C++에서 멤버의 접근성을 위반하는 연산
(An Expression Violating the Member Accessibility in C++)

주 성 용 † 조 장 우 ††
(Seongyong Joo) (Jang-Wu Jo)

요 약 본 논문에서는 기존 C++ 컴파일러에서 검출하지 못하는 클래스 멤버의 접근성을 위반하는 연산을 정의한다. C++에서는 접근 지정자로 클래스 멤버의 접근성을 선언하도록 하고 있다. 접근 지정자 중에서 private과 protected는 객체 외부에서 접근할 수 없는 멤버 지정을 위해서 사용된다. 그러나 C++의 포인터 연산을 이용하면 객체 외부에서 private이나 protected 접근성을 가지는 멤버로 직접 접근 가능하다. 본 논문에서는 멤버 접근성을 위반하는 연산의 원인과 사례를 보이고 이를 정형적으로 정의한다. 본 논문의 공헌은 기존에 다루어지지 않은 문제인 멤버의 접근성을 위반하는 연산을 정의하는 것이다.

키워드 : 멤버 접근성, 정적 분석, 소프트웨어 취약점, 소프트웨어 보안, 포인터 분석

Abstract This paper addresses a problem of violating the member accessibility of a class in C++, which is not detected as an error by existing C++ compilers. The member access specifiers can be used to specify member accessibility. The C++ uses a private or protected specifier for specifying the members which cannot be accessed from outside of an object. However, the private or protected members can be accessed from outside of that object by the pointer arithmetic in C++. We show some violating examples that cannot be detected by existing C++ compilers. The contribution of this paper is to discover and define the new problem of the member accessibility.

Key words : Member accessibility, Static analysis, Software vulnerability, Software security, Points-to analysis

1. 서 론

본 연구에서는 C++ 컴파일러에서 검출하지 못하는 소프트웨어의 취약점인 클래스 멤버의 접근성을 위반하는 연산을 처음으로 제안한다. 이 같은 연산을 멤버 접근성 위반 연산이라 하고 멤버 접근성 위반 연산의 형태를 엄밀하게 정의하고자 한다.

C++의 강제 형 변환(type casting)과 포인터 연산을 이용하면 `private`이나 `protected`로 선언된 데이터 영역으로 직접 접근 가능하다. 이 같은 연산은 C++에서 정의한 클래스 멤버 접근성과 클래스 설계자의 의도를 위반하기 때문에 C++ 클래스 설계자가 의도한 프로그램 신뢰도를 보장할 수 없게 되며 이로 인해서 다른 부가적인 문제들을 야기할 수 있다. 본 논문에서는 이 같은 연산들이 허용되는 이유를 설명하고 이런 연산을 정의한다. 그리고 멤버 접근성을 위반하는 연산을 계산하기 위한 기본적인 아이디어를 제안한다.

본 논문의 구성은 다음과 같다. 2절에서 멤버 접근성을 위반하는 연산의 원인과 예를 설명하고, 3절에서 멤버 접근 규칙을 위반하는 연산식을 설명한다. 그리고 4절에서는 멤버 접근성을 위반하는 연산을 계산하는 예를 보인다. 5절에서는 관련연구로 소프트웨어 취약점을 검출하는 기법과 포인터 분석 기법을 기술하고 본 연구와의 관련성을 설명한다. 마지막으로 6절에서는 결론을 맺는다.

2. 멤버의 접근성을 위반하는 연산

C++ 클래스는 `public`과 `private` 그리고 `protected` 세 가지 접근 지정자(access specifier)를 제공한다. pri-

· 이 논문은 2009년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. 2009-0066820)

† 학생회원 : 동아대학교 컴퓨터공학과
jheaven1@donga.ac.kr
†† 종신회원 : 동아대학교 컴퓨터공학과 교수
jwjo@dau.ac.kr

논문접수 : 2009년 11월 18일
심사완료 : 2010년 1월 6일

정보과학회논문지: 소프트웨어 및 응용 제37권 제3호(2010.3)

vate과 protected 접근 지정자는 객체 외부에서 직접 접근할 수 없는 멤버를 지정하고, public은 객체 외부에서 직접 접근할 수 있는 멤버를 지정한다[1].

C++ 프로그램에서 멤버의 접근성을 위반하는 연산이란 private 또는 protected로 선언된 멤버를 객체 외부에서 직접 접근하는 연산이다.

C++ 컴파일러는 객체 멤버의 이름을 기반으로 멤버 접근성 검사를 수행한다. 객체 멤버 이름으로 객체 멤버를 접근하는 형태는 x.y 또는 px->y이다. 여기서 x는 객체변수, px는 객체포인터변수, y는 객체멤버이름이다. 객체 멤버를 주소로 접근하는 경우 C++ 컴파일러는 멤버 접근성 검사를 수행하지 않는다. 그러므로 포인터 연산을 이용하면 객체 외부에서 private이나 protected 접근성을 가지는 멤버로 직접 접근할 수 있다.

그림 1은 멤버의 접근성을 위반하는 연산의 예이다. 그림 1의 9행에서 cls1 타입 객체인 obj의 멤버 x의 값을 int 타입 변수 v에 대입한다. 이 경우 obj의 멤버 x의 값을 읽기 위해서 객체 멤버이름 x로 접근하기 때문에 컴파일러는 멤버 접근성 검사를 수행하고, 멤버 x의 접근성은 private으로 지정되었기 때문에 컴파일러는 9행에서 오류를 보고한다. 10행에서 int* 타입 변수 pv에 obj의 주소를 대입한다. obj의 멤버 x는 obj의 첫 번째 데이터 멤버이기 때문에 obj와 주소가 동일하다. 그러므로 11행과 같이 *pv에 정수 20을 대입하는 것은 private 접근성을 가지는 멤버 x에 값을 쓰는 결과를 낳게 된다. 그러나 11행에서는 객체 멤버 이름이 아니라 주소로 접근하기 때문에 컴파일러는 멤버 접근성 검사를 수행하지 않는다. 그러므로 컴파일러는 아무런 오류도 보고하지 않고, 이 프로그램을 실행하면 private 데이터 멤버 x로 값을 쓰는 것이 허용된다.

3. 멤버 접근성 위반 연산식

3.1 멤버 접근성 위반 연산 정의

멤버 접근성 위반은 역참조 연산(dereferencing)에 의해서 발생한다. 역참조 연산의 형태는 그림 2와 같다. pv는 포인터 변수이고, iexp는 정수식이다.

```
1:  class cls1  {
2:  private:
3:    int x, y;
4:  };
5:    ...
6:
7:  cls1  obj;
8:  int v,  *pv;
9:  v =  obj.x;
10: pv =  (int*)&obj;
11: *pv = 20;
```

그림 1 멤버 접근성 위반 연산의 예

```
*(pv + iexp)
```

그림 2 역참조 연산의 형태

그림 3은 멤버 접근성을 위반하는 연산을 정의하기 위한 규칙들이다. 그림 3에서 type 함수는 주소를 인수로 받아, 그 주소에 할당된 값의 타입을 반환하는 함수이다. 그리고 acc 함수는 주소를 인수로 받아, 그 주소에 할당된 값의 접근성을 반환하는 함수이다. 그리고 ⊥은 멤버 접근성 위반 연산이다. 예를 들어서 그림 1의 11행에서 변수 pv에 type함수를 적용한 type(pv)는 int이다. 그 이유는 pv의 타입이 int*이므로 pv가 가리키는 곳은 obj의 멤버 x가 할당된 주소가 된다. 그리고 acc(pv)는 x의 접근성인 private을 반환한다.

$$\frac{pv:\tau_1 \quad type(pv+iexp):\tau_2 \quad *\tau_1 \neq \tau_2}{*(pv+iexp) \rightarrow \perp} \text{ (규칙 1)}$$

$$\frac{pv:\tau_1 \quad type(pv+iexp):\tau_2 \quad *\tau_1 = \tau_2 \qquad acc(pv+iexp):\alpha \quad \alpha \neq public}{*(pv+iexp) \rightarrow \perp} \text{ (규칙 2)}$$

그림 3 멤버 접근성을 위반하는 연산

그림 3의 (규칙 1)은 포인터 변수 pv를 역참조한 타입과 포인터 변수 pv의 타입과 포인터 식 (pv+iexp)가 가리키는 주소에 할당된 값의 타입이 일치하지 않으면 멤버 접근성을 위반하는 연산으로 정의한다. 그리고 (규칙 2)는 포인터 변수 pv의 타입과 포인터 식 (pv+iexp)이 가리키는 공간에 할당된 값의 타입이 일치하더라도 그 값의 접근성이 public이 아니면 멤버 접근성을 위반하는 연산으로 정의한다.

(규칙 1)이 필요한 이유는 포인터 변수의 타입과 포인터 식의 타입이 일치하지 않는 경우 데이터 오버플로우에 의해서 멤버 접근성 위반이 발생할 수 있기 때문이다. 그림 4는 데이터 오버플로우에 의한 멤버 접근성 위반 연산의 예를 보인 것이다.

그림 4의 10행에 int 타입 포인터 변수 pv에 cls2 타입 객체 obj의 주소를 대입한다. 그리고 11행에서 주소 pv+1에 정수 10을 대입한다. 이 경우 pv+1이 가리키는 공간은 객체 obj의 멤버 y가 할당된 위치이다. 만일 int의 크기가 4byte고 short의 크기가 2byte라면 *(pv+1)에 어떤 값을 대입하면 데이터 오버플로우에 의해서 y뿐만 아니라 z가 할당된 공간에도 값이 쓰여지게 되는데, z의 접근성은 private이기 때문에 이는 멤버 접근성을 위반하는 것이다.

```
1:  class cls2 {
2:  public:
3:    int x;
4:    short y;
5:  private:
6:    short z;
7:  };
8:  ...
9:  cls2   obj;
10: int *pv   = (int *)(&obj);
11: *(pv+1) = 10;
12: *((short*)(pv+1))=10;
13: *((double*)(pv+0))=10;
```

그림 4 데이터 오버플로우에 의한 멤버 접근성 위반 연산

3.2 강제 형 변환을 고려한 멤버 접근성 위반 연산 정의

강제 형 변환을 사용한 역참조 연산의 형태는 그림 5와 같다.

```
*(Type*)(pv + iexp)
```

그림 5 강제 형 변환을 사용한 역참조 연산의 형태

그림 3의 규칙은 강제 형 변환을 고려하지 않는다. 그림 4의 12행의 경우 그림 3의 규칙에 의하면 pv의 타입은 int*이고 (pv+1)이 가리키는 곳에 할당된 값은 obj의 멤버 y이므로 type(pv+1)이 반환하는 타입은 short이다. 그러므로 pv를 역참조한 타입은 int이기 때문에 멤버 접근성을 위반하는 연산이 된다. 그러나 실제 사용되기 전 강제 형 변환에 의해서 short 타입으로 변환된 뒤 사용되기 때문에 12행 문장은 멤버 접근성을 위반하지 않는다. 반대로 13행의 경우 pv를 역참조한 타입은 int이고 (pv+0)이 가리키는 곳은 obj의 첫 번째 데이터 멤버 x가 할당된 곳이므로 type(pv+0)은 int이다. 그러므로 pv를 역참조한 타입은 int이기 때문에 그림 3의 규칙을 적용하면 멤버 접근성을 위반하지 않는다. 그러나 실제 (pv+0)이 가리키는 공간에 접근할 때는 double타입으로 접근하기 때문에 이 공간에 값을 쓰게 되면 obj의 멤버 x뿐만 아니라, y와 z에도 값을 쓰게 되므로 실제로 이 문장은 멤버 접근성을 위반한다. 그러므로 강제 형 변환을 고려하기 위해서 그림 6의 규칙이 필요하다.

$$\frac{\text{type}(pv + iexp):\tau \quad *\tau_C \neq \tau \quad \mathit{fresh}\ \tau_C}{*(\tau_C)(pv + iexp) \rightarrow \perp} \quad \text{(규칙 3)}$$

$$\frac{\text{type}(pv + iexp):\tau \quad *\tau_C = \tau \quad \text{acc}(pv + iexp):\alpha \quad \alpha \neq \text{public} \quad \mathit{fresh}\ \tau_C}{*(\tau_C)(pv + iexp) \rightarrow \perp} \quad \text{(규칙 4)}$$

그림 6 강제 형 변환을 고려한 멤버 접근성을 위반하는 연산

그림 6의 (규칙 3)은 강제 형 변환 타입이 포인터 식이 가리키는 공간에 할당된 값의 타입과 일치하지 않는 경우 멤버 접근성을 위반하는 연산으로 정의한다. (규칙 3)을 그림 4의 13행에 적용해보면 type(pv+0)는 int이고, 강제 형 변환 타입 (double*)의 역참조 타입은 double이다. 두 타입은 다르기 때문에 13행은 멤버 접근성을 위반하는 연산이다. 그리고 (규칙 4)는 포인터 식이 가리키는 위치에 할당된 값의 타입과 강제 형 변환의 타입을 역참조한 타입이 동일하더라도 포인터 식이 가리키는 값의 접근성이 public이 아니면 멤버 접근성을 위반하는 연산으로 정의한다.

3.3 type함수와 acc함수의 수정

포인터 식이 가리키는 위치에 할당된 값의 타입을 한 가지로 결정할 수 없는 경우가 있다. 그림 7은 이 같은 예를 보인 것이다.

```
1:  class cls3 {
2:  public:
3:    int x;
4:  private:
5:    int y;
6:  };
7:  class cls4 {
8:  public:
9:    cls3 mobj1, mobj2;
10: };
11: ...
12: cls4 obj;
```

그림 7 한 주소가 여러 타입으로 계산되는 예

그림 7의 12행에서 객체 obj는 cls4 타입으로 선언되었다. 그리고 주소 &obj와 &(obj.mobj1) 그리고 &(obj.mobj1.x)는 모두 동일하기 때문에 이 위치에 할당된 객체의 타입은 cls4와 cls3 그리고 int로 계산할 수 있다. 그러나 그림 3과 그림 6의 규칙들에서 사용된 type 함수는 어떤 주소가 가리키고 있는 위치에 할당된 값의 타입을 반환하고 acc 함수는 그 값의 접근성을 반환해야 하지만, 이 경우 타입과 가시성을 한 가지로 결정할 수 없다. 이 같은 문제를 해결하기 위해서 type과 acc 함수의 정의를 변경하고 그림 3과 그림 6의 규칙들을 수정한다.

그림 8은 포인터 식이 가리키는 위치에 할당된 값들의 타입이 여러 개로 계산되는 경우를 고려하기 위해서 수정한 멤버 접근성 위반 연산을 정의하기 위한 규칙이다.

그림 8의 규칙들에서 type′ 함수는 포인터 식이 가리키는 위치에 할당된 값들의 타입들의 집합을 반환하는 함수이며, acc′ 함수는 포인터 식과 타입을 인수로 받아서 포인터 식이 가리키는 위치에 할당된 값들 중 인수로 전달받은 타입을 가지는 객체의 접근성을 반환

하는 함수이다.

그림 8의 (규칙 1)은 type′ 함수가 반환하는 타입 집합 중 포인터 변수 pv의 역참조 타입이 존재하지 않는다면 멤버 접근성을 위반하는 연산으로 정의한다. (규칙 2)는 type′이 반환하는 타입 집합 중 포인터 변수 pv의 역참조 타입이 존재하더라도, 포인터 식이 가리키는 위치에 할당된 객체들 중 포인터 변수 pv의 역참조 타입과 일치하는 값의 접근성이 public이 아니라면 멤버 접근성을 위반하는 연산으로 정의한다. (규칙 3)과 (규칙 4)는 강제 형 변환과 관련된 규칙이다. (규칙 3)은 type′이 반환하는 타입의 집합 중에 강제 형 변환 타입의 역참조 타입이 존재하지 않는다면 멤버 접근성을 위반하는 연산으로 정의한다. 그리고 (규칙 4)는 type′이 반환하는 타입 집합 중 강제 형 변환 타입의 역참조 타입이 존재하더라도, 포인터 식이 가리키는 위치에 할당된 값들 중 강제 형 변환 타입의 역참조 타입과 일치하는 값의 접근성이 public이 아니라면 멤버 접근성을 위반하는 연산으로 정의한다.

$$\frac{pv{:}\tau \quad type'(pv+iexp){:}\tau^{Q} \quad *\tau \notin \tau^{Q}}{*(pv+iexp) \rightarrow \perp} \quad (규칙1')$$

$$\frac{pv{:}\tau \quad type'(pv+iexp){:}\tau^{Q} \quad *\tau \in \tau^{Q} \quad acc'(pv+iexp,*\tau){:}\alpha \quad \alpha \neq public}{*(pv+iexp) \rightarrow \perp} \quad (규칙2')$$

$$\frac{type'(pv+iexp){:}\tau^{Q} \quad *\tau_C \notin \tau^{Q} \quad fresh\ \tau_C}{*(\tau_C)(pv+iexp) \rightarrow \perp} \quad (규칙3')$$

$$\frac{type'(pv+iexp){:}\tau^{Q} \quad *\tau_C \in \tau^{Q} \quad acc'(pv+iexp,*\tau_C){:}\alpha \quad \alpha \neq public \quad fresh\ \tau_C}{*(\tau_C)(pv+iexp) \rightarrow \perp} \quad (규칙4')$$

그림 8 수정된 멤버 접근성 위반 연산 검출 규칙

4. type′ 함수와 acc′ 함수의 계산

그림 9는 type′ 함수와 acc′ 함수의 계산 예를 보이기 위한 코드이다.

그림 10은 cls5 타입 객체의 메모리 배치이다. 주소는 상대주소를 표현한 것이다. 각 주소에는 그 주소에서 시작하는 값들의 타입과 접근성을 표현한다. 주소 0번지는 cls4 객체와 그 객체의 첫 번째 멤버 변수인 변수 a가 위치한다. 주소 2번지의 경우 그 주소에서 시작하는 멤버가 존재하지 않기 때문에 그 주소 공간에 위치하는 값의 타입을 알 수 없다. 이 같은 경우 타입을 알 수 없음을 명시적으로 표현하기 위해서 2행과 4행과 같이 타입 자리에 **UD**를 기술한다. 그러나 2번지의 경우 그 주소 공간 자체는 public 영역에 속하므로 접근성

```
1:  class cls5 {
2:  public:
3:      char a;
4:      short s;
5:  private:
6:      int y;
7:  };
8:  …
9:    cls5 obj;
10:   char   *pc = (char*)&obj;
11:   short  *ps = (short*)&obj;
12:   int *pi = (int*)(pc+3);
13:   *(ps+1)   = 20;
14:   *(pi+0) = 20;
15:  …
```

그림 9 상대 주소를 계산하기 위한 예

상대주소	객체	
0	{cls5, pub}	{char, pub}
1		{short, pub}
2		{UD, pub}
3		{int, pub}
4		{UD, pub}
5		…
7		

그림 10 cls5 객체의 메모리 배치

은 pub로 기술한다.

그림 9의 13행에서 short 타입의 크기가 2byte라면 (ps+1)이 가리키는 곳은 주소 2번지가 된다. 그러므로 type′(ps+1)은 주소 2번지에 할당된 값들의 타입인 {UD}를 반환한다. pv의 타입은 short*이기 때문에 short ∉ {UD}이다. 그러므로 12행의 문장은 (규칙1')에 의해서 멤버 접근성 위반 연산이다. 타입이 일치하지 않는 영역에 값을 쓰는 경우 데이터 오버플로우에 의해서 private이나 protected 영역에 값을 쓸 수 있기 때문에 이 같은 경우 멤버 접근성 위반 연산으로 정의한다.

그림 9의 14행에서 (pi+0)가 가리키는 위치는 주소 3번지이므로, type(pi+0)′이 반환하는 값은 {int}이다. 그러므로 int ∉ {int}이므로 (규칙 1')을 만족한다. 그러나, acc{(pi+0), int}가 반환하는 값은 주소 3번지에 할당된 멤버 중 int 타입을 가지는 멤버의 접근성이므로 {pri}가 된다. 그러므로 14행의 연산은 (규칙 2')에 의해서 멤버 접근성을 위반하는 연산이다.

메모리 정렬(memory alignment) 방법은 컴파일러나 시스템에 따라서 달라질 수 있으나, 분석기 구현 시 각 환경에 적합한 형태로 객체를 표현하면 type′과 acc′ 함수의 수정 없이 제안하는 기법을 적용할 수 있다.

5. 관련연구

C나 C++과 같이 포인터 연산을 허용하는 언어들은 많은 취약점을 가지고 있는 것으로 알려져 있다[2]. 소프트웨어 취약점을 찾기 위한 기존 연구로 타입 한정자(type qualifier)를 이용한 방법이 있고[3-5], 포인터 분석을 위한 방법으로 타입 기반 포인터 분석 방법[6]과 BDD(Binary Decision Diagram)를 이용한 방법[7]이 있다.

타입 한정자를 이용한 방법은 취약점 검출을 위해서 선언된 타입 한정자를 기반으로 제약식을 생성하고, 그 제약식을 만족하는 해를 구한다. 만일 만족하는 해가 존재하지 않는다면 오류로 보고한다. 타입 한정자를 이용하는 방법은 검사를 위해서 취약점 검출을 위한 타입 한정자를 미리 선언해야 한다. 그리고 타입 한정자의 경우는 어떤 변수에 지정할 수 있는 속성이 결정된 경우에만 이용할 수 있다는 단점이 있다. 그러나 C++에서 포인터 변수는 강제 형 변환을 이용하면 모든 타입의 객체를 가리킬 수 있기 때문에 본 논문에서 제안한 문제에 이를 적용하기 어렵다.

포인터 분석을 위해서 타입을 이용하는 방법은 저장공간 모형(storage model)을 이용해서 변수들을 표현하는데, 저장공간 모형은 저장공간 형태 그래프로(SSG)로 표현된다. SSG에서 노드는 메모리 위치를 의미하고 에지는 points-to 관계를 표현한다. 이 방법은 흐름 독립적(flow insensitive) 인 방법을 이용해서 포인터를 분석한다.

BDD를 이용한 방법은 포인터 분석의 한 종류로서 타입을 이용한 방법의 약점인 구조체들의 필드를 다룰 수 있다. 이 방법은 포인터가 실행 시간에 가리킬 수 있는 객체들을 찾기 위한 방법이지만, 본 연구에서는 포인터가 가리킬 수 있는 객체가 아니라 포인터 변수가 접근하는 객체와 그 객체 멤버 접근성을 다루어야 한다는 점에서 다르다.

6. 결 론

멤버 접근성을 위반하는 연산은 보호되어야 할 정보를 외부로 유출하거나 정의되지 않은 방법으로 데이터가 변경될 수 있기 때문에 의도한 프로그램의 신뢰도를 보장할 수 없다. 이 같은 문제를 해결하기 위해서 멤버 접근성 위반 연산의 형태를 엄밀하게 정의하였다.

본 논문의 기여는 기존에 다루어지지 않은 문제인 멤버 접근성을 위반하는 연산을 발견하고 정의한 것이다. 그리고 이를 검출하기 위한 간단한 방법을 제안한 것이다.

향후 연구에서는 접근성을 위반하는 연산을 정적으로 검출하기 위한 방법을 검증하고 실용적인 분석기를 구현할 계획이다.

참 고 문 헌

[1] Working Draft, Standard for Programming Language C++, http://www.openstd.org/jtc1/sc22/ wg21/, p.233.

[2] A. I. SOTIROV, Automatic vulnerability detection using static source code analysis; Final thesis Department of Computer Science in the Graduate School of University of Alabama, April 2007.

[3] Jeffrey S. Foster, M.Fähndrich, and A.Aiken, Flow-Insensitive Points-to Analysis with Term and Set Constraints, Technical Report UCB CSD-97-964, U. of California, Berkeley, August 1997.

[4] Jeffrey S. Foster, Robert Johnsson, John Kodumal, and Alex Aiken, Flow-Insensitive Type Qualifiers, ACM Transactions on Programming Languages and Systems (TOPLAS), vol.28, no.6, 1035-1087, November 2006.

[5] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities, Networking and Distributed System Security Symposium (NDSS), San Diego, California, February 2000.

[6] B. Steensgaard. Points-to analysis in alomost linear time, In Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages, pp.32-41, 1996.

[7] M. Berndl, O. Lhotak, F. Qian, L. Hendren and N. Umanee, Points-to Analysis using BDDs, PLDI 2003, June 2003.

주 성 용

2000년 동아대학교 컴퓨터공학과 졸업(공학사). 2003년 동아대학교 컴퓨터공학과 졸업(공학석사). 2003년~현재 동대학교 전기전자 및 컴퓨터공학부 박사과정 관심분야는 프로그래밍 언어, 컴파일러, 임베디드 시스템 등

조 장 우

1992년 서울대학교 계산통계학과(학사) 1994년 서울대학교 전산과학과(석사). 2003년 한국과학기술원 전산학과(박사). 현재 동아대학교 컴퓨터공학과 교수. 관심분야는 프로그램 분석, 임베디드 시스템