VTrust: Regaining Trust on Virtual Calls

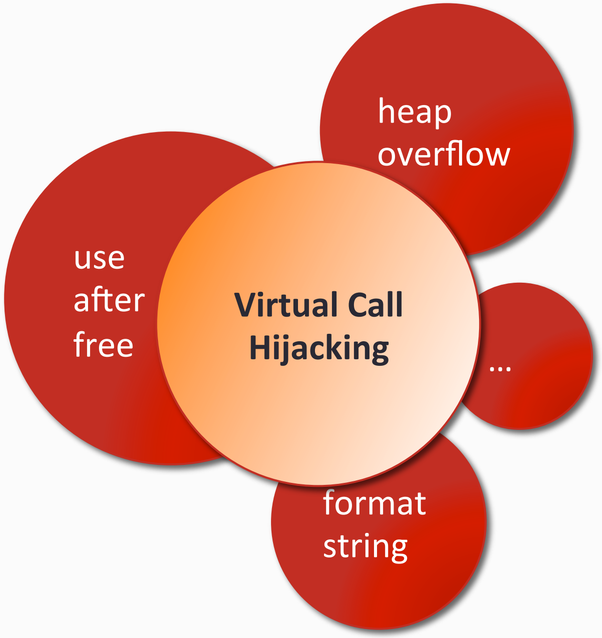
Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, Dawn Song

UC Berkeley, Purdue University, Peking University, Georgia Tech

Virtual Call Hijacking in real world

- written in C++
- heavy use of virtual functions and virtual calls

A common way to exploit:

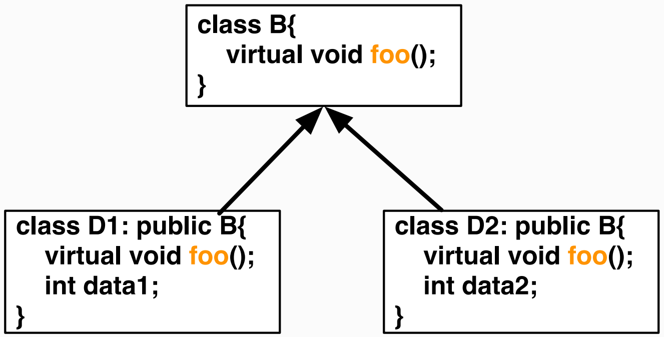


plenty of vulnerabilities:

Google:
"80% attacks exploit use-after-free...”
Microsoft:
50% CVEs targeted Winows7 are UAF

Virtual Calls

Class Hierarchy:

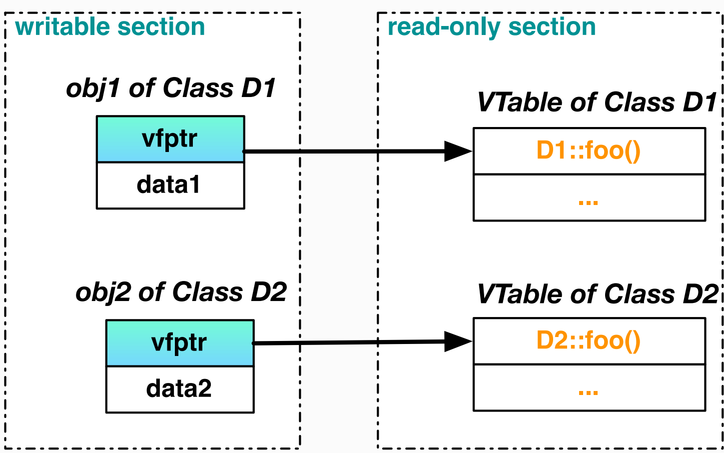



```
void test ( B*  obj )
{
    obj→foo();      // virtual call site
}
```

B::foo, D1::foo, or D2::foo?

- How to resolve the virtual function of an object at runtime?
  - VTable pointers in objects

Runtime Memory:



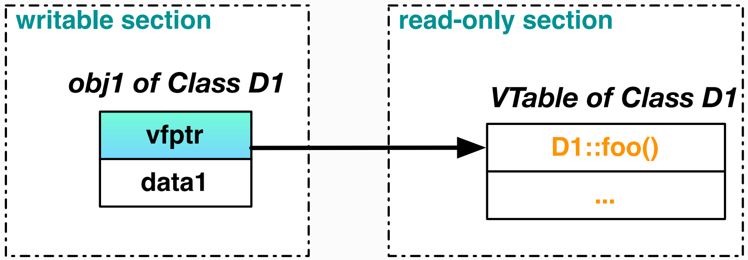
Resolve virtual functions:

Step 1: read VTable pointer from obj

Step 2: read function pointer from VTable

Virtual Call Hijacking

Runtime Memory:

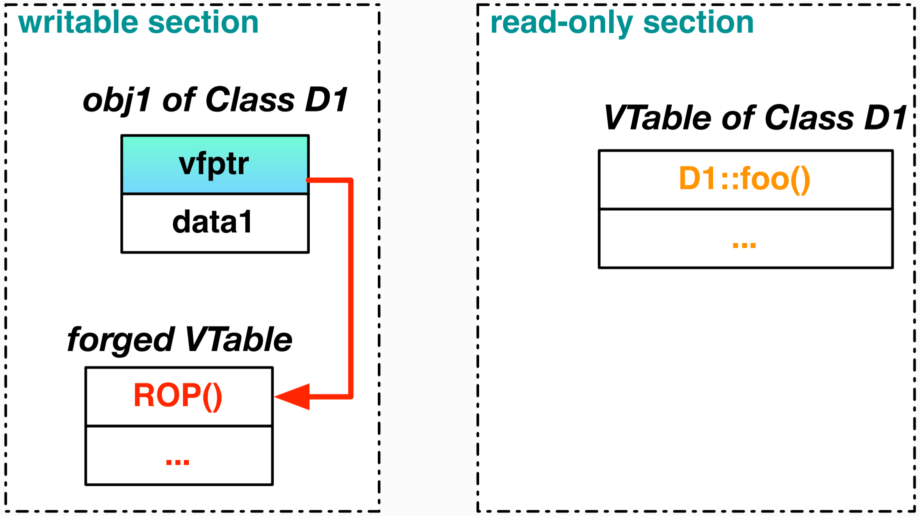


Resolve virtual functions:

Step 1: read VTable pointer from obj

Step 2: read function pointer from VTable

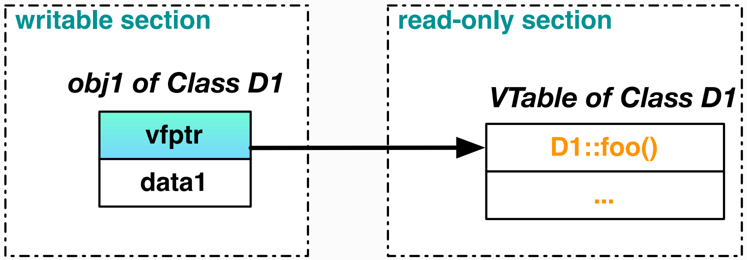
- Attacks: breaking the integrity of VTable pointers
  - VTable injection attack: vfptr points to **forged** VTables



Practical and reliable:
virtual call hijacking + ROP

Virtual Call Hijacking (2)

Runtime Memory:

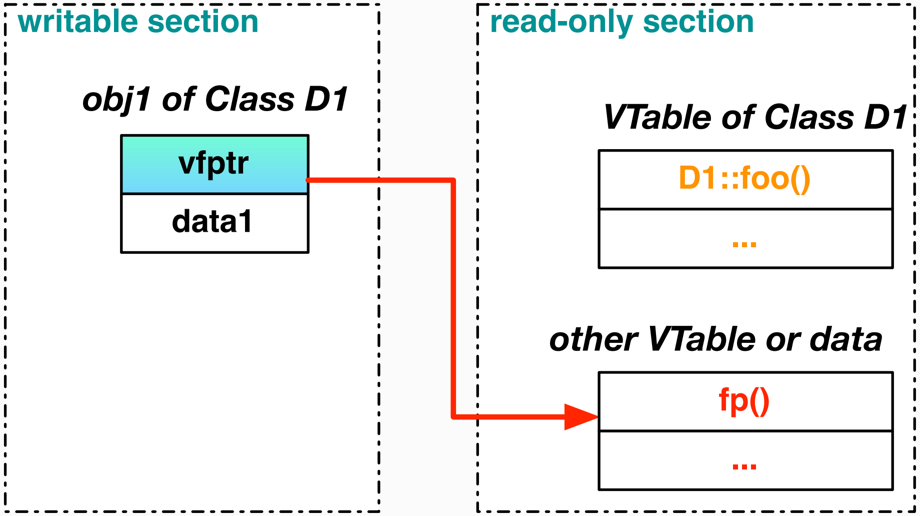


Resolve virtual functions:

Step 1: read VTable pointer from obj

Step 2: read function pointer from VTable

- Attacks: breaking the integrity of VTable pointers
  - VTable injection attack: vfptr points to **forged** VTables
  - VTable reuse attack: vfptr points to **existing** but out-of-context VTables



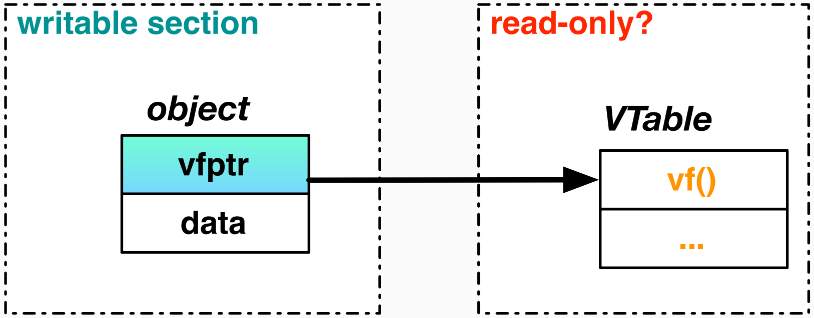
COOP attack [S&P'15]

Outline

- Motivation
- Related Work
- Design
- Implementation
- Evaluation
- Conclusion

Binary Level Defenses

- VTint [NDSS'15]
- T-VIP [ACSAC'14]
  - enforce read-only



- Pro:
  - binary-compatible
  - could defat popular VTable injection attacks
- Con:
  - false positives
  - cannot defeat VTable reuse attacks, e.g., COOP

Source Level Defense: **Forward Edge CFI**

- GCC-VTV [Usenix'14], whitelist-based

```
C *x = ...
ASSERT(VPTR(x) ∈ Valid(C));
x->foo();
```

  - compute **an incomplete set** of legitimate targets at **compile-time**
  - **merge** this set by using initializer functions **at load time**
  - **validate** runtime target against this set **at runtime**

- Pro:
  - support incremental building

- Con:
  - heavy runtime operation, i.e., hash table lookups

Source Level Defense: **RockJIT**

- CCS’15, CFI-based
  - collect **type information** at **compile-time**
  - compute **equivalence classes** of transfer targets at **load time**, based on the collected type information.
  - update the **CFI checks** to only allow indirect transfers (including virtual calls) to one equivalence class at **load-time**

- Pro:
  - support incremental building
- Con:
  - heavy load time operations

Outline

- Motivation
- Related Work
- Design:
  - Virtual Function Type Enforcement
  - VTable Pointer Sanitization
- Implementation
- Evaluation
- Conclusion

Virtual Function Type

```
void test ( B*  obj, int arg1, void*  arg2)
{
     // virtual call site
     obj→foo(arg1, arg2);
}
```

```
class D: public B
{
      // virtual functions
     virtual void foo(int arg1, void*  arg2);
}
```

- A virtual function is allowed at a virtual call site if and only if it has:
  - a matching function name
  - a matching argument type list
  - matching qualifiers (constant, volatile, reference)
  - a compatible class

Virtual Function Type Enforcement

```
// virtual call site: expected type
obj→foo(arg1, arg2);
```

```
// virtual functions definitions: target type
virtual void foo(int arg1, void*  arg2) ;
```

```
ASSERT( expected_type == target_type )
obj→foo(arg1, arg2);
```

- How to encode the type information, to enable fast type lookup and comparison?
  - RTTI–based solutions are too slow

Virtual Function Type Enforcement

// virtual call site: expected type
obj→foo(arg1, arg2);

// virtual functions definitions: target type
virtual void foo(int arg1, void* arg2);

ASSERT(expected_type == target_type)
obj→foo(arg1, arg2);

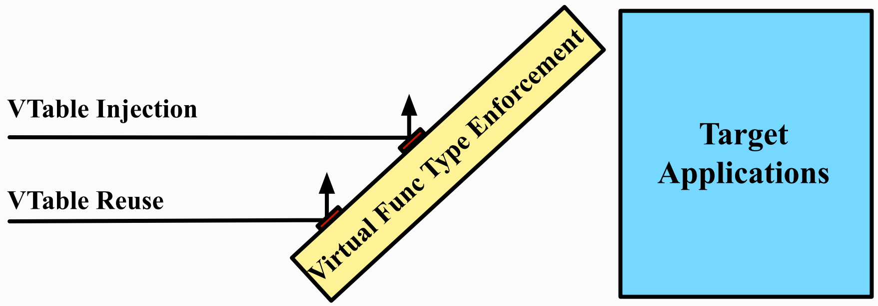
ASSERT(expected_signature == target_signature)
obj→foo(arg1, arg2);

- Our solution: compute a signature for the type

  signature = hash (funcName, typeList, qualifiers, classInfo)

- All signatures can be computed statically and independently.
  - support incremental building
  - don't need extra link-time, load-time or runtime support
  - fast and easy to deploy

Security Analysis



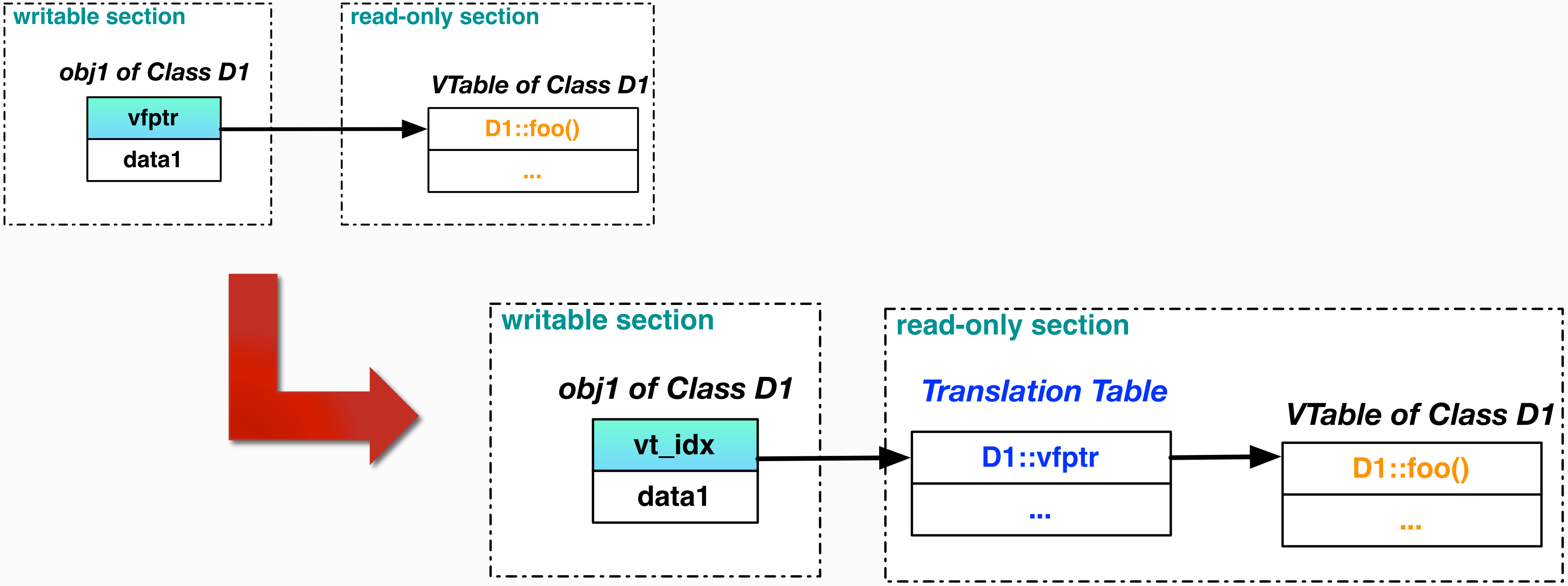
- No matter what VTables are used, target virtual functions must have matching signatures.

- Attackers can forge signatures if and only if
  - target applications have dynamic generated code.

VTable Pointer Sanitization

- Solution: limit the target functions to static code

- How?
  - sanitize VTable pointers, to only allow legitimate VTables used for virtual function lookup.



Outline

- Motivation
- Related Work
- Design
- Implementation
- Evaluation
- Conclusion

Virtual Function Type Enforcement

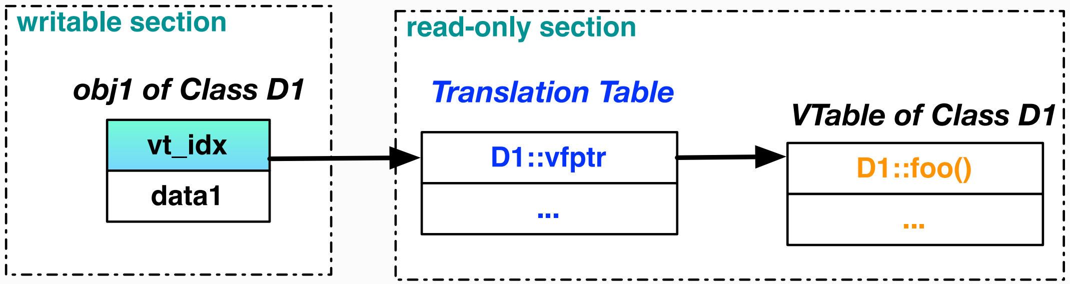
ASSERT(expected_signature == target_signature)

obj→foo(arg1, arg2);

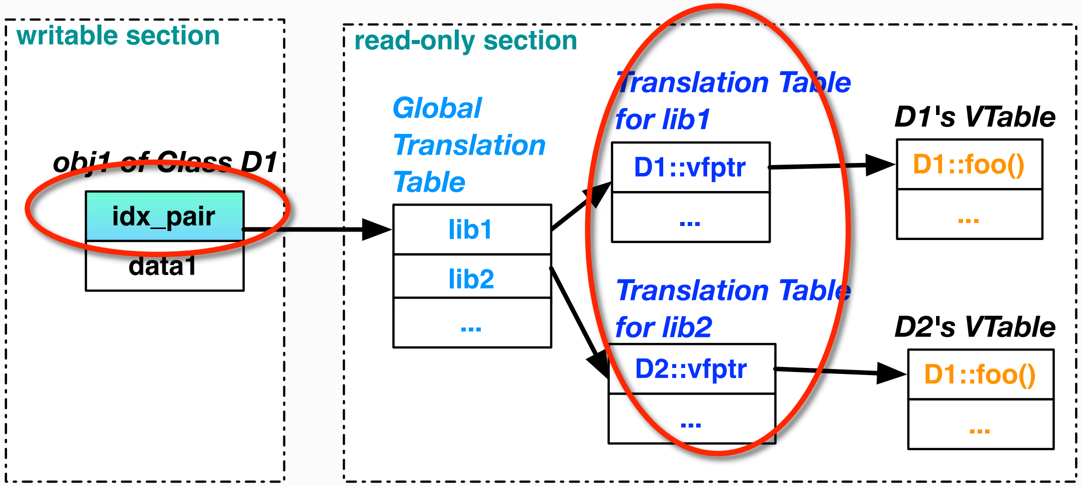
signature = hash (funcName, typeList, qualifiers, classInfo)

- Compute signatures
  - function name:
    - destructor functions
    - member function pointers
  - class info:
    - top-most primary class' name
- Instrument signatures
  - hard-coded before virtual call sites
  - hard-coded before virtual function bodies

VTable Pointer Sanitization

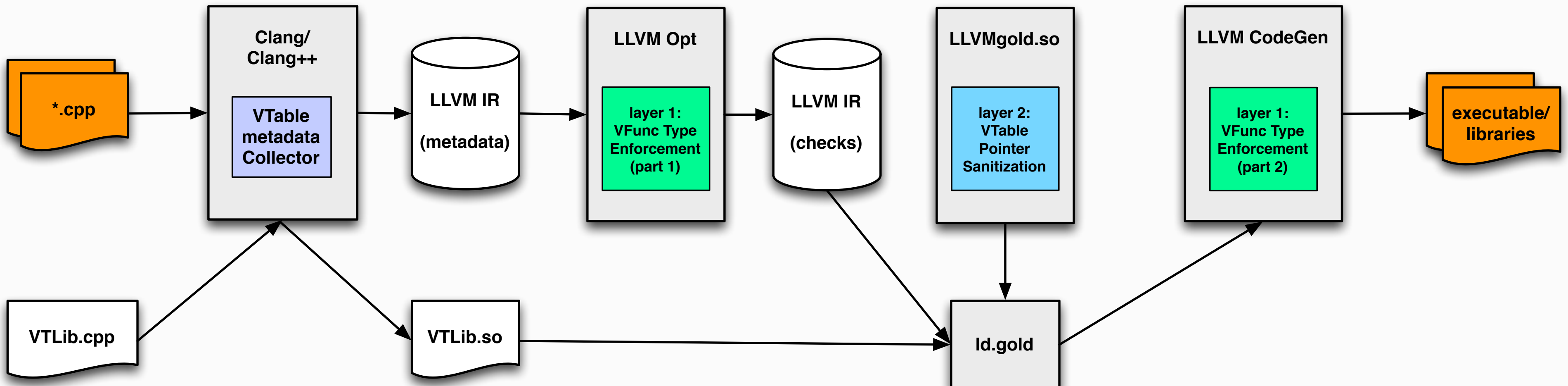


- A centralized translation table is impractical
  - merge this table at load-time
  - update vt_idx in constructor functions
- Our solution: distributed translation table
  - each library has its own translation table



- constant library translation tables.
- constant idx_pair

Overall Workflow

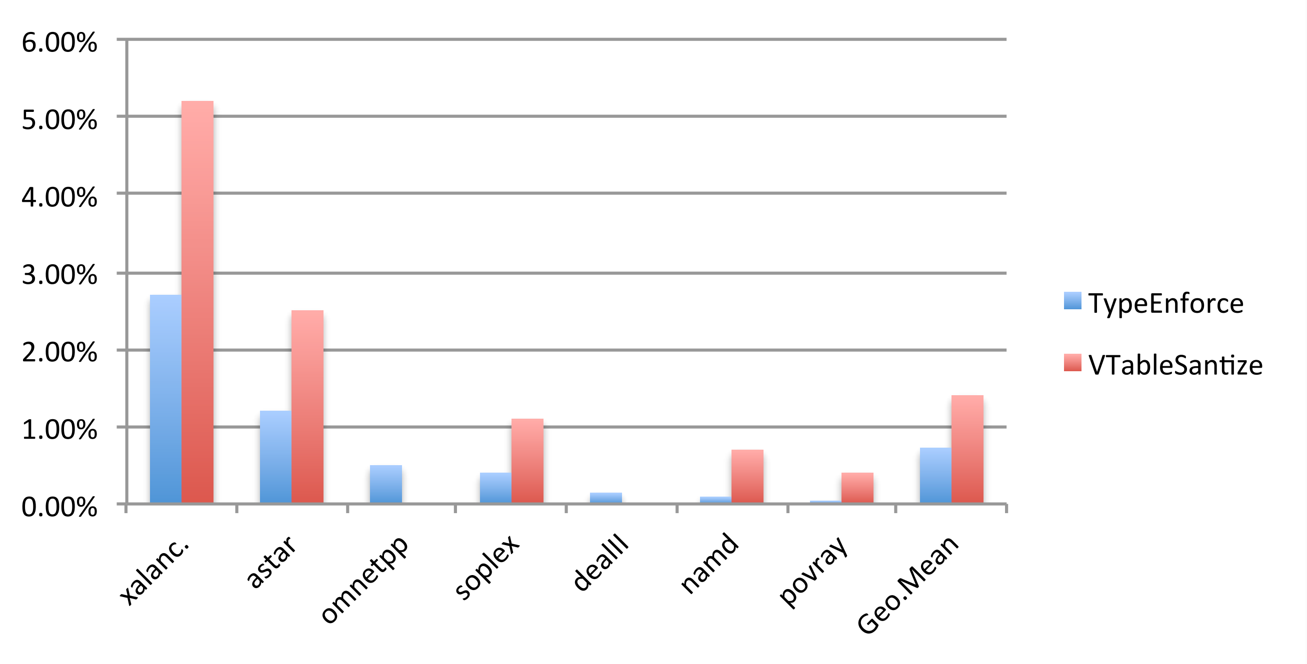


Outline

- Motivation
- Related Work
- Design
- Implementation
- Evaluation
- Conclusion

Performance Overhead

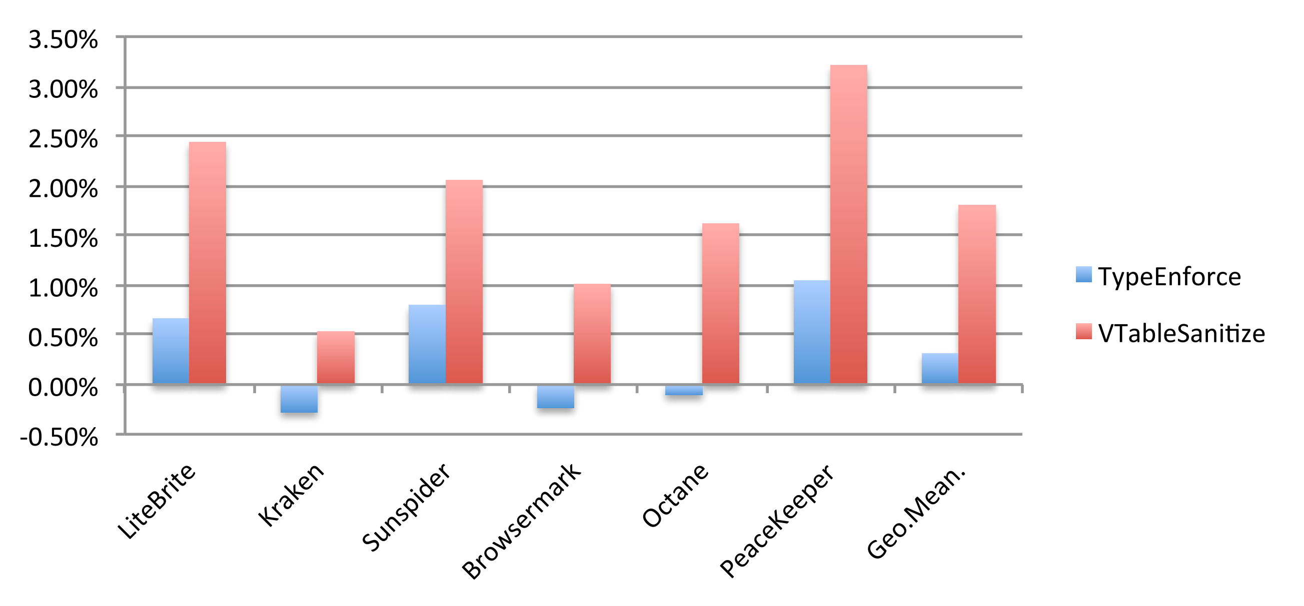
- SPEC 2006
  - avg (two layers together): 2.2% **(~ 0.72% + 1.4%)**
  - worst: 8.0%



The 1st layer defense is much faster than the 2nd layer, sufficient for programs without dynamic generated code.

Performance Overhead

- Firefox
  - avg (two layers together): 2.2% **(~ 0.4% + 1.8%)**
  - worst: 4.3%



Protection Effects

- VTable injection attacks

CVE-ID	Exploit Type	Vul App	Protected
CVE-2013-1690	VTable injection	FF 21	YES
CVE-2013-0753	VTable injection	FF 17	YES
CVE-2011-0065	VTable injection	FF 3	YES

- VTable reuse attacks (few in real world)
  - BCTF challenge *"zhongguancun"*

Corner cases

- Custom virtual function definitions
  - e.g., *nsXPTCStubBase::StubNN()* in Firefox
  - will cause false positives

- Custom virtual call sites
  - e.g., *NS_InvokeByIndex()* in Firefox
  - will leave extra attack surface

- VTrust could identify all these corner cases automatically, and provides a precise protection.

Conclusion

- VTrust provides two layers of defenses against all virtual call hijacking attacks.
- **Virtual function type enforcement** introduces a very low performance overhead, able to defeat all this type of attacks if no dynamic code exists in target applications.
- **VTable pointer sanitization** could help defeat all attacks even if target applications have dynamic code.
- The performance and security evaluation show that VTrust has a low performance overhead, and provides a strong protection.

Thanks!

Q&A