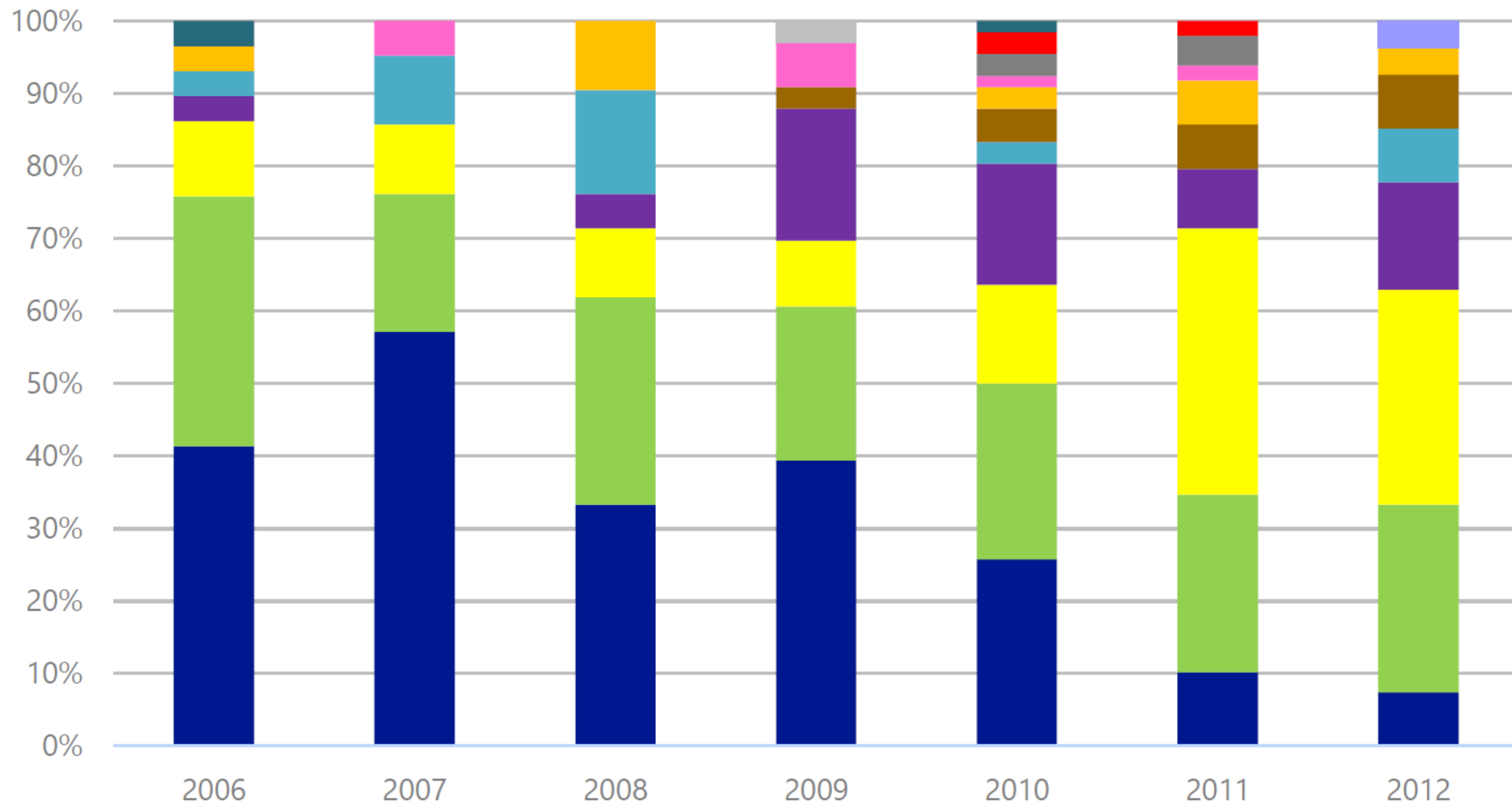
Type Casting Verification: Stopping an Emerging Attack Vector

Byoungyoung Lee, Chengyu Song,
Taesoo Kim, and Wenke Lee

Georgia Institute of Technology

Vulnerability Trends



Microsoft vulnerability trends (2013)

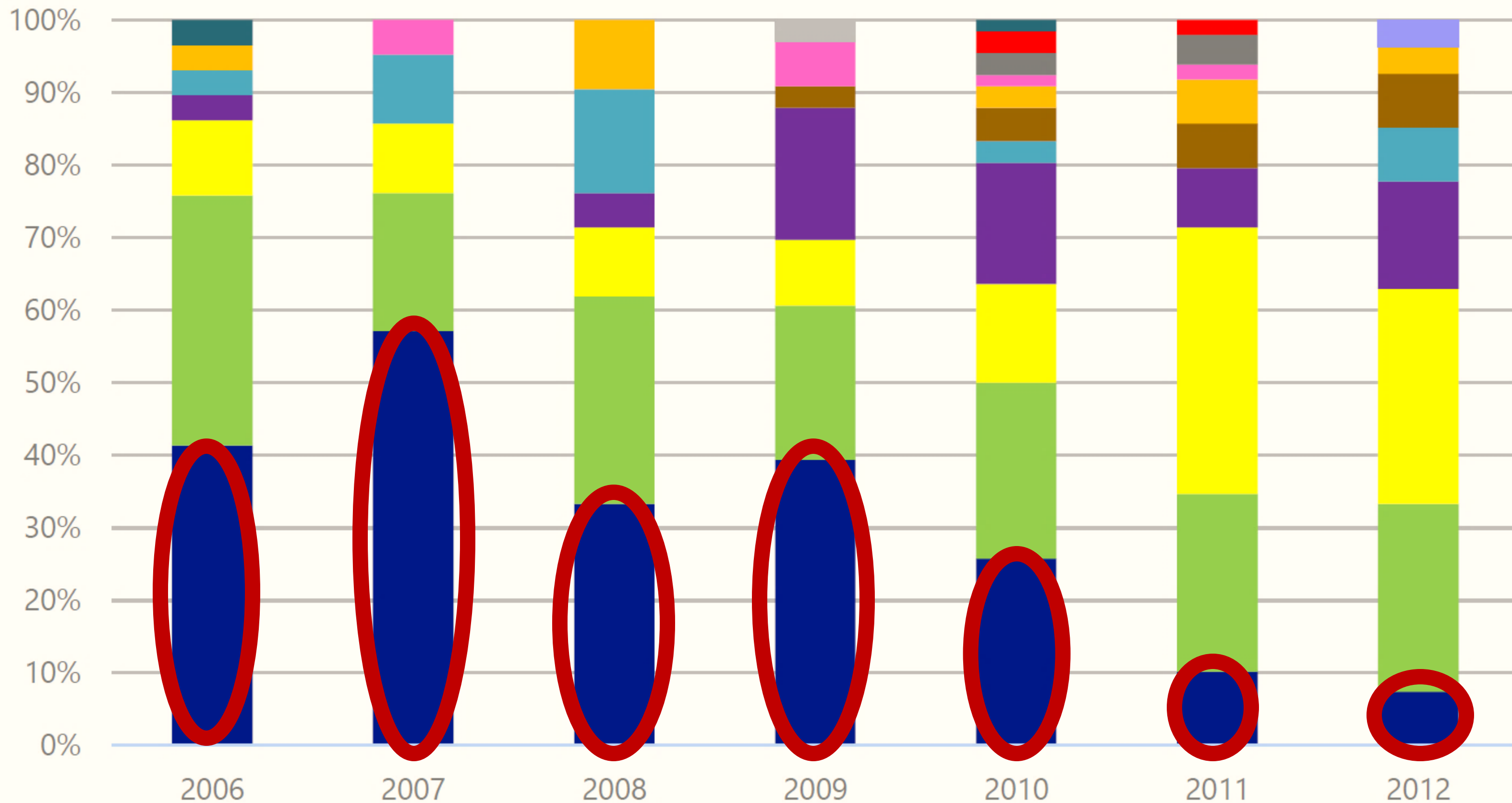
Stack overflow

Use-after-free

Heap overflow

Bad casting (or type confusion)

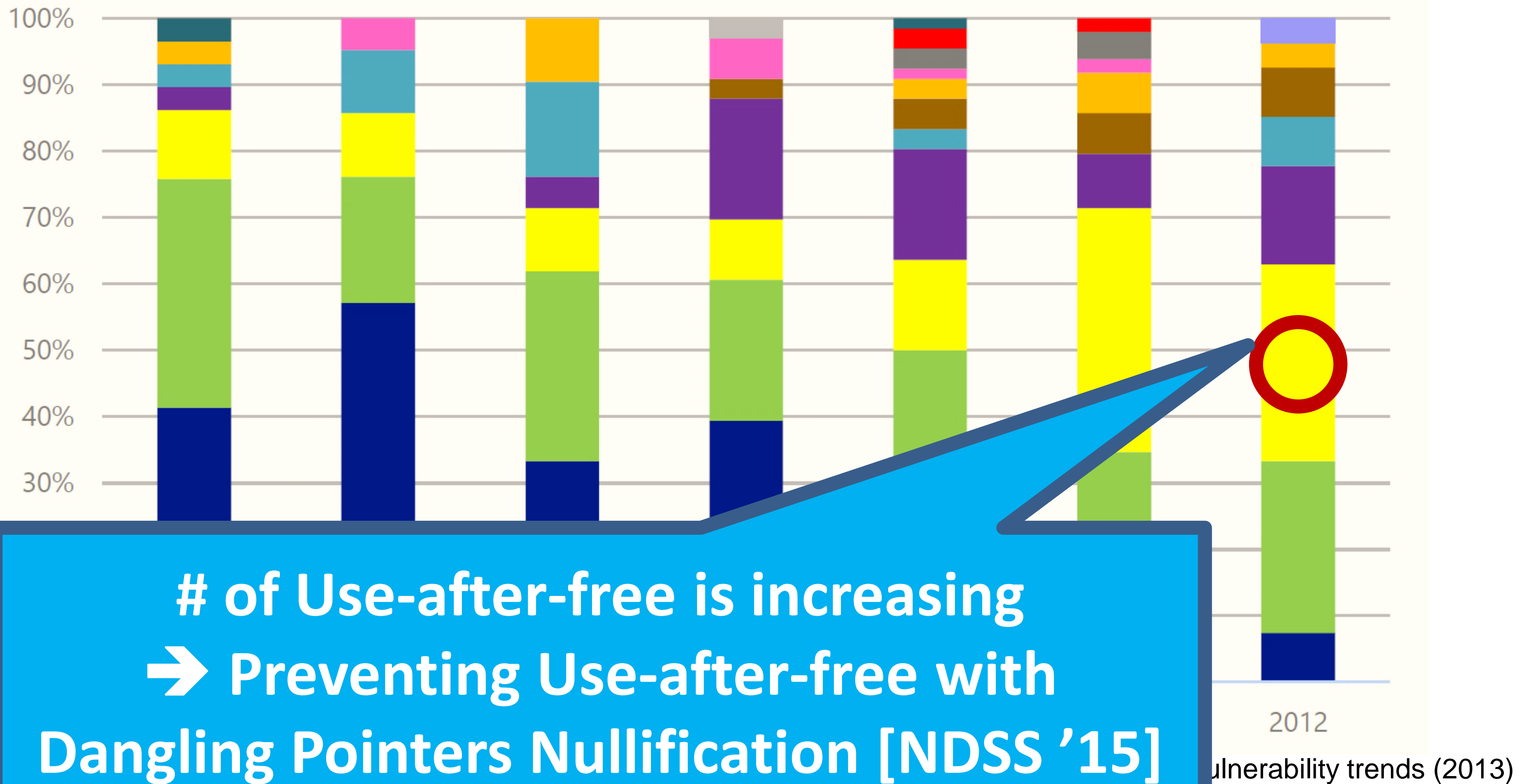
Stack Overflows



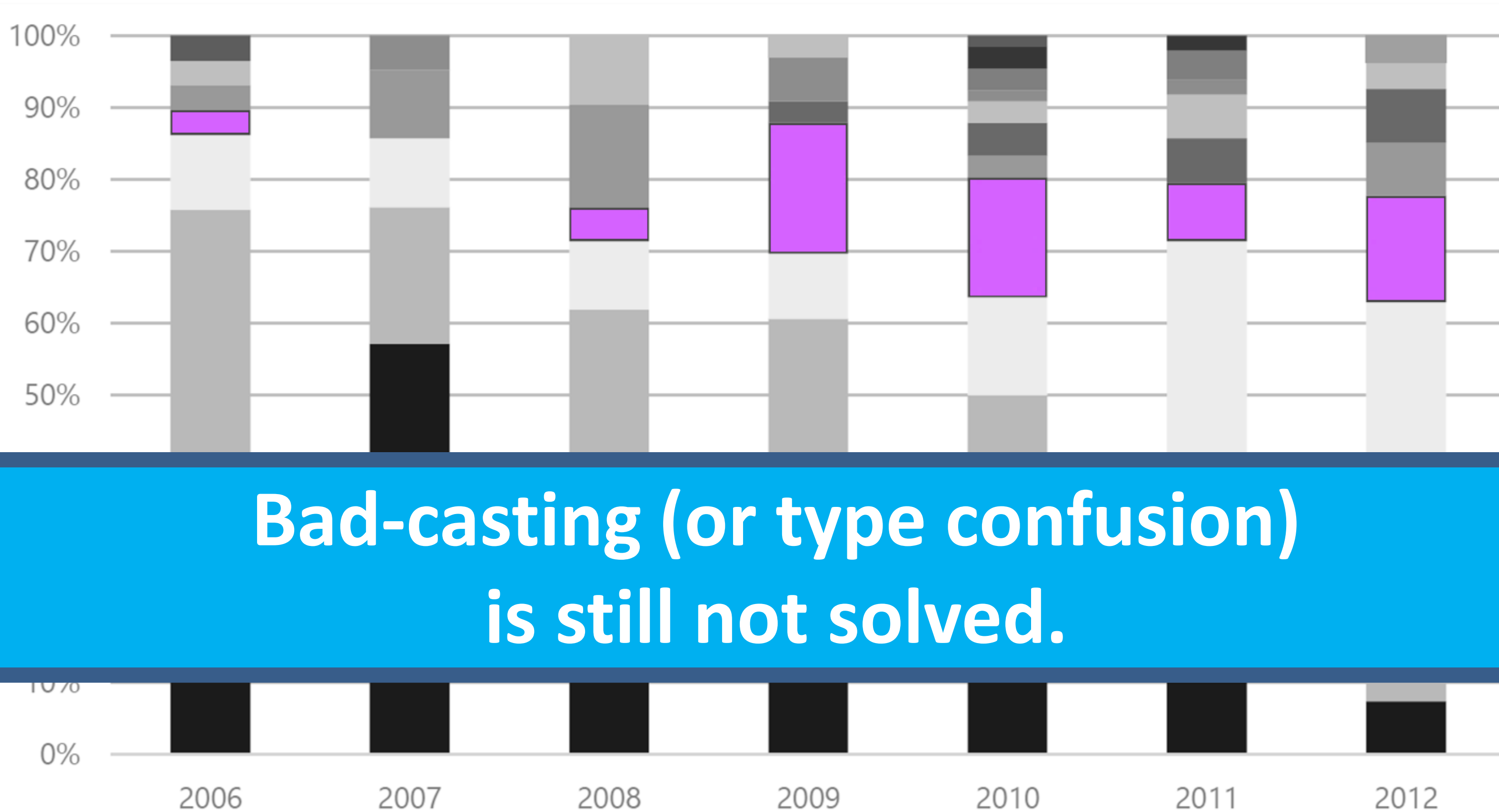
Microsoft vulnerability trends (2013)

# of Stack overflows is decreasing

Use-After-Free



Bad-casting



Bad-casting (or type confusion)
is still not solved.

Type Conversions in C++

- **static_cast**
  - Compile-time conversions
  - **Fast**: no extra verification in run-time
  - No information on actually allocated types in runtime.

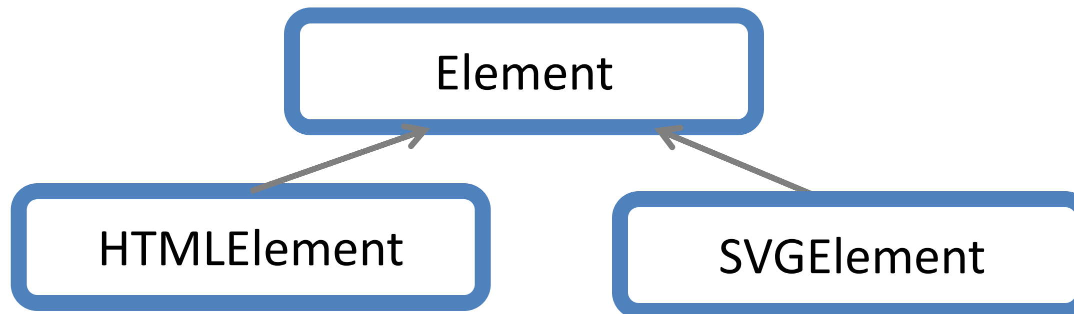
- **dynamic_cast**
  - Run-time conversions
  - Requires **Runtime Type Information (RTTI)**
  - **Slow**: Extra verification by parsing RTTI
  - Typically **prohibited in performance critical applications**

Upcasting and Downcasting

- **Upcasting**
  - From a derived class to its parent class
- **Downcasting**
  - From a parent class to one of its derived classes

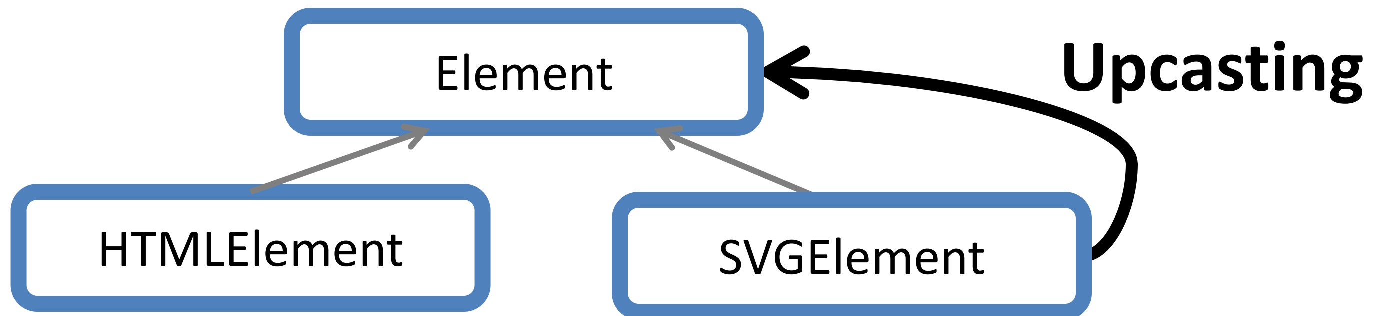
Upcasting and Downcasting

- **Upcasting**
  - From a derived class to its parent class
- **Downcasting**
  - From a parent class to one of its derived classes



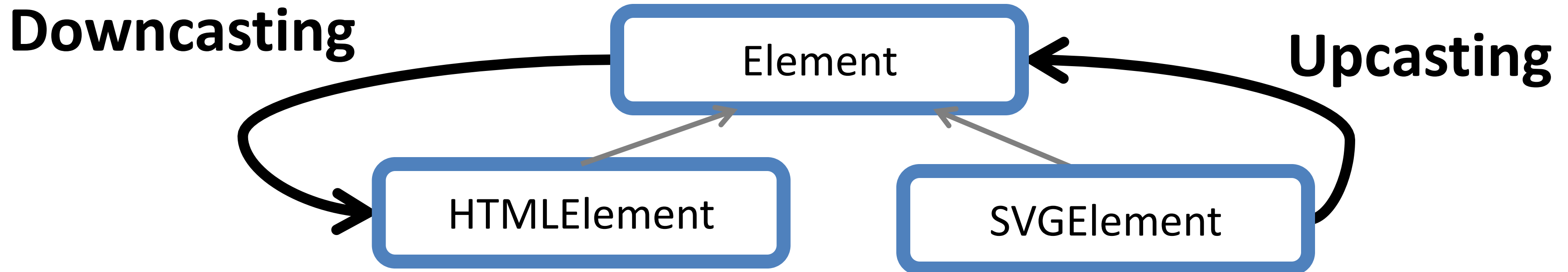
Upcasting and Downcasting

- **Upcasting**
  - From a derived class to its parent class
- **Downcasting**
  - From a parent class to one of its derived classes



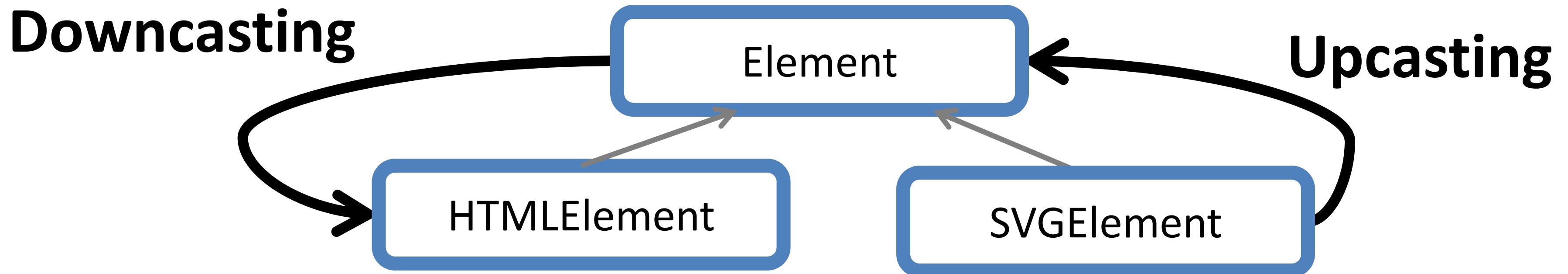
Upcasting and Downcasting

- **Upcasting**
  - From a derived class to its parent class
- **Downcasting**
  - From a parent class to one of its derived classes



Upcasting and Downcasting

- **Upcasting**
  - From a derived class to its parent class
- **Downcasting**
  - From a parent class to one of its derived classes



Upcasting is always safe,
but downcasting is not!

Downcasting is not always safe!

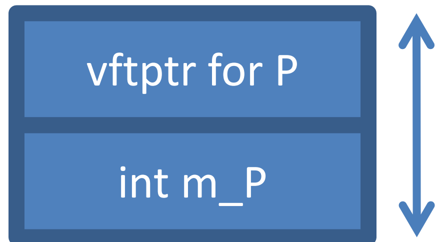
```
class P {
    virtual ~P() {}
    int m_P;
};
```

```
class D: public P {
    virtual ~D() {}
    int m_D;
};
```

Downcasting is not always safe!

```
class P {
    virtual ~P() {}
    int m_P;
};
```

```
class D: public P {
    virtual ~D() {}
    int m_D;
};
```

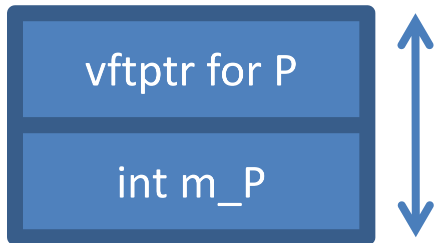


Access scope of P*

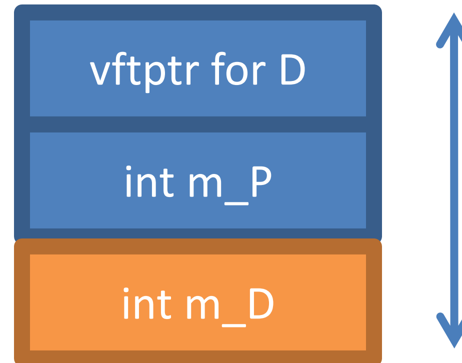
Downcasting is not always safe!

```
class P {
    virtual ~P() {}
    int m_P;
};
```

```
class D: public P {
    virtual ~D() {}
    int m_D;
};
```



Access scope of P*



Access scope of D*

Downcasting can be Bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

Downcasting can be Bad-casting

Bad-casting occurs: D is not a sub-object of P
➔ Undefined behavior

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

Downcasting can be Bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

Memory corruptions

Downcasting can be Bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

Memory corruptions

vftptr for P

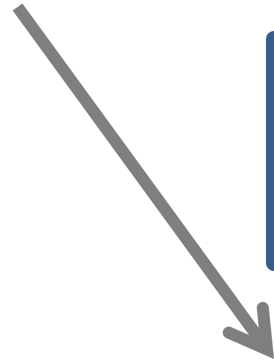
int m_P

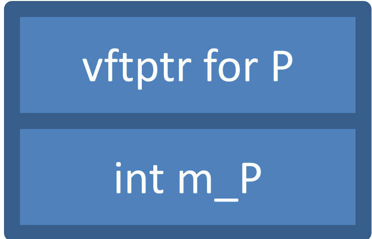
Downcasting can be Bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

Memory corruptions

&(pD->m_D)

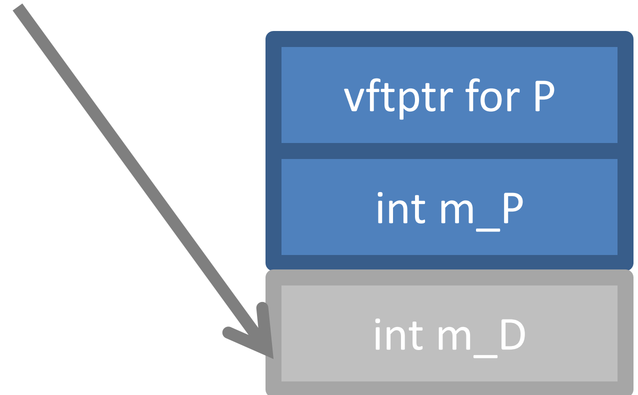




Downcasting can be Bad-casting

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

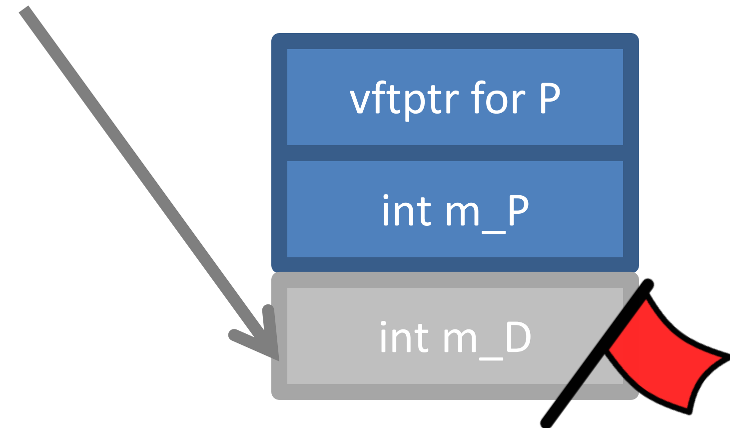
Memory corruptions



Downcasting can be Bad-casting

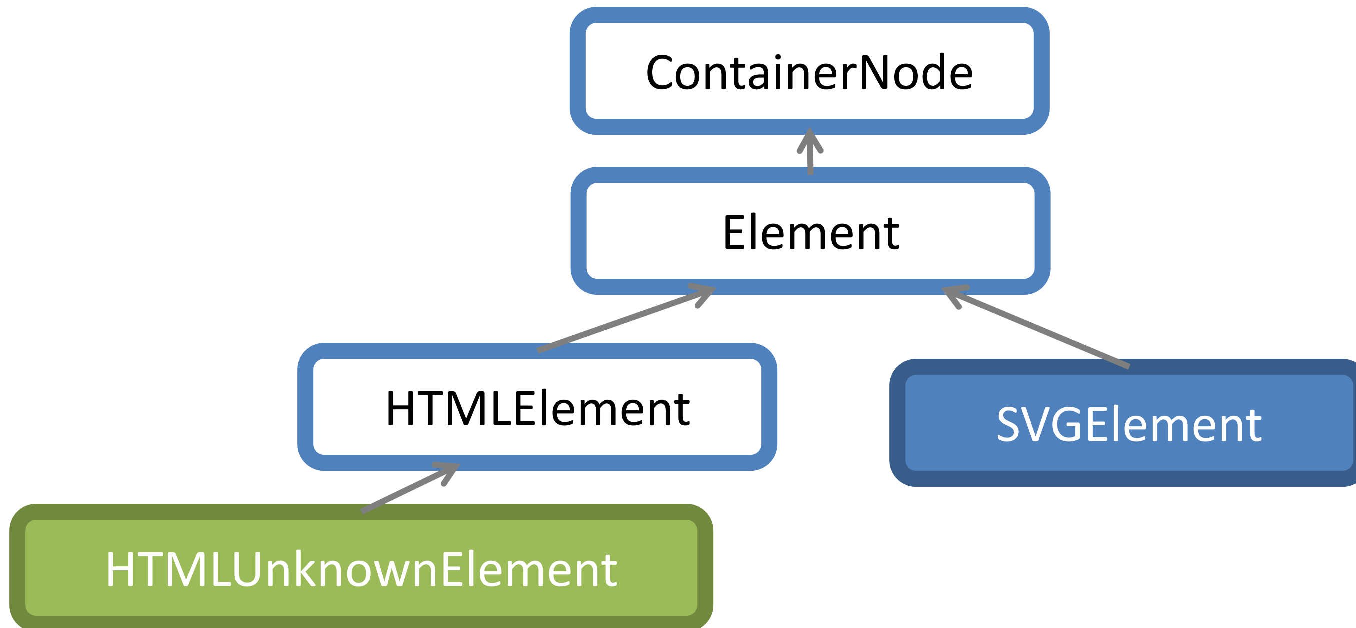
```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D;
```

Memory corruptions



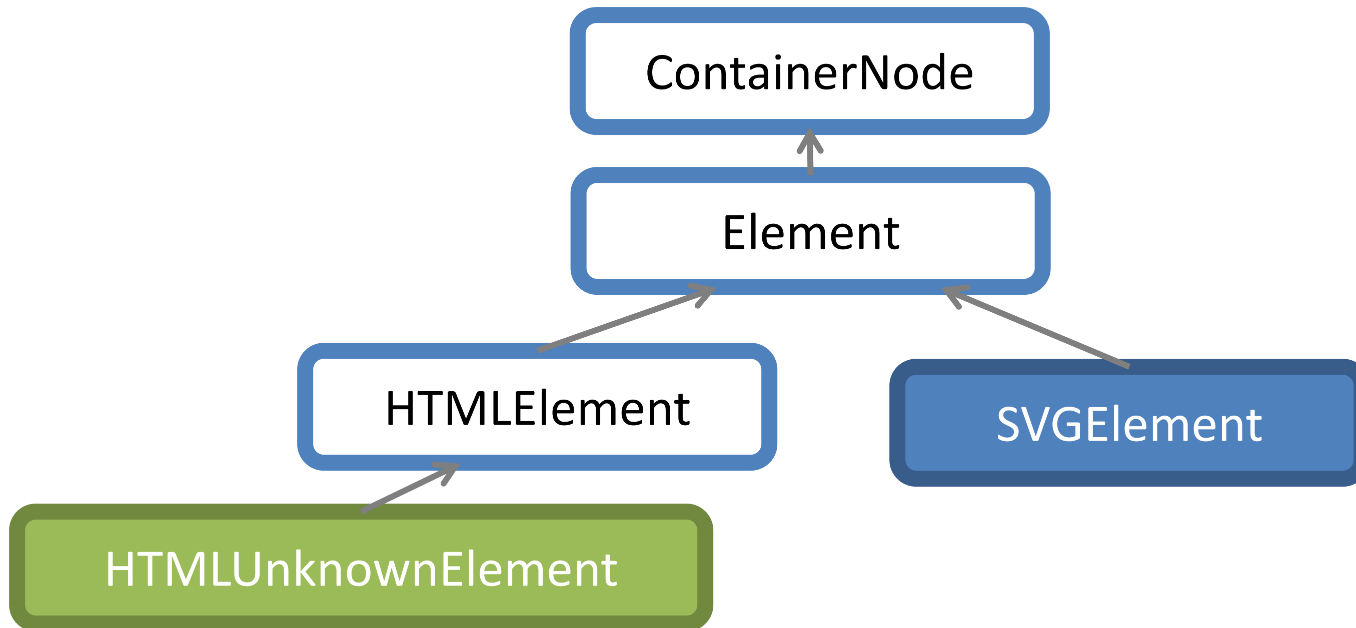
Real-world Exploits on Bad-casting

- **CVE-2013-0912**
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own



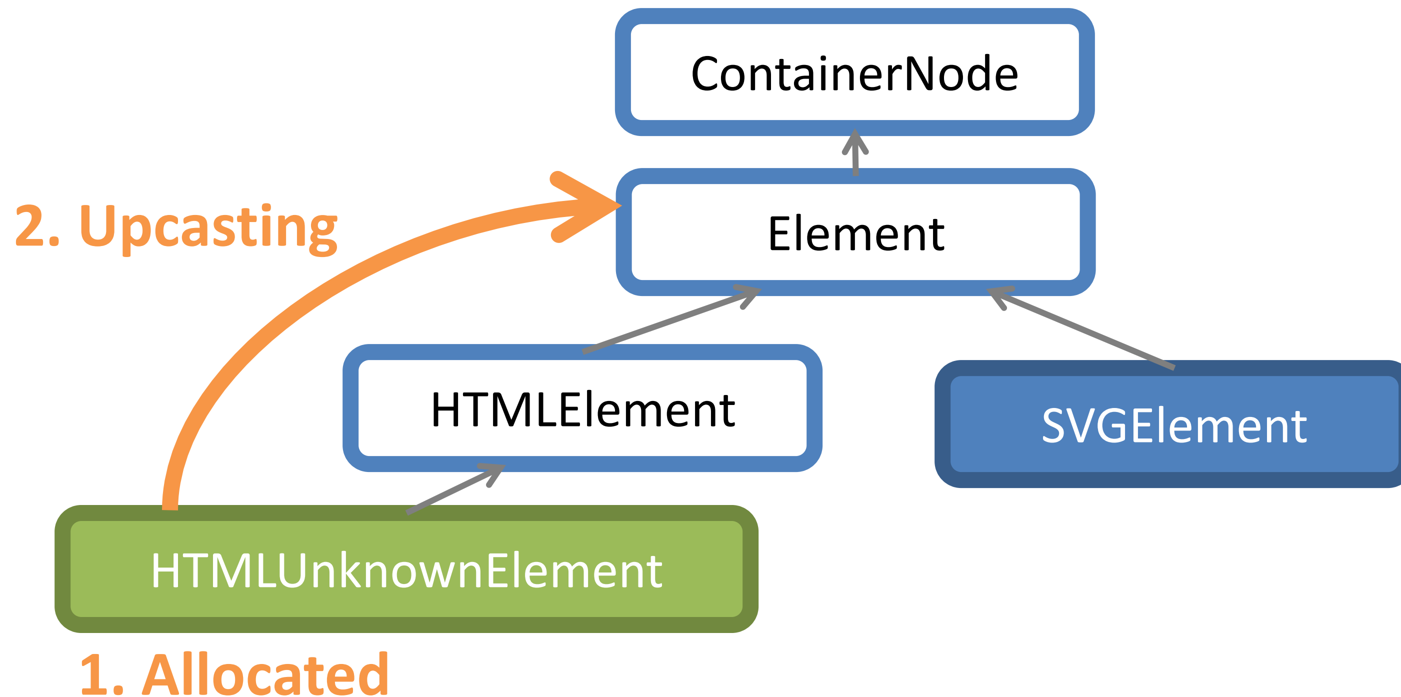
Real-world Exploits on Bad-casting

- **CVE-2013-0912**
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own



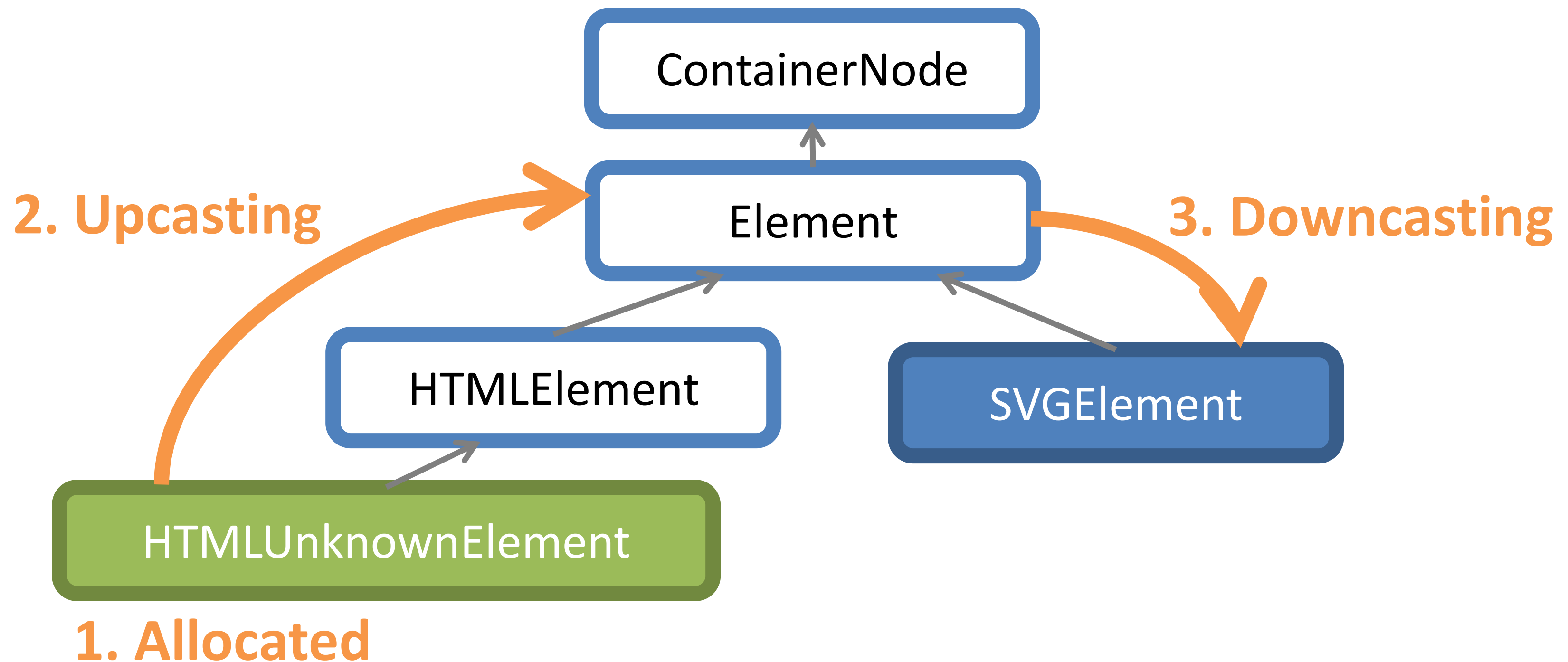
Real-world Exploits on Bad-casting

- **CVE-2013-0912**
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own



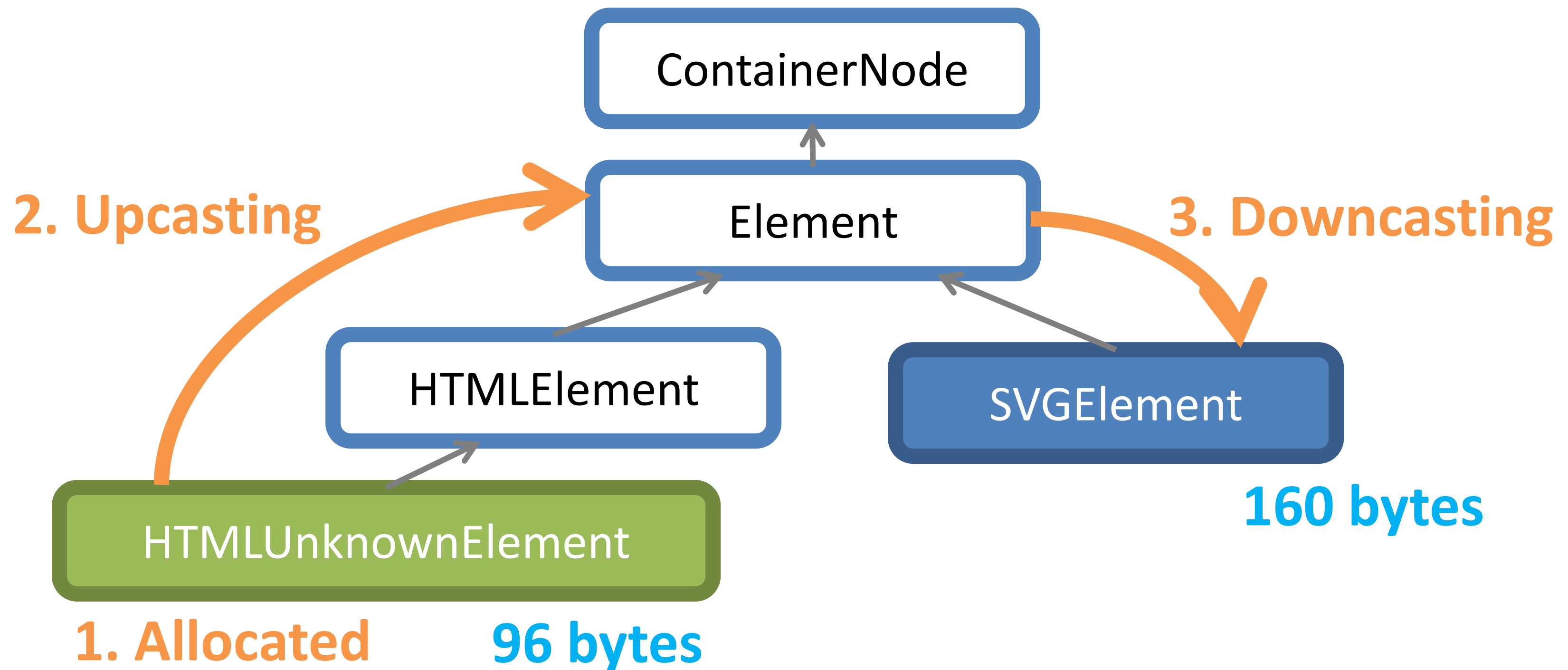
Real-world Exploits on Bad-casting

- **CVE-2013-0912**
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own

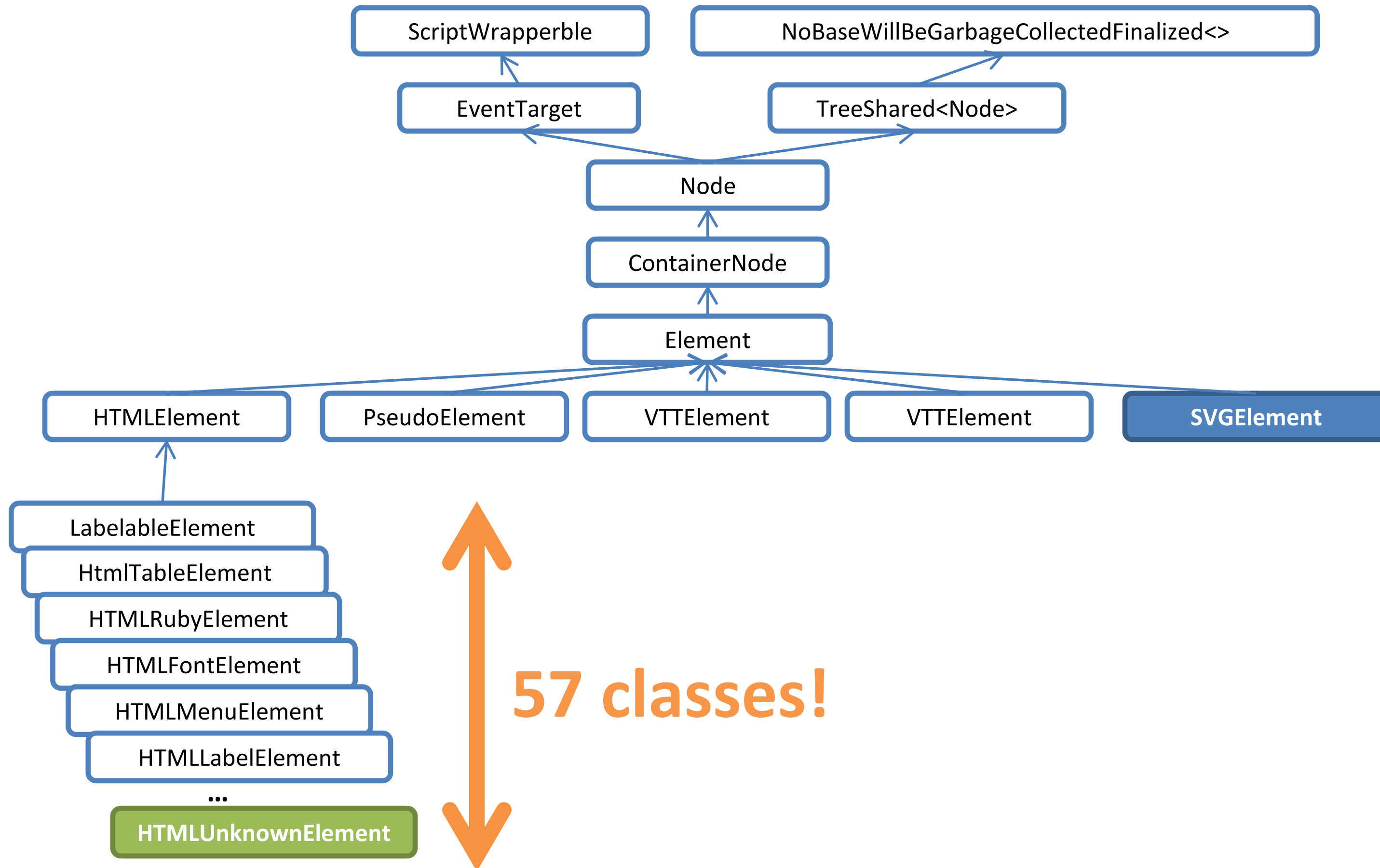


Real-world Exploits on Bad-casting

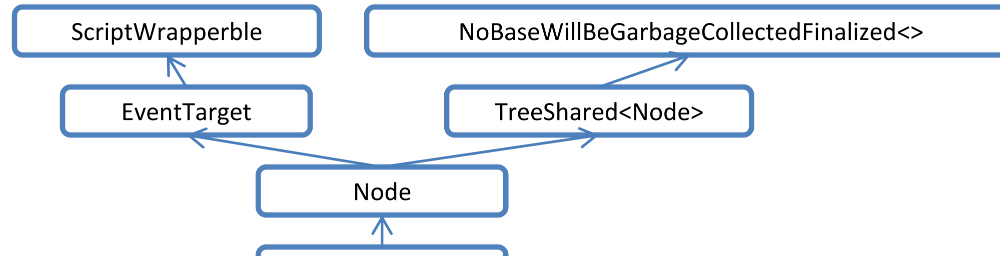
- **CVE-2013-0912**
  - A bad-casting vulnerability in Chrome
  - Used in 2013 Pwn2Own



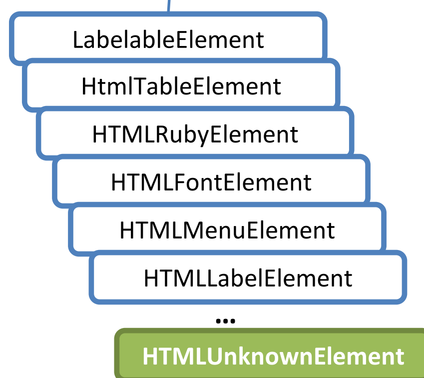
Real-world Exploits on Bad-casting



Real-world Exploits on Bad-casting



Very complex class hierarchies
➔ Error-prone type casting operations

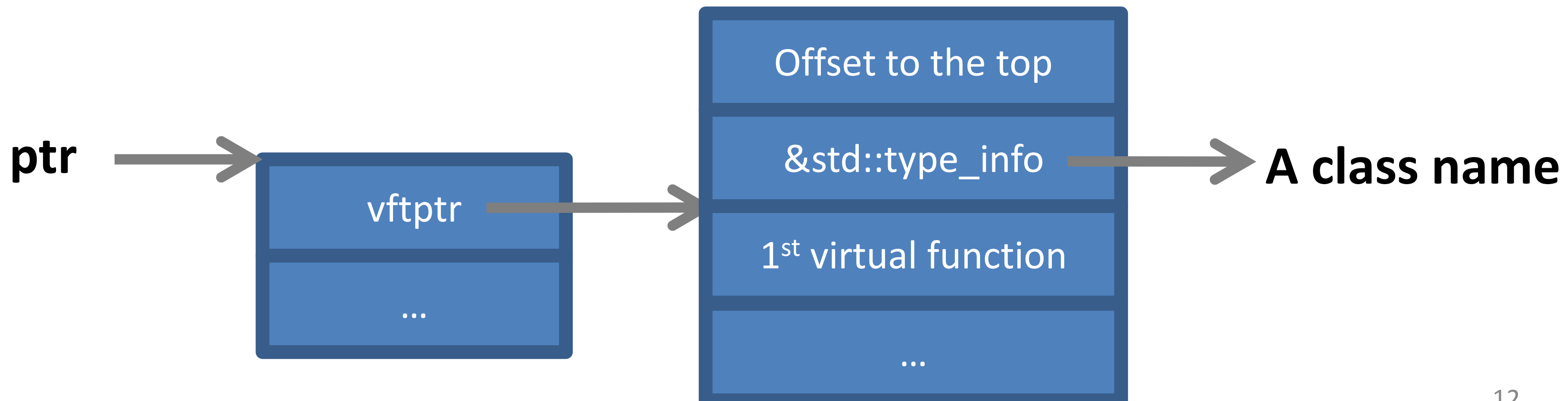


57 classes!

Existing Solutions and Challenges

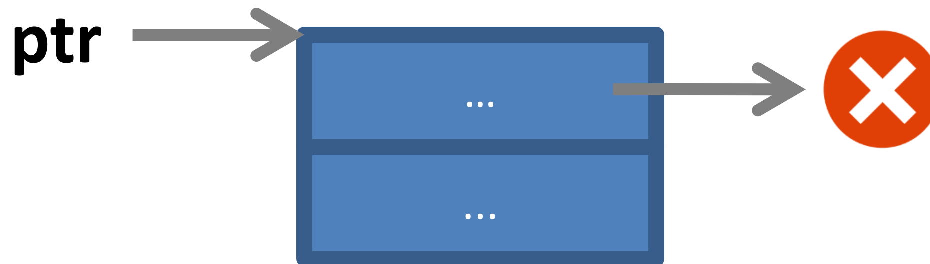
- Replace all static_cast into dynamic_cast

- **dynamic_cast** on a **polymorphic** class (with RTTI)
  - A pointer points to a virtual function table pointer
  - Traversing a virtual function table leads to RTTI



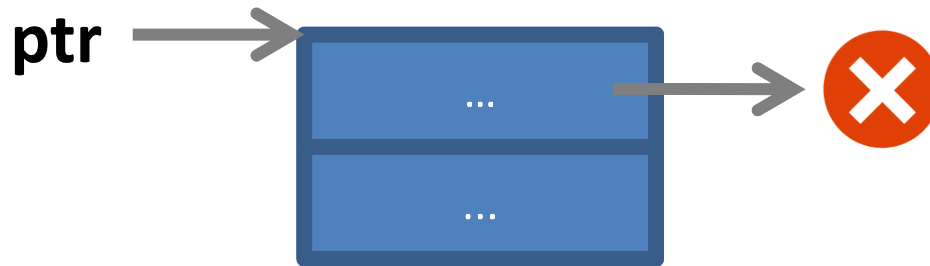
Existing Solutions and Challenges

- **dynamic_cast** on a **non-polymorphic** class
  - A pointer points to the first member variable
  - Simply traversing such a variable leads to **a runtime crash**



Existing Solutions and Challenges

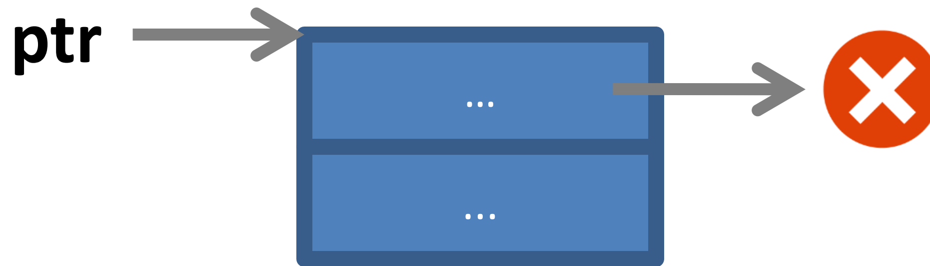
- **dynamic_cast** on a non-polymorphic class
  - A pointer points to the first member variable
  - Simply traversing such a variable leads to **a runtime crash**



C++ supports no reliable methods to resolve whether a pointer points to polymorphic or non-polymorphic classes.

Existing Solutions and Challenges

- **dynamic_cast** on a non-polymorphic class
  - A pointer points to the first member variable
  - Simply traversing such a variable leads to **a runtime crash**



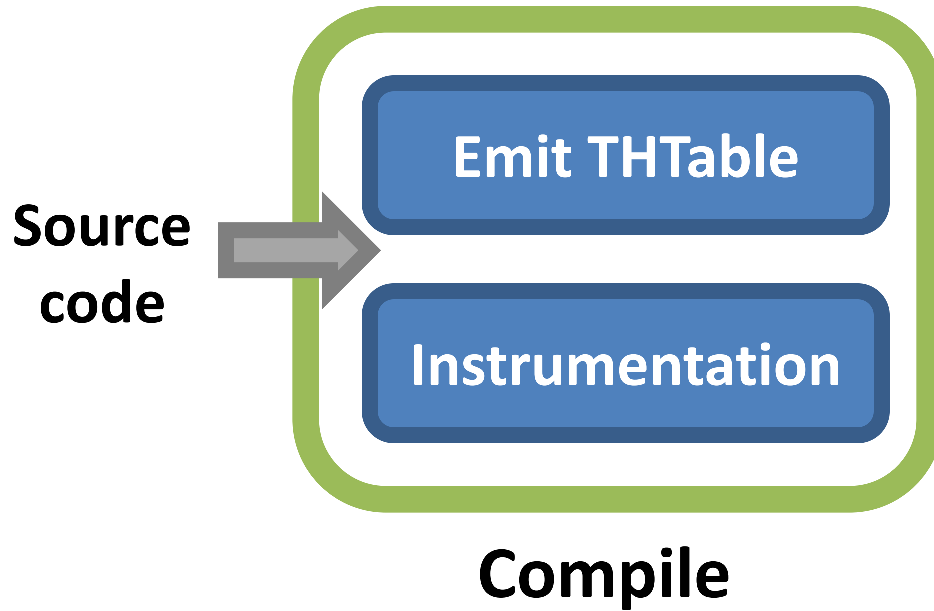
C++ supports no reliable methods to resolve whether a pointer points to polymorphic or non-polymorphic classes.

Previous solutions including Undefined Behavior Sanitizer relies on blacklists.

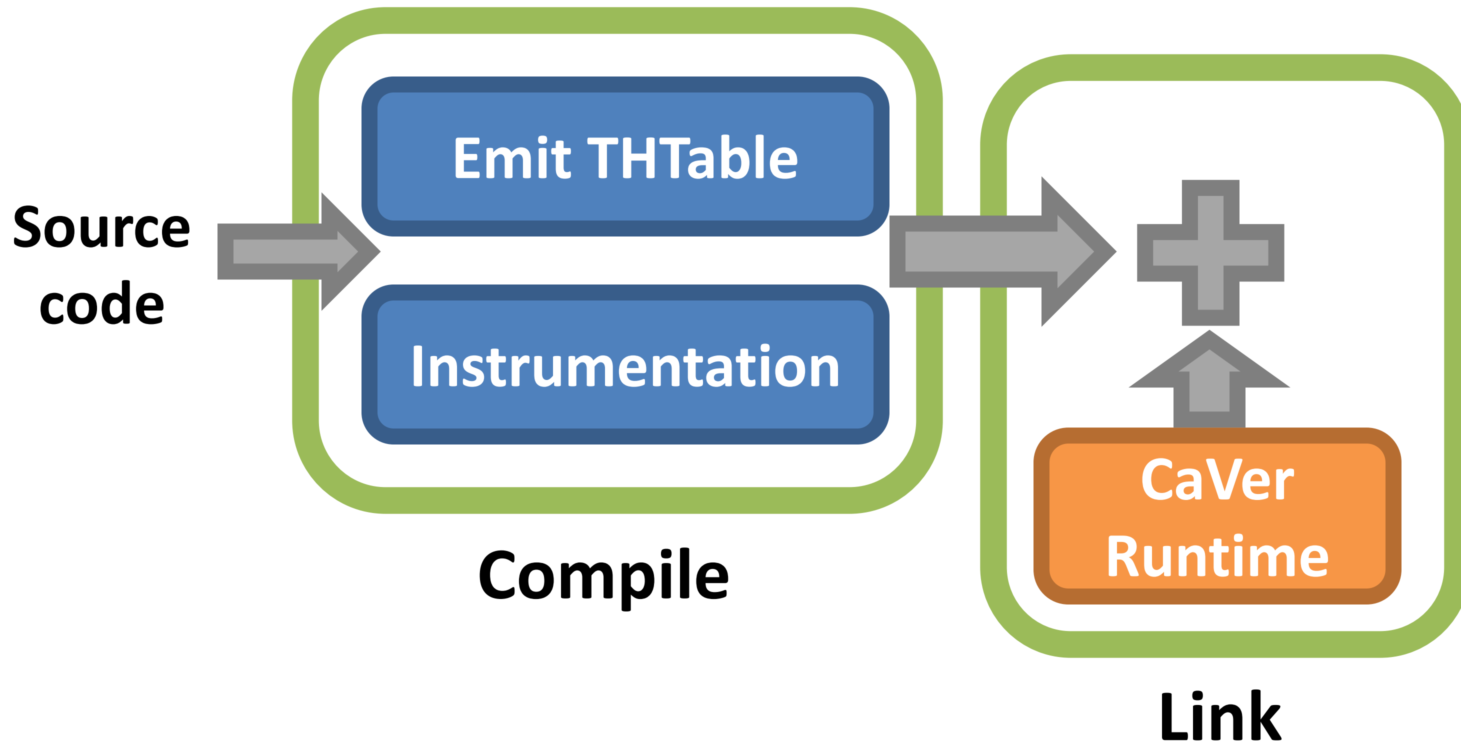
CaVer: CastVerifier

- **CaVer: CastVerifier**
  - A bad-casting detection tool
- Design goals
  - Easy-to-deploy: no blacklists
  - Reasonable runtime performance

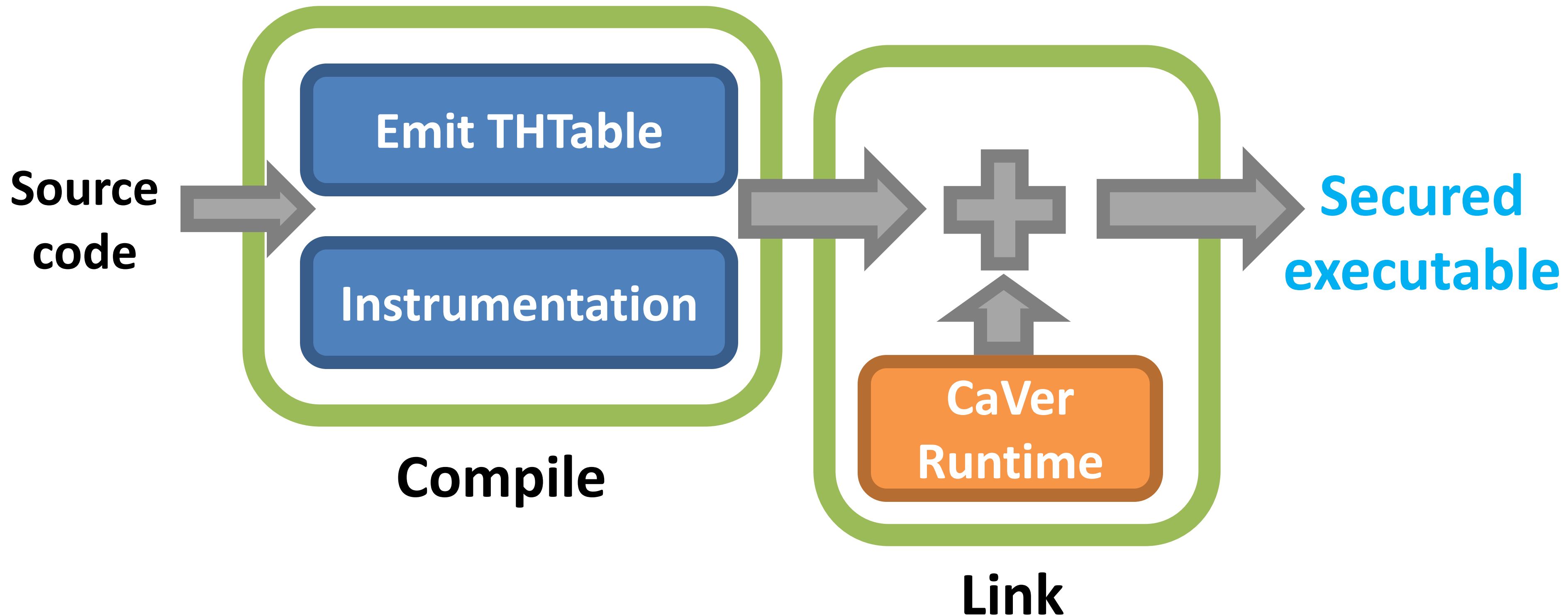
CaVer Overview



CaVer Overview



CaVer Overview



Technical Goal of CaVer

```
P *ptr = new P;
```

```
static_cast<D*>(ptr);
```

Technical Goal of CaVer

```
P *ptr = new P;
```

Allocated

```
static_cast<D*>(ptr);
```

Technical Goal of CaVer

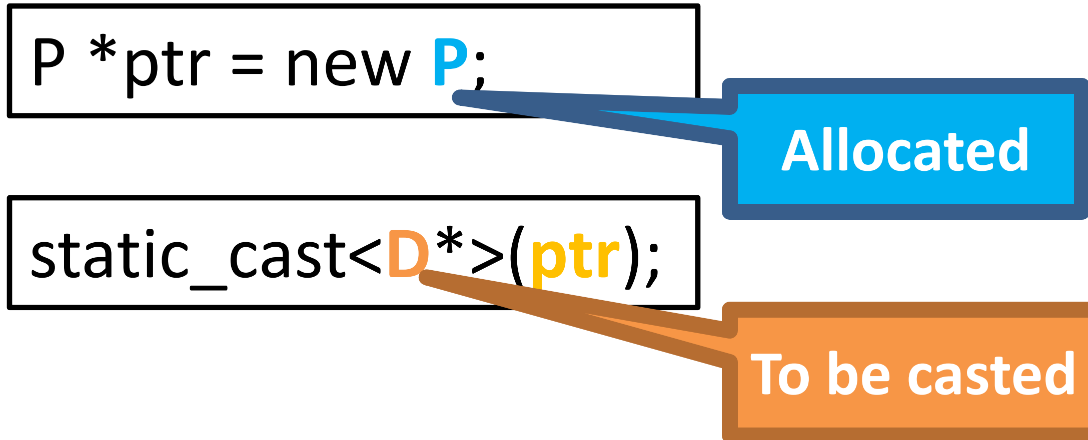
```
P *ptr = new P;
```

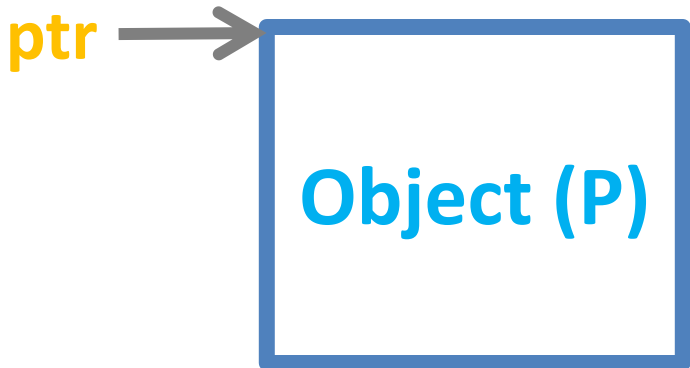
Allocated

```
static_cast<D*>(ptr);
```

To be casted

Technical Goal of CaVer





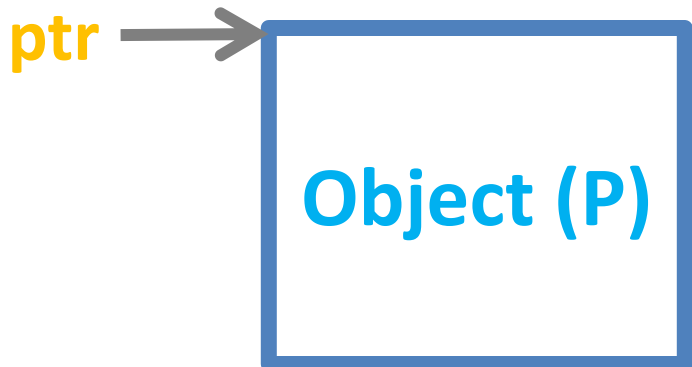
Technical Goal of CaVer

```
P *ptr = new P;
```

Allocated

```
static_cast<D*>(ptr);
```

To be casted



Q. What are the class relationships b/w P and D?
→ THTable

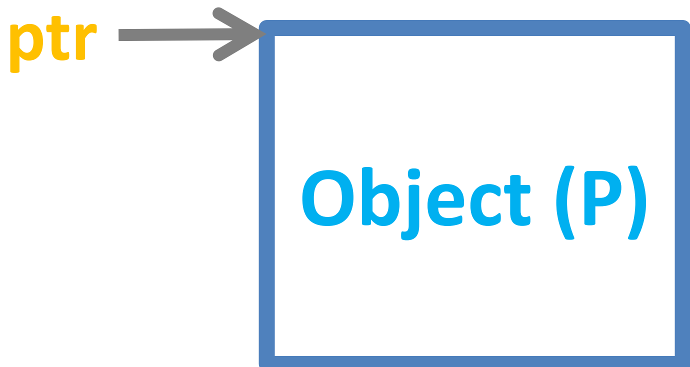
Technical Goal of CaVer

```
P *ptr = new P;
```

Allocated

```
static_cast<D*>(ptr);
```

To be casted



Q. What are the class relationships b/w P and D?
➔ THTable

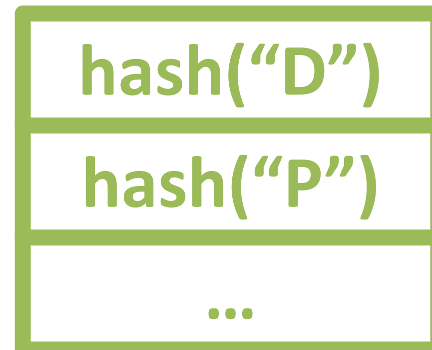
Q. Is ptr points to P or D?
➔ Runtime type tracing

Type Hierarchy Table (THTable)

- A set of all legitimate classes to be converted
  - Class names are **hashed** for fast comparison
  - Hierarchies are **unrolled** to avoid recursive traversal

THTable (P)

THTable (D)

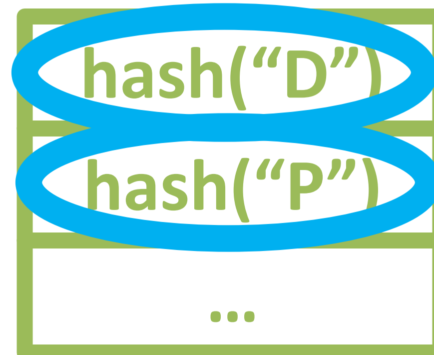


Type Hierarchy Table (THTable)

- A set of all legitimate classes to be converted
  - Class names are **hashed** for fast comparison
  - Hierarchies are **unrolled** to avoid recursive traversal

THTable (P)

THTable (D)



Hashed class names

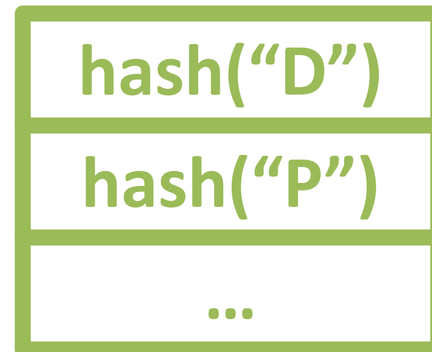
Type Hierarchy Table (THTable)

- A set of all legitimate classes to be converted
  - Class names are **hashed** for fast comparison
  - Hierarchies are **unrolled** to avoid recursive traversal

THTable (P)

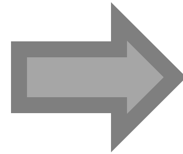
THTable (D)

Unrolled linearly



Runtime Type Tracing

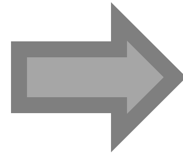
```
P *ptr = new P;
```



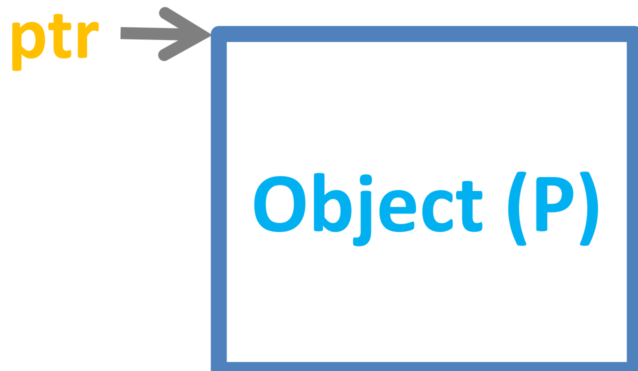
```
P *ptr = new P;
trace(ptr, &THTable(P));
```

Runtime Type Tracing

```
P *ptr = new P;
```

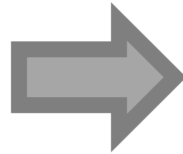


```
P *ptr = new P;
trace(ptr, &THTable(P));
```



Runtime Type Tracing

```
P *ptr = new P;
```



```
P *ptr = new P;
trace(ptr, &THTable(P));
```

&THTable(P)

THTable (P)

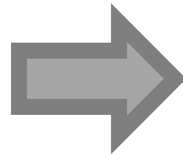
hash("P")
…

ptr

Object (P)

Runtime Type Tracing

```
P *ptr = new P;
```



```
P *ptr = new P;
trace(ptr, &THTable(P));
```

THTable (P)

&THTable(P)

hash("P")

...

ptr

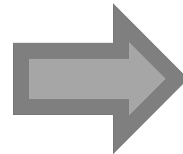
Object (P)

Maintain an internal mapping from objects to metadata

- Heap: Alignment based direct mapping
- Stack: Per-thread red-black tree
- Global : Per-process red-black tree

Runtime Type Tracing

```
P *ptr = new P;
```



```
P *ptr = new P;
trace(ptr, &THTable(P));
```

THTable (P)

&THTable(P)

hash(“P”)

...

ptr

Object (P)

Decoupled metadata
→ Overcome RTTI’s limitation

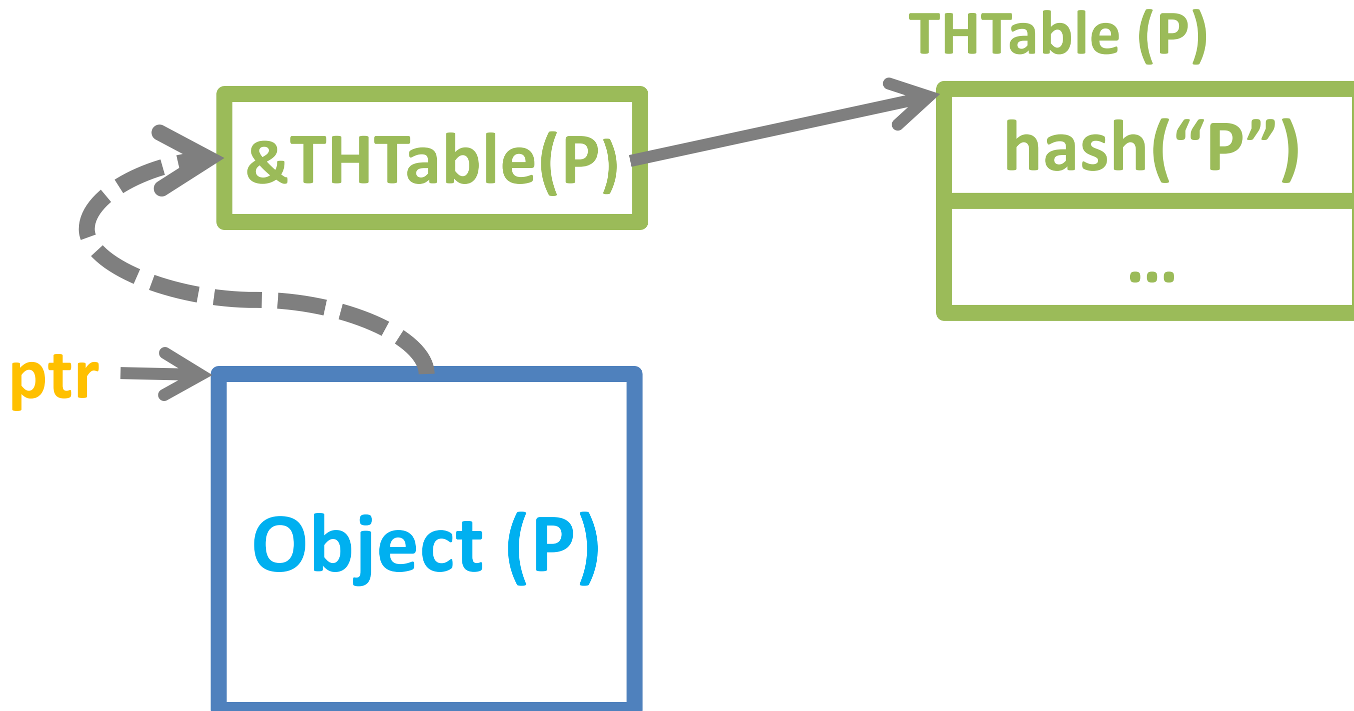
Runtime Type Verification

```
static_cast<D*>(ptr);
```

Runtime Type Verification

```
static_cast<D*>(ptr);
```

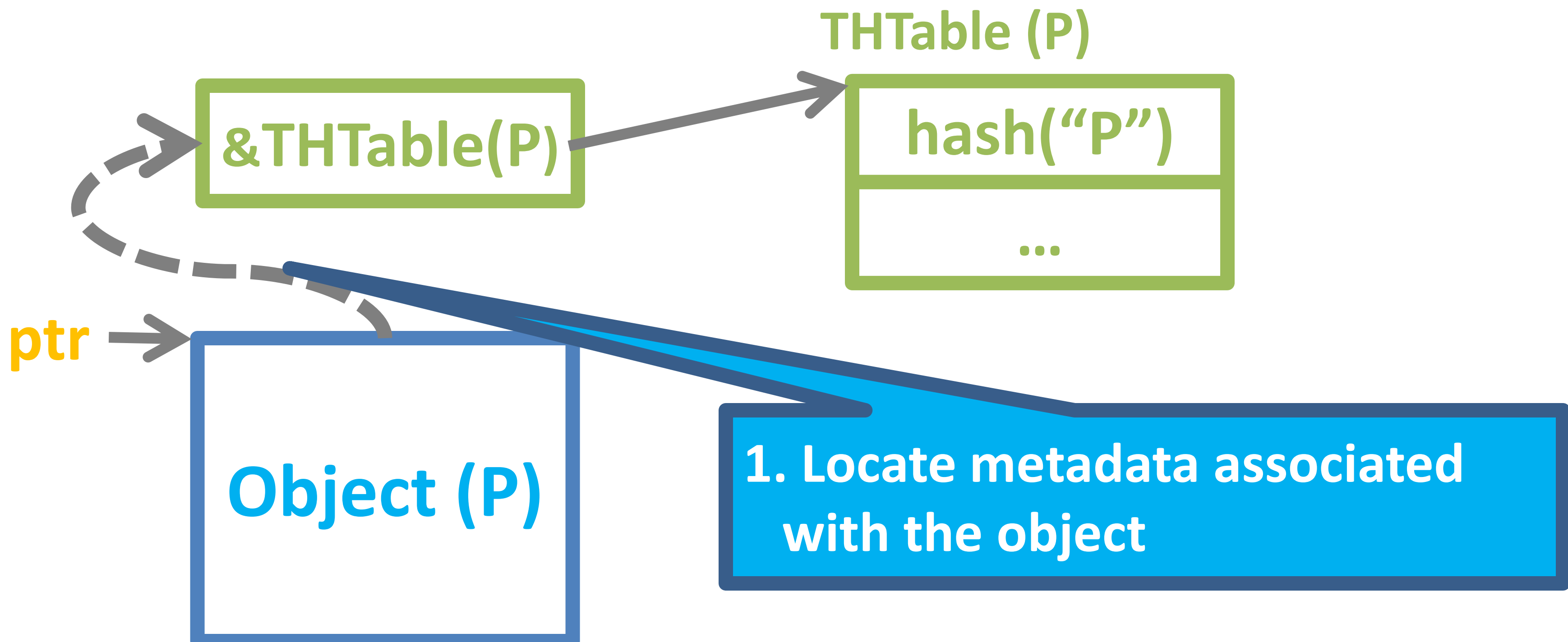
To be casted



Runtime Type Verification

```
static_cast<D*>(ptr);
```

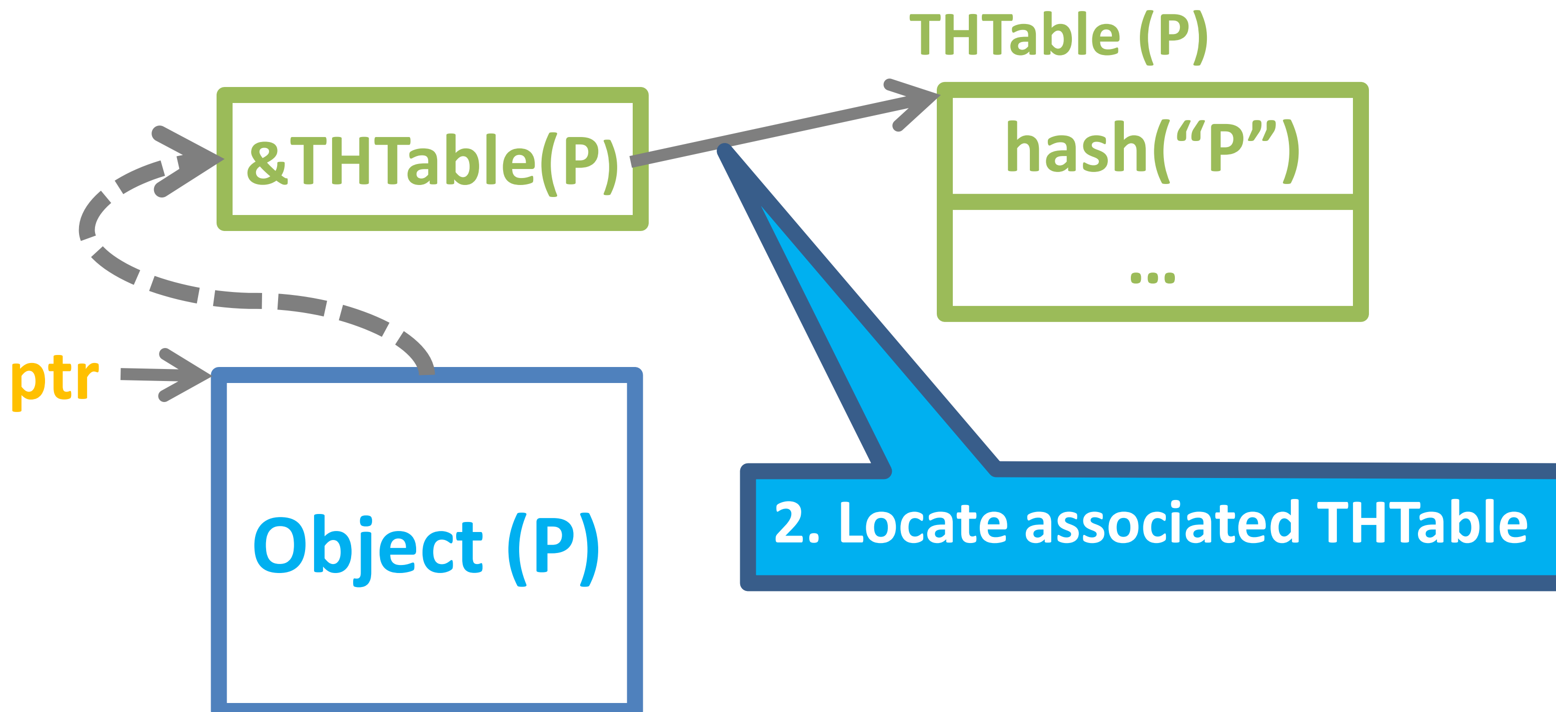
To be casted



Runtime Type Verification

```
static_cast<D*>(ptr);
```

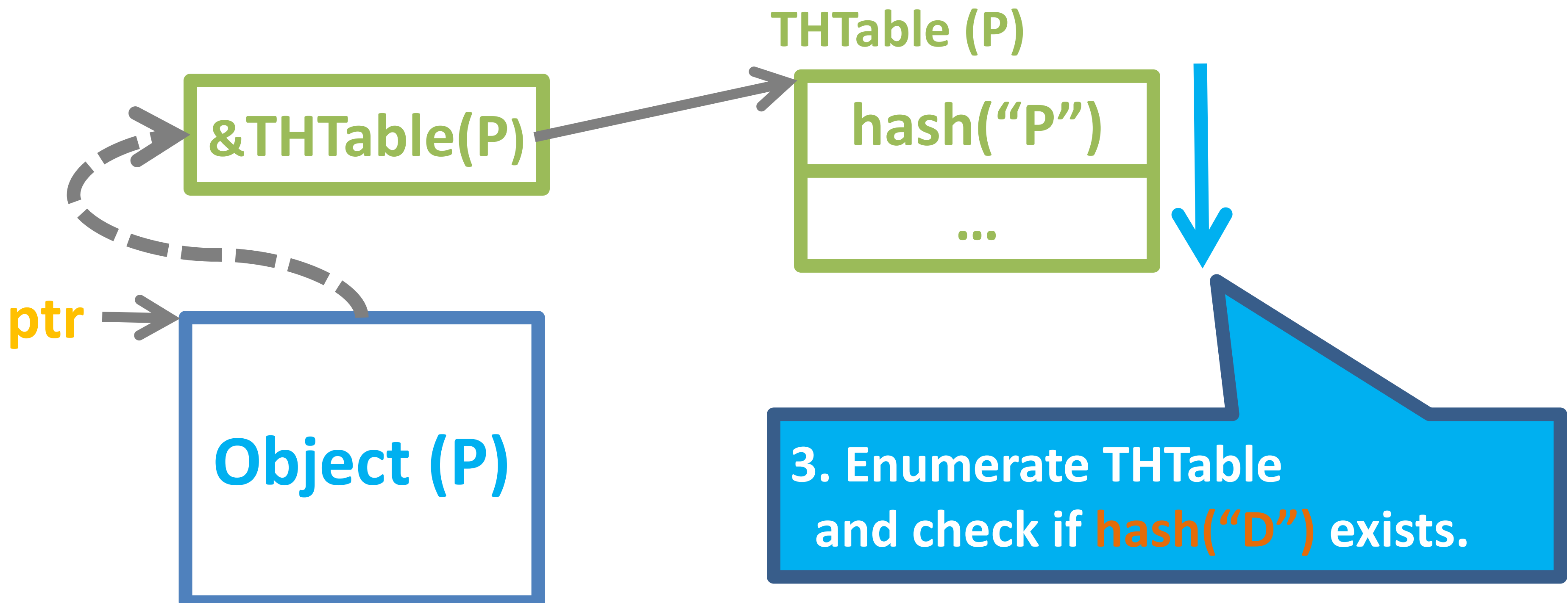
To be casted



Runtime Type Verification

```
static_cast<D*>(ptr);
```

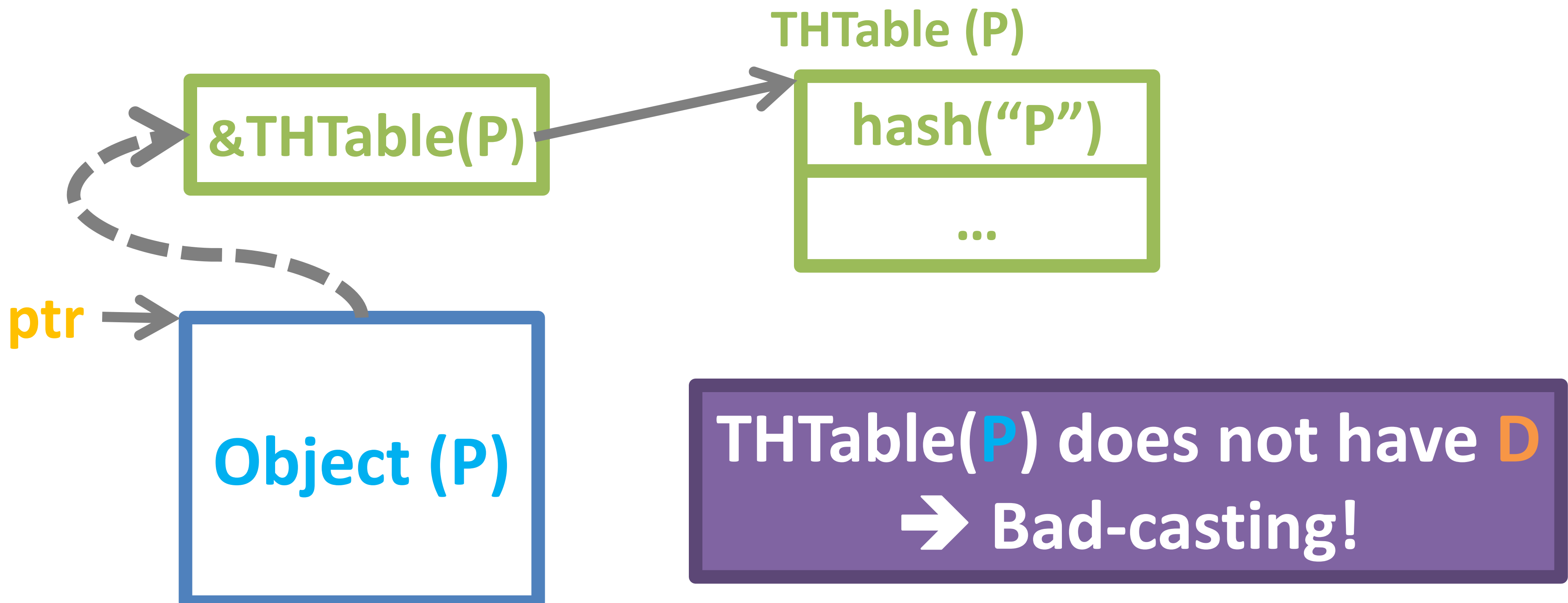
To be casted



Runtime Type Verification

```
static_cast<D*>(ptr);
```

To be casted



Performance Optimization

- **Selective object tracing**
  - Not all objects are involved in downcasting
  - Statically identify such objects, and skip tracing them
- **Reusing verification results**
  - A verification process has to be the same for same class
  - A verification result is cached for reuses

Implementation

- Based on LLVM Compiler suites
  - Added 3,540 lines of C++ code
- Currently support Linux x86-64
- CaVer can be activated with one extra compiler flag

Evaluation

- How much efforts are required to deploy CaVer?
- How effective is CaVer in detecting bad-casting?
- What is the overall runtime overhead of CaVer?

Deployment Efforts

- Build configuration changes
  - 21 and 10 lines were changed in Chromium and Firefox
  - No blacklists are required

- CaVer successfully
  - Build both browsers
  - Run both browsers without runtime crashes

CaVer Report Example

```
== CaVer : Bad-casting detected
@SVGViewSpec.cpp:87:12
  Casting an object of "blink::HTMLUnknownElement"
     from "blink::Element"
     to "blink::SVGElement"
          Pointer           0x60c000008280
          Alloc base        0x60c000008280
          Offset            0x000000000000
          THTable           0x7f7963aa20d0
#1 0x7f795d76f1a4 in viewTarget SVGViewSpec.cpp:87
#2 0x7f795d939d1c in viewTargetAttribute V8SVGViewSpec.cpp:56
...
```

CaVer Report Example

Detailed casting information

```
== CaVer : Bad-casting detected
@SVGViewSpec.cpp:87:12
  Casting an object of “blink::HTMLUnknownElement”
    from “blink::Element”
    to “blink::SVGElement”
        Pointer          0x60c000008280
        Alloc base       0x60c000008280
        Offset           0x000000000000
        THTable          0x7f7963aa20d0
#1 0x7f795d76f1a4 in viewTarget SVGViewSpec.cpp:87
#2 0x7f795d939d1c in viewTargetAttribute V8SVGViewSpec.cpp:56
...
```

CaVer Report Example

```
== CaVer : Bad-casting detected
@SVGViewSpec.cpp:87:12
  Casting an object of "blink::HTMLUnknownElement"
    from "blink::Element"
    to "blink::SVGElement"
        Pointer          0x60c000008280
        Alloc base       0x60c000008280
        Offset           0x000000000000
        THTable          0x7f7963aa20d0
#1 0x7f795d76f1a4 in viewTarget SVGViewSpec.cpp:87
#2 0x7f795d939d1c in viewTargetAttribute V8SVGViewSpec.cpp:56
...
```

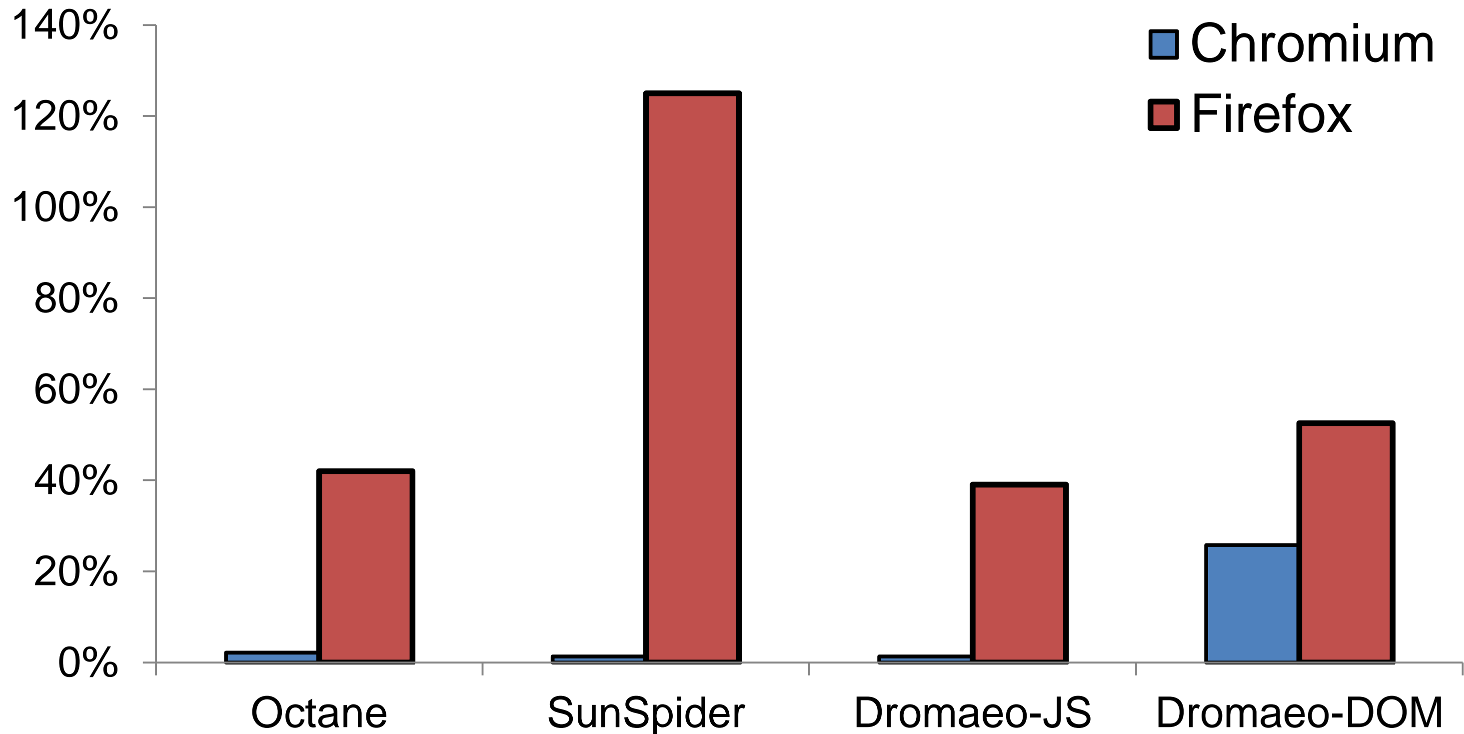
Detailed casting information

Runtime call stacks

New vulnerabilities

- CaVer discovered **11 new vulnerabilities**
  - 2 cases in Firefox (won bug bounty awards)
  - 9 cases in GNU libstdc++
  - All reported to and fixed by vendors

Runtime Overhead



On average,
Chromium: 7.6%
Firefox: 64.6%

Applications of CaVer

- A back-end bug detection tool
- A runtime attack mitigation tool
  - Limitations of previous mitigations techniques
    - Focusing on certain attack methods
      - e.g., CFI or ROP techniques
    - Not effective if an exploit relies on other attack methods
      - e.g., non-control data attack
  - CaVer tackles the root cause of bad-casintg.

Conclusions

- Proposed CaVer, a new runtime bad-casting detection mechanism
- Discovered 11 new bad-casting vulnerabilities in Firefox and libstdc++

Thank you!