기 위해 build.sh 쉘 스크립트 파일이 사용됨

- build.sh 대신 Makefile을 사용함
- Makefile의 형태는 EdDSA의 저자인 Daniel J. Bernstein의 Makefile을 참조함

과제 구현 - 모듈화

```
int main(void)
{
    int n = 5; // total # of party
    int t = 3; // threshold
    VSS_ind_set ind_set = {n, t, {1, 3, 4}};

    //Module_TEST();
    MAIN_TEST(ind_set);
    return 0;
}
```

main.c를 통한 Threshold ECDSA 구현

과제 구현 - Makefile

```
gcc -c ./src/PCS.c      -I ./include -I ./depends/bc-cis/include/crypto
gcc -c ./src/MtA.c     -I ./include -I ./depends/bc-cis/include/crypto
gcc -c ./src/DLProof.c -I ./include -I ./depends/bc-cis/include/crypto
gcc -c ./src/VSS.c     -I ./include -I ./depends/bc-cis/include/crypto
gcc -c ./src/keygen.c  -I ./include -I ./depends/bc-cis/include/crypto
gcc -c ./src/Sign.c    -I ./include -I ./depends/bc-cis/include/crypto
gcc -c ./src/HC.c      -I ./include -I ./depends/bc-cis/include/crypto
gcc -c ./src/util.c    -I ./include -I ./depends/bc-cis/include/crypto
gcc -c ./src/ZKP.c     -I ./include -I ./depends/bc-cis/include/crypto
gcc -c main.c          -I ./include -I ./depends/bc-cis/include/crypto

gcc PCS.o MtA.o DLProof.o VSS.o keygen.o Sign.o HC.o util.o ZKP.o main.o
    -o main -L ./depends/bc-cis -lbc_cis -lgmp
rm PCS.o MtA.o DLProof.o VSS.o keygen.o Sign.o HC.o util.o ZKP.o main.o

./main
```

build.sh를 통한 Threshold ECDSA 구현

과제 구현 - Makefile

```

Makefile
You, 11 hours ago | 1 author (You)
1  LDLIBS=-Ldepends/bc-cis -lbc_cis
2  CC = /usr/bin/gcc
3  CFLAGS = -Iinclude/ -Idepends/bc-cis/include/crypto -g -Wall -Wextra -Wpedantic -O0 -std=c99
4  # Release: -O3, Debug: -O0
5
6  SOURCES = curve25519.c eddsa.c hc.c keygen.c schnorr.c sign.c util.c verify.c vss.c dist_vss.c feldman_tp.c
7
8  .PHONY: clean test
9  TESTS = test/thres_eddsa \
10         test/vss \
11         test/dist_vss \
12         test/tools \
13         test/eddsa \
14         test/thres_eddsa_keygen \
15
16  default: tests
17  all: tests
18  tests: $(TESTS)
19  test: $(TESTS:=.exec)
20  test/%: test/%.c $(SOURCES)
21         $(CC) $(CFLAGS) -o $@ $@ $(SOURCES) $< $(LDLIBS)
22  test/%.exec: test/%
23         @$<
24
25  clean:
26         -$(RM) $(TESTS)

```

Makefile을 통한 Threshold EdDSA 구현

디버깅을 위해 O0 옵션으로 개발하였으나, 실제 적용 시 O3 옵션 사용을 권장

과제 구현 - Unit Tests

- 1차 과제는 main.c 파일 내부에서 Module_TEST() 함수를 통해 필요한 테스트를 수행하였음
- 본 과제는 test/ subdirectory 내부의 Unit Test 파일들을 통해 필요한 테스트를 수행함
- 해당 파일들은 Makefile을 통해 관리됨
- 주요 Unit Test 기능은 별도로 설명

과제 구현 - 버전 관리

History for `thres_eddsa / sign.c`

Commit Message	Author	Time	View	Diff
Comments on Feb 3, 2020				
Cleanup	woojung3	commented 2 hours ago	8c9420	D
Clean up	woojung3	commented 2 hours ago	9ab50ac	D
Optimizing	woojung3	commented 8 hours ago	fc1404c	D
Comments on Jan 31, 2020				
change num_players to plen	woojung3	commented 2 days ago	781a249	D
Draft done.	woojung3	commented 3 days ago	8b4950c	D
Comments on Jan 26, 2020				
Working on 4.2 eq 3.	woojung3	commented 6 days ago	2729023	D
Comments on Jan 24, 2020				
- thres_eddsa done (except for eq. 3)	woojung3	commented 10 days ago	e30c1d3	D
- Debugging sign.c	woojung3	commented 10 days ago	874e77c	D
Comments on Jan 23, 2020				
- thres_vss bug fixed	woojung3	commented 12 days ago	320a333	D
Comments on Jan 22, 2020				
- Working on dist_vss bugs	woojung3	commented 12 days ago	36c6a30	D
Comments on Jan 15, 2020				
- Working on hidman_vss bug	woojung3	commented 19 days ago	0566c91	D

Git를 통해 버전 관리 수행

Table of contents

- 1** 연구 범위 및 내용
 - 연구 범위
 - 연구 내용

- 2** 1차 과제 대비 변경
 - ECDSA vs EdDSA
 - 최적화, 보안 위협, 기타 수정

- 3** API 사용 및 주의사항
 - API 사용
 - 주의사항

- 4** Unit Tests

- 5** 참고문헌

API 사용

개발된 Threshold EdDSA의 사용법을 기술함

- 본 과제는 Network를 통한 Local 연산과 연산 결과 공유는 추가적으로 진행될 것으로 가정함⁹
- 이하에서는 현재 개발된 Threshold EdDSA의 사용법을 설명하고,
- 이후 코드를 추가할 때 주의해야 할 사항을 별도 기술함

Threshold EdDSA 사용을 위한 API는 다음 파일에서 관리됨

- thres_eddsa.h (헤더)
- keygen.c
- sign.c
- verify.c

⁹1차 과제 [3]와 동일한 사양

keygen.c

```
ERT crypto_sign_keypair(keys **partial_keys , VSS_scheme *scheme, BYTE *pk, int n, int t)
{
    if (FAIL == feldman_vss(pk, partial_keys , scheme, n, t))
        return FAIL;

    return SUCCESS;
}
```

Threshold EdDSA에서 Key Generation은 Feldman VSS를 통해 수행됨

- Key generation을 통해 각 플레이어가 지니고 있어야 하는 keys *의 벡터인 keys **partial_keys가 생성됨
- 또한 생성된 각 keys *를 검증할 수 있는 VSS_scheme *scheme이 생성됨(Stinson Random Secret Sharing의 *evidence*에 해당됨)

sign.c

```
ERT crypto_sign_signature(BYTE *sig, keys **partial_keys, VSS_scheme *scheme, const BYTE *pk,
                          const void *m, BWT mlen, int *p_index, int plen,
                          const void *ctx, BYTE ctxlen, BYTE flag);
```

- sign.c는 서명을 생성함. 생성된 서명은 BYTE *sig로 출력됨
- keygen.c를 통해 생성된 partial_keys 중 서명 생성 과정에 참여하는 플레이어들의 partial_keys와 그를 나타내는 index인 p_index, p_index의 길이인 plen이 입력으로 사용됨
- 검증 과정을 위한 evidence외에 공개키 pk, 메시지 m, 메시지 길이 mlen이 사용됨

verify.c

```
ERT crypto_sign_verify(const BYTE *sig, const BYTE *pk, const BYTE *m, BT64 mlen,
                       const void *ctx, BYTE ctxlen, BYTE flag)
{
    return PCIS_ED25519_Verify(sig, pk, m, mlen, ctx, ctxlen, flag);
}
```

Threshold EdDSA에서 Verification은 기존 EdDSA 코드 [6]를 통해 수행됨

본 과제를 통해 신규 개발한 서명 코드가 기존의 검증 코드를 통해 검증되는 것을 통해 코드의 정합성이 확인됨

API 전체 사용 예시

```

printf("-- Threshold EDDSA with n/t = %d/%d (%d) players --\n", n, t, plen);

// BEGIN Key generation
VSS_scheme scheme;

if (SUCCESS != crypto_sign_keypair(partial_keys, &scheme, pk, n, t))
    return FAIL;
// END Key generation

// BEGIN Selecting partial keys
for (int i=0; i<plen; i++)
    keys_Copy(sub_partial_keys[p[i]-1], partial_keys[p[i]-1]);
// END Selecting partial keys

// Signature generation
if (SUCCESS==crypto_sign_signature(sig, sub_partial_keys, &scheme, pk, msg, msg_len, p, plen, ctx, ctxlen, flag))
    printf(COLOR_GREEN "Signature generation OK." COLOR_RESET "\n");
else
{
    printf(COLOR_RED "Signature generation FAILED." COLOR_RESET "\n");
    return FAIL;
}

if (SUCCESS==crypto_sign_verify(sig, pk, msg, msg_len, ctx, ctxlen, flag))
    printf(COLOR_GREEN "Verification OK." COLOR_RESET "\n");
else
{
    printf(COLOR_GREEN "Verification FAILED." COLOR_RESET "\n");
    return FAIL;
}
  
```

위 코드는 keygen.c, sign.c, verify.c를 연속적으로 사용하는 코드임
 전체 코드는 test/thres_eddsa.c에서 확인 가능

주의사항

API 사용에서 언급한 바와 같이 본 과제의 Threshold EdDSA 산출물은 각 플레이어의 Local 연산 결과를 Network를 통해 수합하거나 검증하는 등의 코드는 포함하고 있지 않고, 추가적으로 개발이 진행될 것으로 가정하였음¹⁰

추가될 코드의 형태에 따라 현재 API의 개형도 대폭 변경될 수 있기 때문에 Keygen, Sign의 IN/OUTPUT으로 keys **partial_keys를 사용하여 코드 복잡도를 줄여둔 상태임

코드 추가 시 feldman_tp.c와 sign.c를 특히 주의해서 수정해야 하며, 수정이 필요한 부분에 주석 처리하였음

- 구현 시 TODO: LOCALLY 로 마크된 주석을 참조
- 이하 슬라이드에서 상세 설명

¹⁰이는 1차 과제 [3]와 동일한 사양임

feldman_tp.c I

```
ERT feldman_vss(BYTE *pk, keys **partial_keys, VSS_scheme *scheme, int n, int t)
{
    int p_index[n];
    for (int i=0; i<n; i++)
        p_index[i] = i+1;
    return feldman_vss_subset(pk, partial_keys, scheme, n, t, p_index, n);
}
```

- 구현상의 편의를 위해 `partial_keys`를 OUTPUT으로 사용하였으나,
- 실제 구현 시에는 `partial_keys`의 각 `key`들은 각 플레이어만이 가지고 있어야 함
- `keys`는 `prefix`, `sk`, `u`, `x`, `y`를 내부 값으로 가지며,
- 이 중 외부로 공개되어도 문제가 없는 값은 Point `y` 뿐임
- 이외의 값을 노출시킬 때에는 보안에 문제가 생기지 않는지 반드시 확인해야 함

feldman_tp.c II

```
ERT feldman_vss_subset(BYTE *pk, keys **partial_keys, VSS_scheme *scheme,
                      int n, int t, int *p_index, int plen)
{
    if (plen < t)
        return FAIL;

    VEC_VSS_scheme *vec_scheme;
    VEC_VSS_share *vec_share;
    VSS_shares_user shares[n];
    VEC_KGBM *vec_bcm;
    VEC_KGDM *vec_decom;
    VEC_DLProof *vec_proof;

    vec_scheme = VEC_VSS_scheme_New(n);
    vec_share = VEC_VSS_share_New(n);
    vec_bcm = VEC_KGBM_New(n);
    vec_decom = VEC_KGDM_New(n);
    vec_proof = VEC_DLProof_New(n);
}
```

feldman_tp.c III

- feldman_vss 함수와 기본적으로 같음
- partial_keys 내부의 각 keys의 값 중 u가 채워져 있는 경우 채워져 있는 u를 랜덤 값으로 사용
- 채워져 있지 않은 경우 랜덤 값을 처음부터 뽑아서 사용함
- 이는 키 생성 후 서명 생성 시에 feldman_vss가 특수 값을 사용하기 때문임

feldman_tp.c IV

```
/**
 * TODO: LOCALLY: partial key generation
 * OUTPUT: vec_bcm, vec_decom
 */
for (int i = 0; i < plen; i++)
{
    // partial key generation
    create(partial_keys[p_index[i]-1], t, n, p_index[i]);

    // commitment generation
    BYTE data[32];
    pcis_ED25519_Compress(partial_keys[p_index[i]-1]->y_i, data);
    HC_Commit(vec_bcm->data[p_index[i]-1]->cmt, vec_decom->data[p_index[i]-1], data, 32);
}

// Third-Party: Verifies commitment, new VSS generation, and pk generation.
if (FAIL == com_vrfy_pk_gen(pk, vec_bcm, vec_decom, p_index, plen))
{
    return FAIL;
}
```

feldman_tp.c V

- 각 플레이어 p_i 가 자신의 keys값을 create함수를 통해 생성
- 이후 keys값에 해당하는 commitment를 생성
- vec_bcm 과 vec_decom 이 출력으로 생성되며, 이는 본 구현에서 Third-Party를 통해 검증됨

현재 코드 내부에 들어있는 keys 값 및 commitment 생성부를 각 플레이어 p_i 가 수행하도록 수정해야 하며, Third-party(현재 구현) 혹은 각 플레이어가 $com_vrfy_pk_gen$ 을 통해 검증해야 함

feldman_tp.c VI

```
/**
 * TODO: LOCALLY: VSS schemes/shares generation
 * OUTPUT: vec_scheme, vec_share
 */
for (int i = 0; i < plen; i++)
{
    VSS_share(vec_scheme->data[p_index[i]-1], vec_share->data[p_index[i]-1], t, n,
    partial_keys[p_index[i]-1]->u_i, p_index, plen);
}
```

- 각 플레이어 p_i 가 자신의 keys값에 해당하는 VSS share를 생성
- 모든 플레이어의 VSS share가 `vec_share`에 저장됨 (Third party가 수합하거나 분산 공유)
- `vec_share`를 검증할 수 있는 evidence가 `vec_scheme`에 저장됨

feldman_tp.c VII

```
// Third-Party: shares transfer
for (int i = 0; i < plen; i++)
{
    shares[p_index[i]-1].len = plen;
    shares[p_index[i]-1].party_index = p_index[i];
    for (int j = 0; j < plen; j++)
        MINT_Copy(shares[p_index[i]-1].data + p_index[j]-1,
            vec_share->data[p_index[j]-1]->data + (p_index[i]-1));
}

// Third-Party: Verifies VSS
for (int i = 0; i < plen; i++)
{
    // VSS verification
    if (FAIL == VSS_verify(vec_scheme->data[p_index[i]-1], shares[p_index[i]-1].party_index,
        shares[p_index[i]-1].data+p_index[i]-1))
        return FAIL;
}
```

- Share Transfer와 VSS 검증은 개별 플레이어가 수행하거나, Third-Party가 수합한 뒤 검증을 수행할 수 있음

feldman_tp.c VIII

```
/**
 * TODO: LOCALLY: map shares to private share
 * OUTPUT: None
 */
for (int i = 0; i < plen; i++)
{
    // Private_share x_i generation from shares i.e (sum of x_i's) = (sum of u_i's)
    VSS_shares_to_private_share(&shares[p_index[i]-1], partial_keys[p_index[i]-1], p_index, plen);
}
```

- 각 플레이어 p_i 는 shares transfer과정을 통해 자신에게 분배된 partial data를 합쳐 신규 partial data를 생성
- 이 값이 최종적인 p_i 의 partial data가 됨
- (별도의 출력 값 없이 partial_keys에 저장된 partial data값만이 변경됨)

feldman_tp.c IX

```
/**
 * TODO: LOCALLY: zk proof
 * OUTPUT: vec_proof
 */
for (int i = 0; i < plen; i++)
{
    //DL proof generation using x_i
    DLProof_Prove(vec_proof->data[p_index[i]-1], partial_keys[p_index[i]-1]->x_i);
}

// Third-Party: Verify ZK proof
schemes2scheme(scheme, vec_scheme, p_index, plen);
if (SUCCESS != dlog_proof_vrfy(scheme, vec_proof, p_index, plen))
    return FAIL;
```

- 각 플레이어 p_i 는 ZK proof를 준비함
- 플레이어간 분산 통신 혹은 Third-Party를 통해 ZK proof를 수행함

sign.c

R 생성을 위한 partial r 작성과 생성된 R 을 이용한 partial s 생성이 개별 플레이어에 의해 수행되어야 함

- R 생성 과정:

- $r_partial_keys$ 의 u_i 에 개별 플레이어가 필요한 값을 계산하여 채움
- $feldman_vss$ 를 통해 R 을 생성

- partial s 생성 과정:

- 생성된 R 을 통해 EdDSA 서명 과정을 통해 partial s 를 생성

partial s 는 Third-party에 의해 수합되어 검증되거나, 개별 플레이어에 의해 검증될 수 있음(현재 구현에서는 Third-party가 수합하는 것을 가정함)

Table of contents

- 1 연구 범위 및 내용
 - 연구 범위
 - 연구 내용

- 2 1차 과제 대비 변경
 - ECDSA vs EdDSA
 - 최적화, 보안 위협, 기타 수정

- 3 API 사용 및 주의사항
 - API 사용
 - 주의사항

- 4 Unit Tests**

- 5 참고문헌

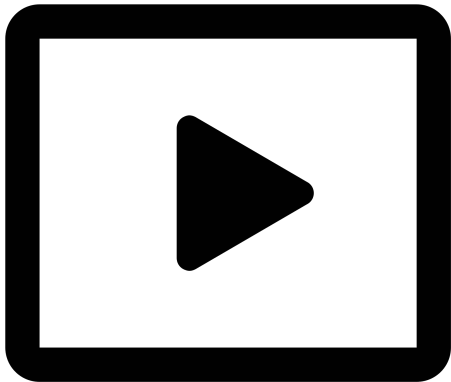
Unit Tests

- 1 Threshold EdDSA
- 2 VSS
- 3 Feldman VSS
- 4 Tools

List

Unit Test는 test/ subdirectory에서
확인 가능.

thres_eddsa, vss, dist_vss, tools,
eddsa, thres_eddsa_keygen



Unit Tests

- 1 Threshold EdDSA
- 2 VSS
- 3 Feldman VSS
- 4 Tools

[vss.c]
8가지 case에 대해 VSS의 작동을
확인하는 Unit Test

Unit Tests

- 1 Threshold EdDSA
- 2 VSS
- 3 **Feldman VSS**
- 4 Tools

[dist_vss.c]

Feldman VSS의 기본 작동 확인

Unit Tests

- 1 Threshold EdDSA
- 2 VSS
- 3 Feldman VSS
- 4 Tools

[tools.c]
Encoding, Decoding, Negate 등
기초 연산 확인

참고문헌 I

 Stinson, D. R., & Strobl, R. (2001, July).


Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates.

In Australasian Conference on Information Security and Privacy (pp. 417-434). Springer, Berlin, Heidelberg.

 Gennaro, R., & Goldfeder, S. (2018, January).




Fast multiparty threshold ECDSA with fast trustless setup.

In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (pp. 1179-1194).

 고려대학교 (2019, July).

Threshold ECDSA scheme 참조 구현 보고서, 2019, 별도공유

참고문헌 II

-  Internet Research Task Force (as of February, 2020).
RFC 8032 - Edwards-Curve Digital Signature Algorithm (EdDSA).
<https://tools.ietf.org/html/rfc8032>
-  Frank Yellin (2014).
ed25519.js, Ed25519 Elliptic Curve
<http://google.github.io/end-to-end/api/source/src/javascript/crypto/e2e/ecc/point/ed25519.js.src.html>
-  CycloneCrypto (as of February, 2020).
CycloneCrypto EdDSA Github.
<https://github.com/Oryx-Embedded/CycloneCrypto>

Q&A

