TyPro: Forward CFI for C-Style Indirect Function Calls Using Type Propagation

Markus Bauer, Ilya Grishchenko, Christian Rossow

Contact: markus.bauer@cispa.saarland
Code: https://github.com/typro-type-propagation/TyPro-CFI

Function Pointers and Indirect Calls in C

Store in Function Pointer Variables

```c
void (*func_ptr)(int);
func_ptr = &function1;
```

→

Indirect Calls using Function Pointers

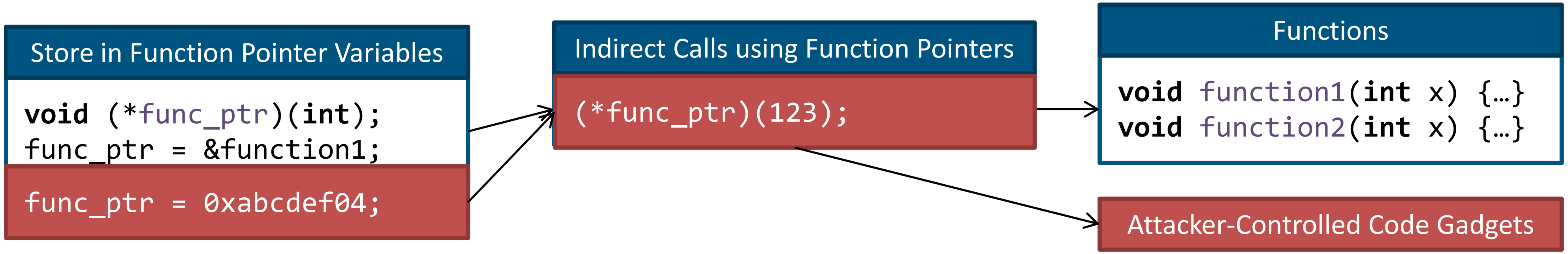
```c
(*func_ptr)(123);
```

→

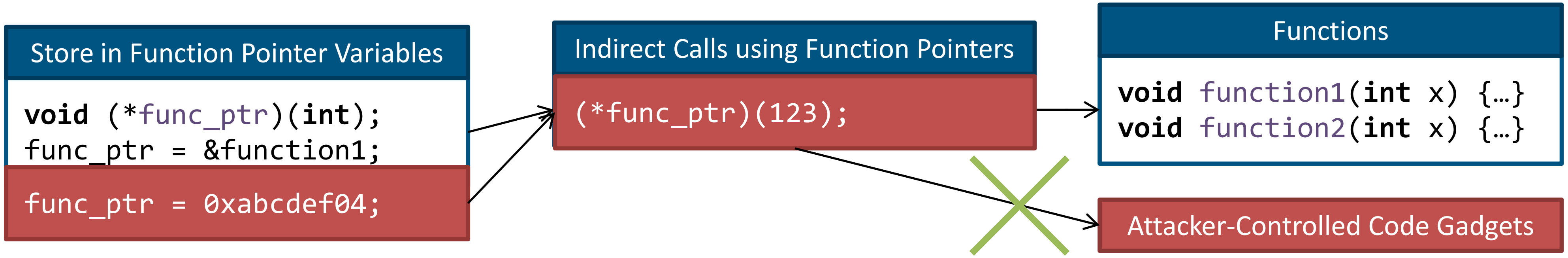
Functions

```c
void function1(int x) {…}
void function2(int x) {…}
```

Function Pointers and Indirect Calls in C



Control Flow Integrity (CFI) for Indirect Calls



- Indirect Calls should only call legitimate functions
- Control Flow Integrity Schemes:
  - Build **Target Sets** for each indirect call
  - Ensure only valid targets can be called

TargetSet = {function1, function2}

Control Flow Integrity (CFI) in Practice

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

functional

Control Flow Guard
(Microsoft)

- Each function is a target for each indirect call
- **Risky overapproximation** (possibly >1000 targets)

most possibly secure

Clang CFI

- A function is a target if function type == pointer type
- **Breaks applications**

Problems of Strict CFI

Unmatched Argument Types (lighttpd, hmmer)

```
long (*filter_ptr)(void *,    data *);
long  filter_func (plugin *p, data *d) {}

int (*compare_ptr)(const void *, const void *);
int  compare_func (long a,       long b) {}
```

Variadic Function Types (gcc, nginx)

```
rtx (*gen_ptr)(rtx, ...);
rtx  gen_func (rtx a, rtx b, rtx c) {}

handler_t (*mod_ptr) (...);
handler_t mod_deflate(request *r, void *data) {}
```

Control Flow Integrity (CFI) in Practice

functional	functional <u>and</u> secure	most possibly secure
Control Flow Guard (Microsoft)	TyPro	Clang CFI

- Each function is a target for each indirect call
- **Risky overapproximation** (possibly >1000 targets)

- A function is a target if function type == pointer type
- **Breaks applications**

Problems of Strict CFI

Unmatched Argument Types (lighttpd, hmmer)

```
long (*filter_ptr)(void *,     data *);
long  filter_func (plugin *p, data *d) {}

int (*compare_ptr)(const void *, const void *);
int  compare_func (long a,       long b) {}
```

Variadic Function Types (gcc, nginx)

```
rtx (*gen_ptr)(rtx, ...);
rtx  gen_func (rtx a, rtx b, rtx c) {}

handler_t (*mod_ptr) (...);
handler_t mod_deflate(request *r, void *data) {}
```

TyPro: Type Propagation

- C compiler prototype based on Clang / LLVM 10
- Protects indirect calls from memory corruption vulnerabilities
- Protects existing software *without modifications*
- Features:
  - Multiple architectures (x86, ARM, MIPS)
  - 100% protected binaries (using musl libc)
  - Dynamic loading of additional code
  - No performance impact

Our Approach #1: Track C Type Casts

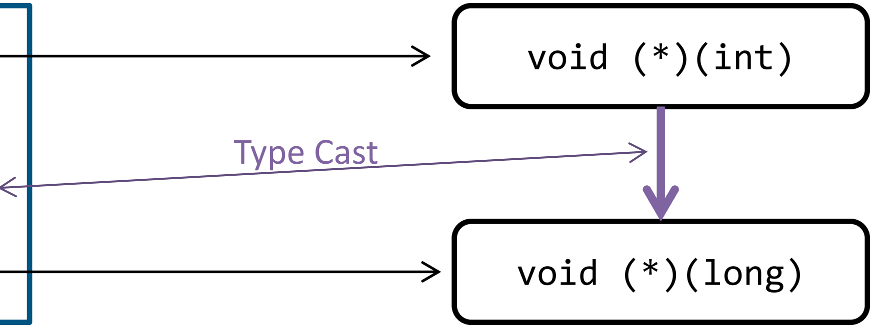
- Types of function pointers change is explicit: Type Casts

- Collect all type casts from C source
- Track reachability between function and indirect call types

```c
void function1(int x) {}
void (*func_ptr)(long);

func_ptr = (void (*)(long)) &function1;

(*func_ptr)(123);
```



Our Approach #2: Indirect Type Casts

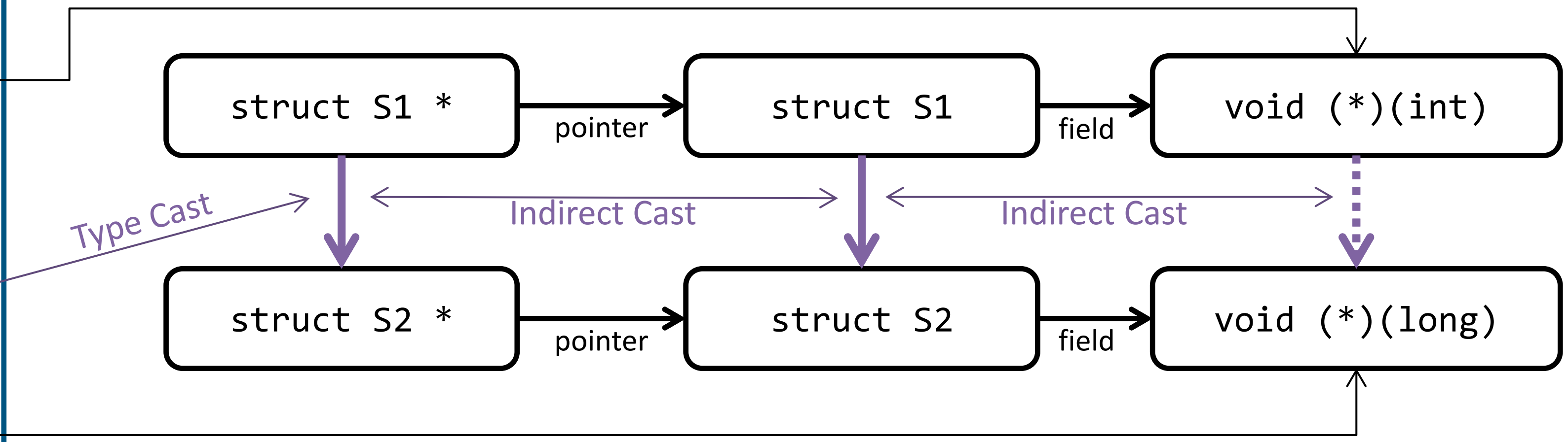
- Function pointer types can be part of composed types (structs, pointers, …)

→ We must track implicit changes (from composed types)

```
struct S1 {
    void (*field)(int) = &function;
} s1;

struct S2 {
    void (*field)(long);
} s2;

S2 *s = (S2*) s1;
s->field(123);
```



Our Approach #3: Track Types per Function

- One general cast in one function could affect all target sets

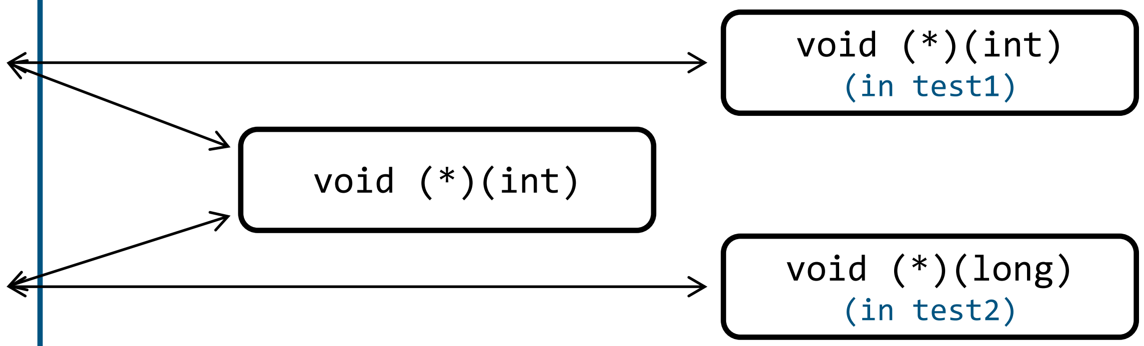
  For example: casts from/to `void*`

→ We consider types and casts *per function*

→ (and the type propagation between functions)

```
void test1() {
    void (*func_ptr1)(int) = &function1;
    …
}

void test2() {
    void (*func_ptr2)(int) = &function2;
    …
}
```



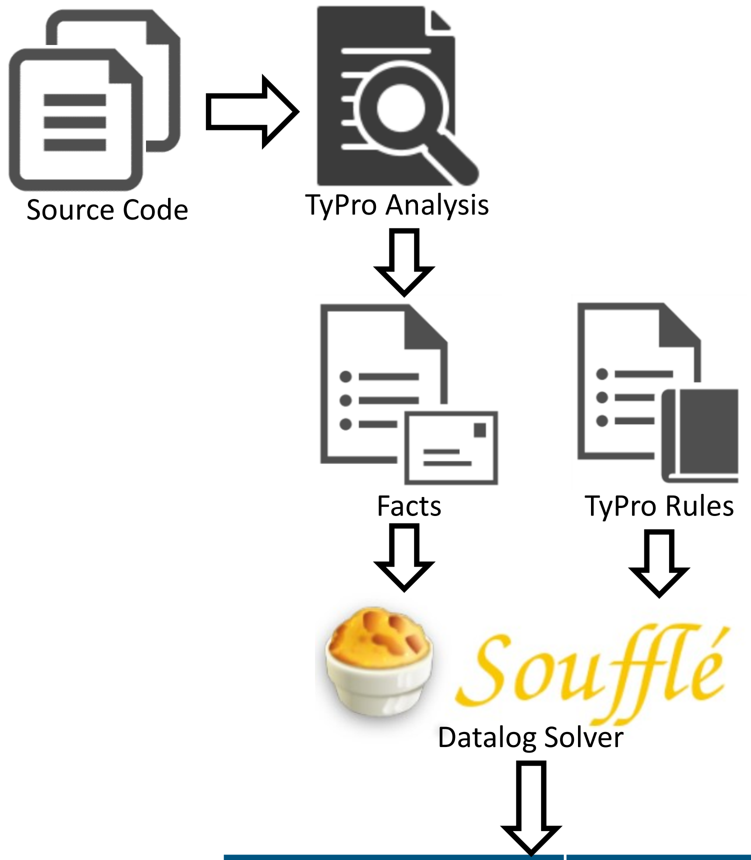
TyPro: Type Propagation

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

- Uses a formal, rule-based system to follow type casts

- Collect types and casts during compilation (“facts”)
- Compute indirect casts and reachability during linking

  (using pre-defined “rules”)
- Compute target sets for all indirect calls



Indirect Call	Target Set
(*ptr1)(…) in test1	{f1, f2}
(*ptr2)(…) in test2	{f1}
(*ptr3)(…) in test3	{f1, f3}

Enforcing Target Sets

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

- Replace function pointers by numbers (function ID)
- Replace indirect calls with `switch` over safe, direct calls

→ no unsafe indirect call remains

C code before TyPro

```
void (*func_ptr)(int);
func_ptr = &function1;
func_ptr = &function2;

(*func_ptr)(123);
```

Function Addresses

indirect calls

TyPro

Target Set = {`function1, function2`}

Function IDs

direct calls

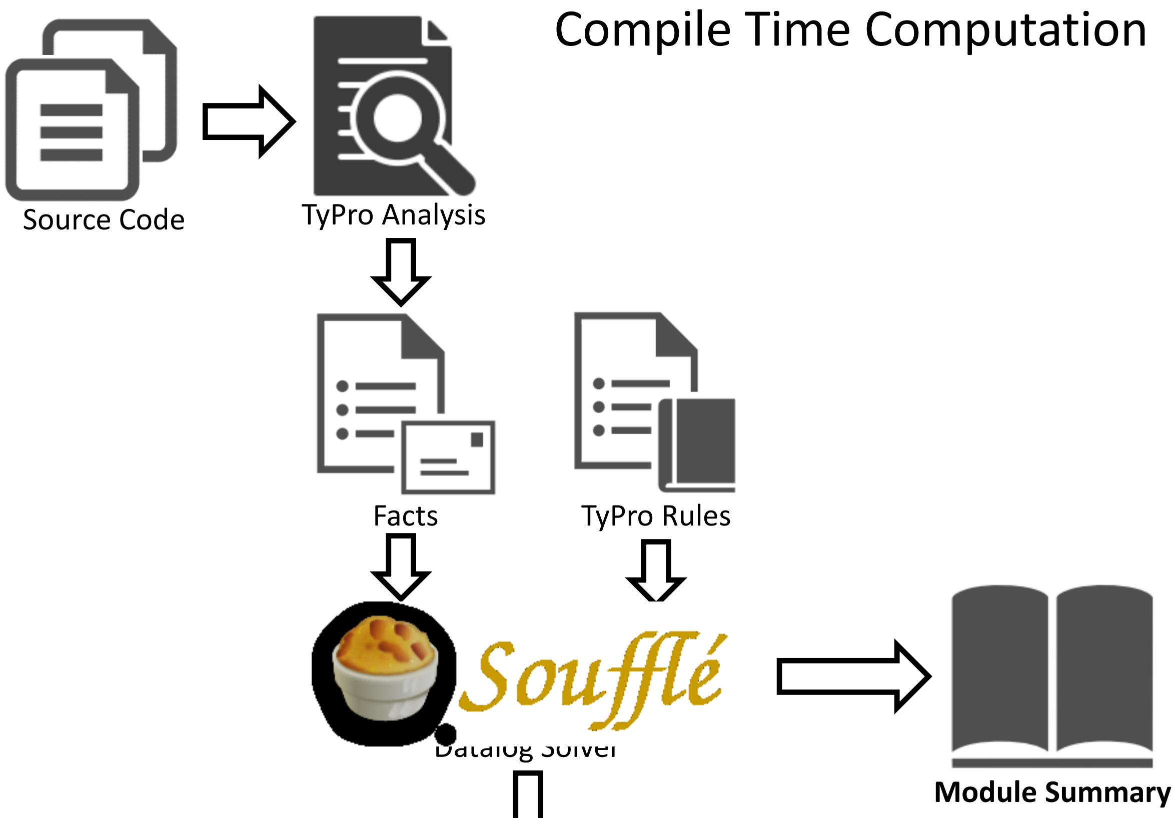
C code after TyPro

```
void (*func_ptr)(int);
func_ptr = 1;
func_ptr = 2;

switch (func_ptr) {
  case 1:
    function1(123);
    break;
  case 2:
    function2(123);
    break;
  default:
    abort();
}
```

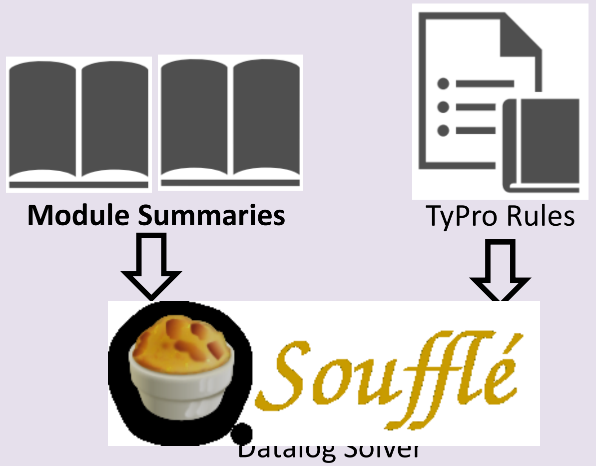
Dynamic Linking – Adding Code at Runtime

Compile Time Computation



Indirect Call	Target Set
(*ptr1)(…) in test1	{f1, f2}
(*ptr2)(…) in test2	{f1}
(*ptr3)(…) in test3	{f1, f3}

Runtime Computation



Indirect Call	Target Set Updates
(*ptr1)(…) in test1	{f3}

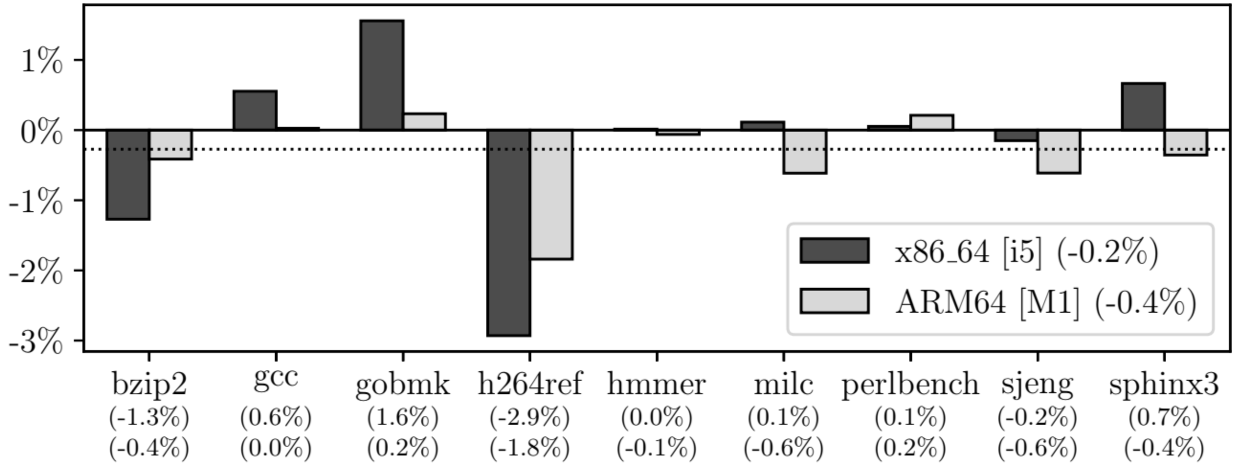
Evaluation of Type Propagation

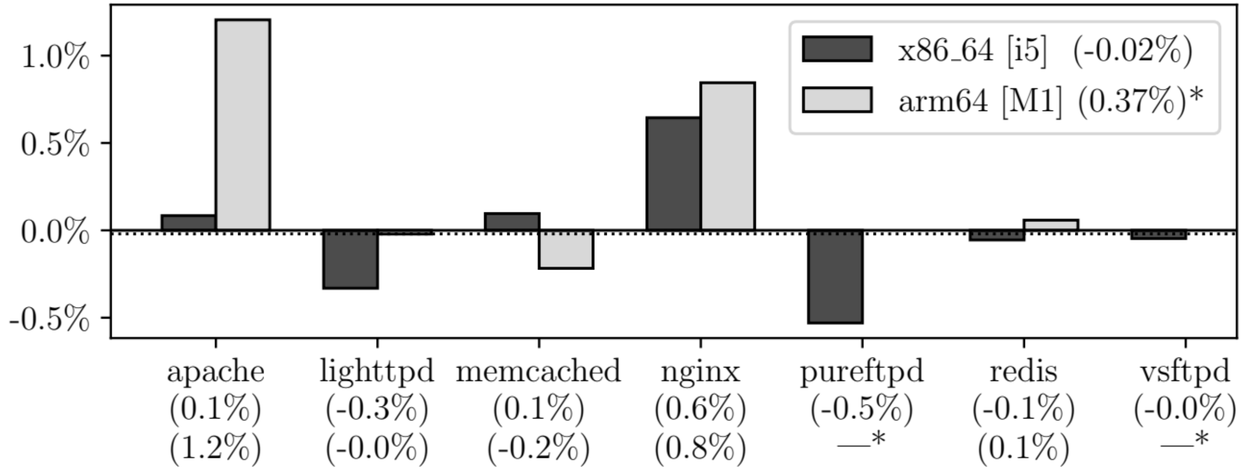
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

	functional	functional and secure	most possibly secure
	Control Flow Guard (Microsoft)	TyPro	Clang CFI
SPEC CPU 2006 is: perl, bzip2, gcc, milc, gobmk, hmmer, sjeng, h264ref, sphinx3	**SPEC CPU** up to 1786 targets	**SPEC CPU** removed ~79.0% of all targets	**SPEC CPU** removed ~79.2% of all targets **2/9 applications break**
Real-World Applications: vsftpd	**Real-World Applications** up to 2267 targets	**Real-World Applications** removed ~95.2% of all targets	**Real-World Applications** removed ~97.6% of all targets **4/7 applications break**

TyPro: Performance

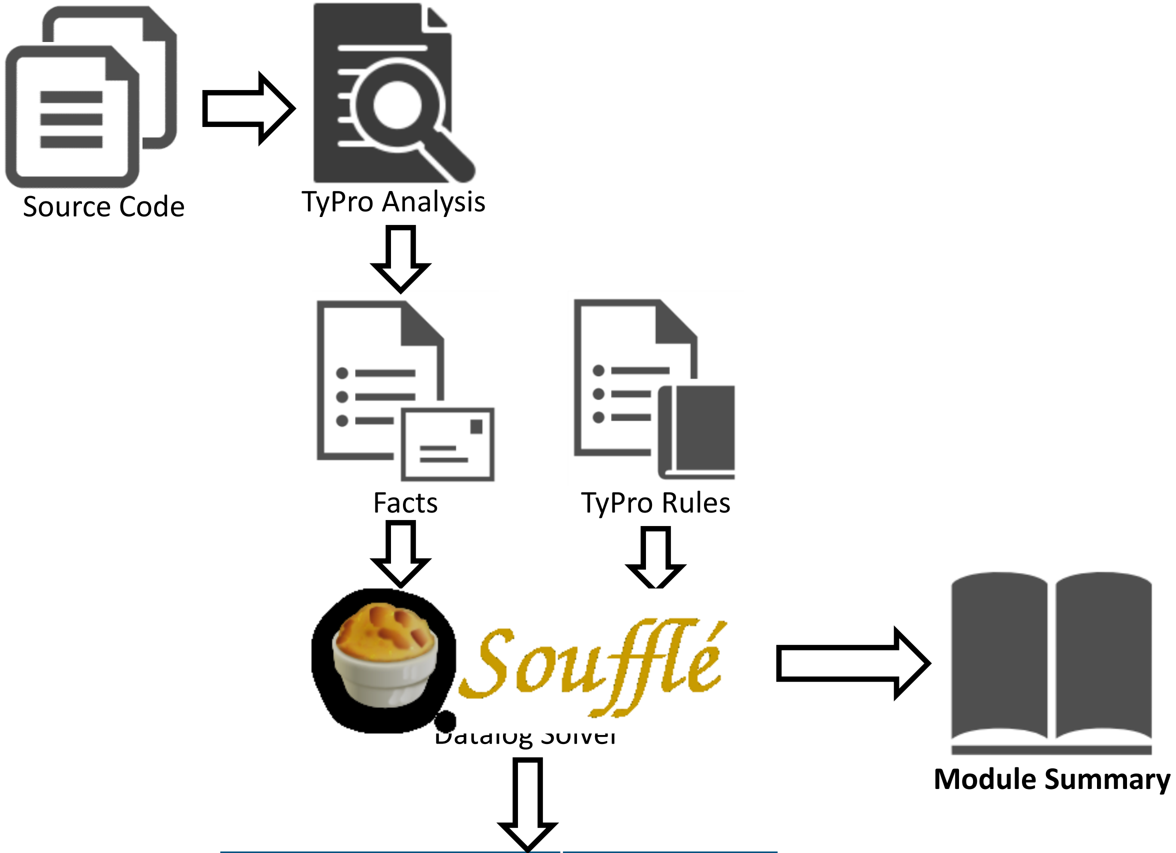
- On average, programs get (negligibly) faster
  - No checks necessary
  - More suitable compiler optimizations
- On average, programs get ~9% larger





Summary / Questions?

- Type-based analysis following casts
- Compute small yet correct target sets
- Rewrite indirect into direct calls
- Compatible, secure, and fast



Indirect Call	Target Set
(*ptr1)(…) in test1	{f1, f2}
(*ptr2)(…) in test2	{f1}
(*ptr3)(…) in test3	{f1, f3}

C code after TyPro

```
void (*func_ptr)(int);
func_ptr = 1;  // Function ID

switch (func_ptr) {
  case 1:  function1(123); break;
  case 2:  function2(123); break;
  default: abort();
}
```


Contact: markus.bauer@cispa.saarland

Code: https://github.com/typro-type-propagation/TyPro-CFI