Google Ph.D. Student Summit on Compiler & Programming Technology, 5-7 Dec. 2016, Munich, Germany

vTableShield: Precise Protecting of Virtual Function Dispatches in C++ Programs

(work in progress)

Technical University of Munich, Computer Science Department, Chair for IT-Security
Paul Muntean, Peng Xu, and Claudia Eckert

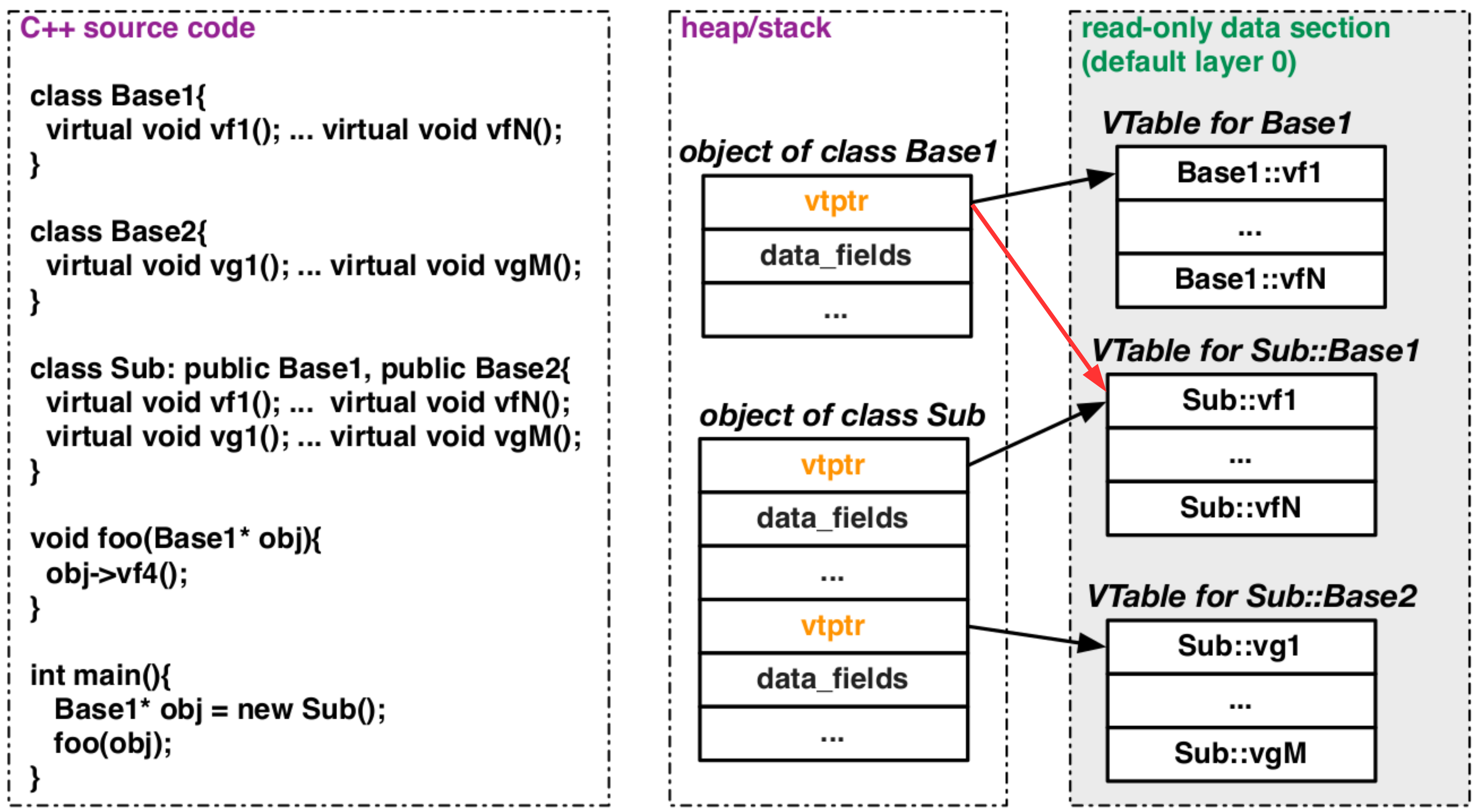
1. Introduction

- **Problem:** The currently used Control-Flow Integrity (**CFI**) protection schema in [1] is too permissive; it allows too many **(and impermissible)** call-targets per call-site.
- **Current Solutions:**
  a)*Compiler-based techniques:*
  [Bounov et al. ***NDSS'16***][1]
  [ShrinkWrap, ***ACSAC'15***][2], [IFCC/VTV, ***USENIX'14***][3],
  b)*Binary-based techniques:* [vTint, ***NDSS'15***][4], [TypeArmor, ***S&P'16***][5]
  c)*Run-time-based techniques:* Intel CET [6], Windows CFGuard [7].
- **Limitations of Current Solutions:**
  a) **Precision:** of caller/callee mapping can still be improved.
  b) **Performance:** *worst-case* run-time overhead,7-8%; drops to 2%.
  c) **Identification:** accuracy of call-site/call-target (for binaries) is low
- **Our Insight:** The number of *call-targets* per *call-site* can be ***reduced*** by carefully analyzing the class and virtual table hierarchies.
  → we use the call-site object type (base class) and the virtual table of the calling object.

2. Contributions

- We **reduced** the number of call-targets per call-site, thus improving the precision of our mapping. ***(precision)***
- We **decreased** the performance overhead w.r.t [1]. ***(performance)***
- We **shrinked** the binary blow-up size. ***(binary size)***
- We **improved** the protection coverage. ***(increased security level)***

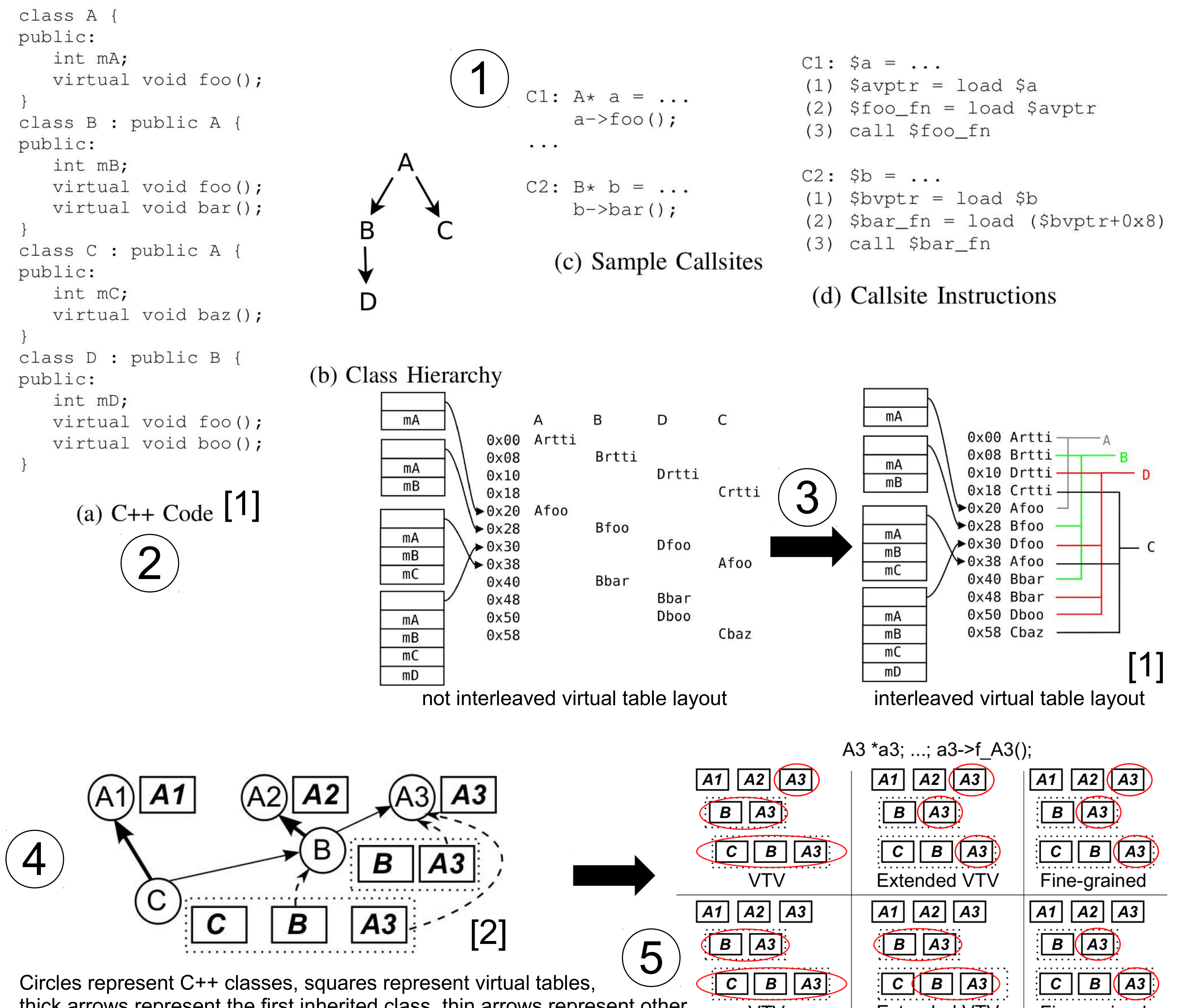
3. Motivating Example



```
class Base1{
  virtual void vf1(); ... virtual void vfN();
}

class Base2{
  virtual void vg1(); ... virtual void vgM();
}

class Sub: public Base1, public Base2{
  virtual void vf1(); ...  virtual void vfN();
  virtual void vg1(); ... virtual void vgM();
}

void foo(Base1* obj){
  obj->vf4();
}

int main(){
   Base1* obj = new Sub();
   foo(obj);
}
```

- Consider: ***Base1* obj2 = new Base1(); obj2→ vfN();***
- The virtual pointer can be corrupted (i.e., red arrow from above Figure) to point into a different virtual table. The new virtual table is (not) in the expected class or virtual table hierarchy.

4. Background



```
class A {
public:
   int mA;
   virtual void foo();
}
class B : public A {
public:
   int mB;
   virtual void foo();
   virtual void bar();
}
class C : public A {
public:
   int mC;
   virtual void baz();
}
class D : public B {
public:
   int mD;
   virtual void foo();
   virtual void boo();
}
```

(a) C++ Code [1]

(b) Class Hierarchy

```
C1: A* a = ...
    a->foo();
...

C2: B* b = ...
    b->bar();
```

(c) Sample Callsites

```
C1: $a = ...
(1) $avptr = load $a
(2) $foo_fn = load $avptr
(3) call $foo_fn

C2: $b = ...
(1) $bvptr = load $b
(2) $bar_fn = load ($bvptr+0x8)
(3) call $bar_fn
```

(d) Callsite Instructions

not interleaved virtual table layout

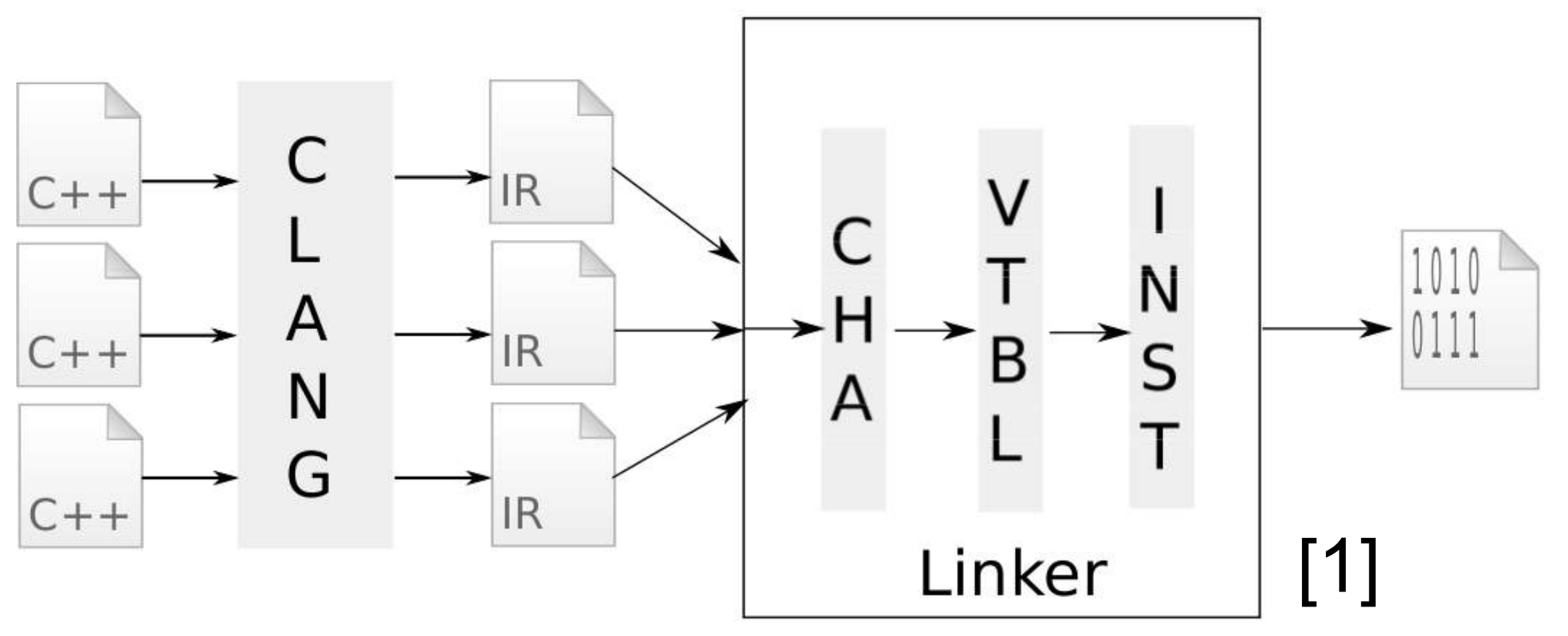
interleaved virtual table layout

Circles represent C++ classes, squares represent virtual tables, thick arrows represent the first inherited class, thin arrows represent other inherited classes and dashed arrows represent inheritance relations between the virtual tables.

5. Design

- Obtain the ***object type*** and the ***virtual table*** used during the object dispatch.
- We ***interleave*** the virtual table layouts such that we obtain the smallest possible range for each indirect call site.
- We ***filter the resulted ranges*** based on virtual table inheritance paths such that we obtain the smallest candidate range per call-site.

6. Implementation



- The Clang (LLVM front-end) is extended in order to provide the virtual tables as meta data during LLVM link time.
- The class hierarchy analysis (CHA) is used do compute virtual table inheritance paths.
- The virtual table inheritance paths are analyzed in order to derive permissible and impermissible ranges for each call-site.
- The new range checks are added before each indirect call site.

7. Evaluation

- **Research Questions**
  RQ1: How precise is ***vTableShield***? (call-site/call-targets mapping)
  RQ2: What is the performance of ***vTableShield***?
  RQ3: What is the binary blow-up after adding the CFI checks?
  RQ4: What is the protection coverage (sec. level) w.r.t. other tools?
- **Test Programs**
  Google Chrome, Google V8 Engine, SPEC 2006, etc.
- **Methodology**
  We run ***vTableShield*** on each program in order to compute the smallest possible range for each call-site; next range checks habe been added during LLVM link time.
- **Experimental Setup**
  For testing purpose we use a system having an 64-bit Linux kernel, and the Intel i5-3230 CPU@2.60GHz×4.
- **Results**
  RQ1(precision): The number of call-targets per call-site was reduced of up to 50% w.r.t. [1] on average.
  RQ2 (performance): We improved the run-time performance overhead w.r.t. previous work.
  RQ3 (binary size blow-up): The binary size has been reduced w.r.t. previous work.
  RQ4 (increased level of security): By reducing the call-targets per call-site ratio (i.e., thousands of call-targets per call-site are no longer available to the attacker) we raised the bar for any attacker who tries to use the remaining attack surface.

8. Conclusion and Future Work

- In this work, we presented ***vTableShield***, a compiler based tool used during run-time to enforce the most precise range of virtual tables per call-site.
- The results depicted in Section 7 considerably raise the bar for any attacker who wants to exploit: Google Chome, Google V8 Engine, etc.
- In future, we want to further improve the forward CFI protection schema and provide a similar protection schema for backward edges (e.g., virtual function returns).

9.References

[1] Bounov et al. “Protecting C++ Dynamic Dispatch Through Vtable Interleaving”, In: **NDSS'16**.
[2] Haller et al. “ShrinkWrap: VTable Protection without Loose Ends”, In: **ACSAC'15**.
[3] Tice et al. “Enforcing Forward-edge Control-flow Integrity in GCC and LLVM”, In: **USENIX SEC'14.**
[4] Zhang et al. “vTint: Protecting Virtual Function Tables’ Integrity”, In: **NDSS'15.**
[5] van der Veen et al. “A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level”, In: **S&P'16.**
[6] Intel Control-flow Enforcement Technology (CET), URL: http://blogs.intel.com/evangelists/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks/
[7] Microsoft, Windows Contol Flow Guard, URL: https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

Supported by the Bavarian State Ministry of Education, Science and the Arts as part of the FORSEC research association.

Bayerisches Staatsministerium für
Bildung und Kultus, Wissenschaft und Kunst