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

void (*func_ptr)(int);
func_ptr = &function1;
func_ptr = 0xabcdef04;

(*func_ptr)(123);

void function1(int x) {...}
void function2(int x) {...}

Functions

Attacker-Controlled Code Gadgets
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

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

### 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 *, void *data) {}`
Control Flow Integrity (CFI) in Practice

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

### 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);
```

![Diagram showing type casts and function calls]

Type Cast
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)

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

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

S2 *s = (S2*) s1;
s->field(123);
```
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)
**TYPRO: Type Propagation**

- 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

<table>
<thead>
<tr>
<th>Indirect Call</th>
<th>Target Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*ptr1)(...) in test1</td>
<td>{f1, f2}</td>
</tr>
<tr>
<td>(*ptr2)(...) in test2</td>
<td>{f1}</td>
</tr>
<tr>
<td>(*ptr3)(...) in test3</td>
<td>{f1, f3}</td>
</tr>
</tbody>
</table>
Enforcing Target Sets

- 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

```c
void (*func_ptr)(int);
func_ptr = &function1;
func_ptr = &function2;
(*func_ptr)(123);
```

C code after TyPro

```c
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

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}

Just-in-time Compiler

Module Summaries

TyPro Rules
Evaluation of Type Propagation

**SPEC CPU 2006** is:
perl, bzip2, gcc, milc, gobmk, hmmer, sjeng, h264ref, sphinx3

2/9 applications break

**Real-World Applications:**

- vsftpd
- APACHE
- LIGHTTPD
- NGINX
- redis
- PureFTP
- vsftpd

**SPEC CPU**
- Control Flow Guard (Microsoft)
  - up to 1786 targets
  - removed ~79.0% of all targets
  - 2/9 applications break
- TyPro
  - functional and secure
  - up to 1786 targets
  - removed ~79.2% of all targets
  - 2/9 applications break
- Clang CFI
  - most possibly secure
  - up to 2267 targets
  - removed ~97.6% of all targets
  - 4/7 applications break

**TyPro**
- functional and secure
  - up to 1786 targets
  - removed ~79.2% of all targets
  - 2/9 applications break

**Clang CFI**
- most possibly secure
  - up to 2267 targets
  - 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 $\sim9\%$ larger
Summary / Questions?

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

### C code after TyPro

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

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

### Table: Indirect Call vs Target Set

<table>
<thead>
<tr>
<th>Indirect Call</th>
<th>Target Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*ptr1)(...) in test1</td>
<td>{f1, f2}</td>
</tr>
<tr>
<td>(*ptr2)(...) in test2</td>
<td>{f1}</td>
</tr>
<tr>
<td>(*ptr3)(...) in test3</td>
<td>{f1, f3}</td>
</tr>
</tbody>
</table>

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