Await() a Second: Evading Control Flow Integrity by Hijacking C++ Coroutines

Marcos Bajo
CISPA Helmholtz Center for Information Security

Christian Rossow
CISPA Helmholtz Center for Information Security

Abstract

Code reuse attacks exploit legitimate code sequences in a binary to execute malicious actions without introducing new code. Control Flow Integrity (CFI) defenses mitigate these attacks by restricting program execution to valid code paths. However, new programming paradigms, like C++20 coroutines, expose gaps in current CFI protections. We demonstrate that, despite rigorous standardization, C++ coroutines present new vulnerabilities that undermine both coarse-grained and fine-grained CFI defenses. Coroutines, widely used in asynchronous programming, store critical execution data in writable heap memory, making them susceptible to exploitation. This paper introduces Coroutine Frame-Oriented Programming (CFOP), a novel code reuse attack that leverages these vulnerabilities across major compilers. We demonstrate how CFOP allows attackers to hijack program execution and manipulate data in CFI-protected environments. Through a series of Proof of Concept (PoC) exploits, we show the practical impact of CFOP. We also propose defensive measures to enhance coroutine security and address this emerging threat.

1 Introduction

Code reuse attacks exploit the existing (benign) instructions within a binary to execute malicious operations without introducing new code in the system. The attacker begins by scanning the binary to identify specific sequences of instructions, known as *gadgets*, which can be chained together to mimic the intended shellcode. This approach is feasible due to the typically large size of modern programs and libraries. Then, the attacker redirects the program execution flow to ensure that the gadgets are executed sequentially, ultimately achieving arbitrary code execution. Code reuse attacks exist in multiple forms, such as *Return-Oriented Programming* (ROP) [50], *Jump-Oriented Programming* (JOP) [6], *Sigreturn-Oriented Programming* (SROP) [7], and *Counterfeit Object-Oriented Programming* (COOP) [45].

Control Flow Integrity (CFI) [2] schemes defend against code reuse attacks. They aim to ensure the program's execution flow is valid during runtime. For this, *fine-grained* CFI schemes extract the program's Control-Flow Graph [2], which describes the intended possible code paths. Fine-grained CFI schemes then enforce the rule that only such valid code paths can be taken. *Coarse-grained* CFI schemes relax these restrictions by limiting control transfers to the start of any function only; while offering reduced security, they have negligible performance overheads [18]. Examples are Intel CET [27] and Microsoft Control Flow Guard [35], which are widely deployed in Linux and Windows, respectively.

CFI defenses systematically cover those programming paradigms that were current at the moment of their creation. Consequently, they protect vulnerable function pointers from known concepts like callback functions or C++ virtual tables. However, naturally, CFI may only protect those pointers it knows. Programming languages evolve and add new paradigms, often neglecting their security consequences. Failing to consider such new paradigms may leave function pointers unprotected, undermining CFI's security.

In this paper, we study one such recent feature introduced to C++20: *coroutines* [29]. Coroutines are functions that can be suspended and resumed during their execution. Present in many programming languages such as Python, C#, Golang, and Kotlin (throughout the *async/await* pattern); they are most typically used to implement asynchronous code (such as non-blocking I/O operations) and algorithms used to lazily generate infinite sequences of data. Despite their relatively recent introduction in C++, all major compilers already support them in both Windows and Linux, including GCC (since GCC 10), MSVC (since MSVC 19), and Clang (since Clang 8) [17]. Hence, C++ coroutines are already part of C++ software, and were deployed to over 400 million devices even *before* becoming a C++ standard [1]. We found coroutines in 130+ unique popular GitHub repositories (500+ stars), of which 24 have more than 10k stars. They have become a fundamental paradigm for implementing databases (e.g., ScyllaDB [47], RocksDB [23], ArangoDB [4]), popular libraries (e.g., asio [13]), and Windows software (e.g., PowerToys [37], the Windows terminal [38]). Considering that compiler sup-

port was only recently added, we predict that the adoption of coroutines will be even more widespread in the near future.

Unfortunately, despite having undergone a rigorous standardization effort, we find that C++ coroutines undermine all existing CFI schemes. We show that C++ coroutines can be exploited in code reuse attacks despite state-of-the-art CFI defenses. This novel code reuse attack, which we call *Coroutine Frame-Oriented Programming* (CFOP), leverages the insecure C++ coroutines implementation common to the three major compilers both in Linux and Windows. In a nutshell, we find that any program using C++ coroutines stores critical data in heap memory. This data includes (1) memory addresses pointing to program code, which are used during the coroutine execution as call and jump destinations and which are CFI-unprotected; (2) pointers to coroutine heap structures and other coroutine internal data; and (3) the value of arguments, variables and other data accessed during a coroutine execution and that usually would only be found in the stack. In all these cases, this data is in writable memory and is subject to being modified by an attacker capable of corrupting the program memory. As a result, attackers can launch not only data-only attacks that result in executing one or multiple coroutines with corrupted data (e.g., hijacking the argument to a call to *system()* that was already present in the coroutine code); but also call arbitrary functions with controlled parameters (e.g., a function in a linked shared library), while on a CFI-protected program.

To the best of our knowledge, we are the first to study C++ coroutines from a security perspective and to highlight how they can be exploited to bypass CFI protections. We implement a series of Proof of Concept (PoC) exploits that showcase CFOP attacks against sample programs using coroutines. Moreover, we prepare a PoC attack against two real programs that use coroutines, exploiting ScyllaDB (13.2k+ stars on GitHub) and SerenityOS (30k+ stars), compiled with Intel CET enabled. To demonstrate the CFOP principles, we assume a simple memory-corruption vulnerability in ScyllaDB, equivalent to the ones needed to launch any other code reuse attack such as ROP, whilst we port the CVE-2021-4327 vulnerability to SerenityOS. We show that using CFOP, we can execute any functions and system calls with arbitrary arguments. These attacks work when attackers are bound to only call the start of functions due to coarse-grained CFI and even extend to fine-grained CFI schemes as none of them accounts for coroutines. Finally, we propose a series of implementation improvements and defenses that protect against CFOP attacks in the future.

To summarize, our contributions are:

- We study C++ coroutines from a security perspective, illustrating how an attacker can exploit all their existing implementations for CFOP code reuse attacks.
- We show that CFI schemes do not consider C++ internal semantics and structures of C++ coroutines. For this, we find that none of the widely available and academic-proposed CFI schemes sufficiently cover coroutines.
- We showcase our attack techniques by implementing exploits for two real CFI-enabled x86_64 Linux programs, and a series of PoC exploits that showcase different attacks against sample coroutine programs.
- We propose fundamental changes in the C++ coroutines implementation to protect them against CFOP attacks.

2 Background

This section describes the most common use cases for C++ coroutines and provides a high-level overview of their functionality. Additionally, we will present background information on how CFI protects programs.

2.1 Coroutines

The typical workflow for a normal function in C++ is that (1) the function is *called* by some other caller function; (2) the function is executed from start to finish; and (3) the function *returns*, reinstating the execution to the caller function.

In turn, coroutines are like normal functions, but they can also be suspended during their execution so that some other external code gets executed. They can later resume execution from where they were previously paused. The typical workflow of a coroutine is as follows: (1) the coroutine is *resumed* (instead of called), starting its execution from the beginning; (2) during execution, the coroutine gets *suspended*, pausing itself and returning to the caller function; (3) the caller function can then resume the coroutine again, continuing the execution from the point it was paused; and (4) the coroutine *returns* to the caller function once the complete coroutine body is executed. Once the coroutine returns for a final time, it can never be resumed again. Note that steps (2) and (3) may be repeated any number of times, allowing the coroutine to be suspended and resumed as needed during its execution. We further explain the internal implementation of the resumption and suspension of coroutines in § 3.2.

2.1.1 Suspension Point

Developers decide when a coroutine is suspended by setting *suspension points* (SPs). There may be one or more SPs in the coroutine, each of which can be reached multiple times (e.g., when in a loop). The coroutines API defines three operators, which implicitly set an SP and allow for customizing the coroutine's actions when reached [29]. *co_yield* <*value*> returns a value to the caller and suspends the coroutine (usually, to be resumed later). *co_return* returns and suspends the coroutine for a final time. Finally, *co_await* <*Awaitable*> evaluates an *awaitable* object.

```
task_fib fibonacci_gen() {
    int f0 = 0, f1 = 1;
    for(int ii = 0;; ++ii ){
        co_yield f0+f1;
        int temp = f1;
        f1 = f0 + f1;
        f0 = temp;
    }
}
void main(){
    task_fib generator = fibonacci_gen();
    std::cout << generator(); //prints '1'
    std::cout << generator(); //prints '2'
    std::cout << generator(); //prints '3'
}
```

Listing 1: Program using a coroutine to print three numbers of the Fibonacci series.

```
task_aw write_op(Socket s){ //c3
    write(s, msg, msg.length()); //Send message
}
task_aw client_run(Socket s) { //c2
    //Carry out multiple operations on client
    co_await write_op(s);
    co_await write_op(s);
}
task_aw launch_server() { //c1
    while(true){ //Listen for clients
        Socket s = servSocket.accept();
        co_await client_run(s);
    }
}
void main() {
    std::coroutine_handle<> t = c1().handle;
    t.resume();
}
```

Listing 2: Server program that uses *co_await* to address multiple client connections, executing three nested coroutines.

Next, we describe two of the most common use cases where a coroutine-based implementation is ideal, showing how to use *co_yield* and *co_await* respectively.

Generators: These are functions that continuously generate a sequence of elements—one each time the coroutine is resumed. Listing 1 shows an example of generator, where a coroutine uses the operator *co_yield* to return the values to its caller. The coroutine is an infinite loop, which is resumed every time we need a new number from the Fibonacci series and then suspended until its next call.

Asynchronous Jobs and Awaitable Coroutines: The operator *co_await* is used to transfer the control flow throughout coroutines. Invoking *co_await* suspends the coroutine and evaluates an *awaitable* object. Inside this *awaitable*, the developer implements an *awaiter* that will dictate the actions to undertake next via three main functions executed in order: (1) *await_ready()*, that determines whether the coroutine should suspend itself; (2) *await_suspend()*, that evaluates the current coroutine and decides how the control flow should continue after it is suspended; and (3) *await_resume()*, that determines the return value of the *co_await* call. Notably, the implementation of *await_suspend()* is highly variable between programs, returning the control flow to the caller of the current coroutine, resuming other coroutines, or executing some code.

We commonly find coroutines using *co_await* to implement asynchronous jobs. Here, the coroutine calls *co_await* on an *awaitable* object with an *awaiter* whose *await_suspend()* function executes the asynchronous code; then resumes the original coroutine. Another common use case for using *co_await* is *awaitable coroutines*. In this case, the coroutine calls *co_await* on another coroutine (which must implement an *awaitable*); the function *await_suspend()* in the *awaiter* resumes the new coroutine to be executed. Listing 2 shows a simple program using awaitable coroutines, where coroutines are nested, resuming and suspending each other.

Coroutine Schedulers: Another common use case for coroutines is implementing programs that rely on cooperative multitasking. In this model, the program handles multiple concurrent tasks, and a centralized scheduler is responsible for allocating CPU time among them. Each task is represented as a coroutine, which can yield control back to the scheduler at any SP. The scheduler—typically implemented as a queue or list of suspended coroutines—determines which coroutine to resume next. When a coroutine runs, it executes until it reaches a suspension point, at which point it re-enqueues itself in the scheduler's queue. This approach enables multiple tasks to run within a single thread, mimicking CPU context switching at the application level.

2.1.2 The Coroutine Handle, Promise and Awaiter

C++ coroutines are implemented in a coroutine object, usually referred to as the *task*. This *task* (e.g., *task_fib* in Listing 1) contains every element in the coroutine, including the *promise* and *awaiters*, which we explain next; and a key element—the *coroutine handle*—used to directly interact with the coroutine. Through it, we can resume the coroutine (*handle.resume()*), and destroy it, freeing its memory (*handle.destroy()*).

The *promise* is an object present in every coroutine that (1) stores the coroutine return value and (2) controls the coroutine behavior. When a coroutine returns or is suspended, it saves its return value (set via *co_yield* or *co_return*), if any, in its promise object. This value can then be subsequently queried at any moment from a function or another coroutine.

The *promise* also controls the coroutine behavior via two functions that return an *awaiter*. Firstly, *initial_suspend()* decides the coroutine behavior after invoking it the first time. Returning *suspend_never* implies that the coroutine has an *eager start*; as soon as we create a coroutine object, it is automatically resumed for the first time. Returning *suspend_always* dictates a *lazy start*; we need to explicitly resume the coroutine after a coroutine object is created for it to run. Secondly,

final_suspend() decides what happens after the coroutine reaches the final SP; i.e., it can resume another coroutine, return a special value, or run any other code. Again, there are two settings: *suspend_never* assumes that the coroutine will never be used (not even for accessing the return value) after the final SP, so the coroutine object is destroyed; and *suspend_always* assumes that the coroutine may be accessed after the final suspension point, so it is just suspended but not destroyed. In addition, programs commonly use a custom *awaiter* with *final_suspend()*. Inside its *await_suspend()* method, developers usually place functionality to run after the coroutine reaches its final SP.

We show in Appendix A an example of *task* definition that corresponds to *task_aw* in Listing 2. Note that this is a simplified example intended for illustration purposes; in a real program, the coroutine implementation would typically include additional elements (e.g., execution in multiple threads).

2.2 Control Flow Integrity

CFI schemes aim to ensure that the program's execution follows the *intended* flow. CFI strives to prevent attackers from diverting the control flow to other areas, potentially executing ("reusing") arbitrary code by hijacking indirect pointers.

We distinguish between *forward-edge* and *backward-edge CFI*. Forward-edge CFI protects indirect calls, including function pointers whose branch or call target values are computed at runtime or other indirect control-flow transfers, such as virtual functions. Backward-edge CFI protects the return addresses of functions that are saved in the program stack.

Irrespective of the CFI direction, schemes operate on different levels of protection. *Fine-grained* CFI closely approximates the program's Control Flow Graph [2], while *coarse-grained* CFI is less precise and works with relaxed versions of the graph. While fine-grained CFI introduces more restrictive controls over the control flow and thus greater security, coarse-grained CFI strives to find a balance between the level of protection and the overhead introduced by the scheme [44].

The attacks described in this paper consider state-of-the-art coarse-grained and fine-grained CFI schemes, as we detail in § 4.2. These include Intel's Control-flow Enforcement Technology (CET) [27], a form of coarse-grained CFI with both forward and backward CFI protections; Control Flow Guard (CFG) [35], a more fine-grained CFI scheme widely available in Windows systems; and Clang CFI [14], a fine-grained scheme readily incorporated in the Clang/LLVM compiler. These policies prevent standard code-reuse techniques based on indirect calls (like ROP and JOP) and vtable hijacking (like COOP). Notably, Intel CET's forward-edge protection, Indirect Branch Tracking (IBT), enforces that functions can only be executed from their entry point, i.e., it forbids the common code-reuse attack technique of jumping to the middle of functions. This principle is also fundamental and shared by all fine-grained schemes, including CFG and Clang CFI.

3 Attacking Coroutines

With our understanding of how C++ coroutines work from a high-level perspective, we will now explore how they work internally and how this can be exploited. For this, we will study the low-level details of how coroutine code is compiled and how their data is represented in memory at runtime.

Throughout this section, we will discuss the code generated by the (at the time) newest compiler versions GCC 13.2, Clang/LLVM 18.1.8 and MSVC 19.39.33523. We will consider code optimized using *-O3*, although our techniques also work with other optimization levels.

3.1 Threat Model

We consider the typical threat model of CFI schemes, assuming a malicious actor that can (1) read arbitrary memory data to learn the memory layout and bypass ASLR protections [31, 40]; and (2) write arbitrary data into arbitrary sections of memory [34]. Specifically, we allow an attacker to abuse one or more information leak vulnerabilities to disclose any necessary memory addresses and data, and to abuse a *single* memory corruption vulnerability that results in a memory write with attacker-controlled data. We assume that the vulnerable program has no pages marked both as writable and executable ($W \oplus X$) and that the attacker cannot directly modify the value of the program registers.

Additionally, we assume that attackers aiming for code-reuse attacks can only reuse entire functions. That is, we assume that CFI enforces that functions must be executed starting from the beginning, as in Intel CET or CFG. Under this setting, we aim to bypass both coarse-grained CFI schemes (by leveraging the restricted but still controllable jump and call targets) and fine-grained CFI implementations (as they do not protect C++ coroutines; see § 4.2). Note that we focus on bypassing forward-edge protections since backward-edge CFI is orthogonal to our coroutine-based attacks (which do not tamper with return addresses).

Finally, we assume the program features at least one C++ coroutine, which must be suspended or running when the attacker exploits the arbitrary write vulnerability.

We further address relaxed versions of this threat model in Appendix E (i.e., what happens if the attacker can only overflow a buffer instead of an arbitrary memory write).

3.2 Coroutine Implementations

To evaluate the security of coroutines, it is vital that we understand how they are implemented. Once the compiler recognizes a coroutine, it performs a series of transformations that transparently add the code necessary for the coroutine functionality (e.g., suspending). In this subsection, we thus systematically explore these transformations, identifying potential security risks that we will explore in § 3.3.

3.2.1 The Coroutine Frame

Unlike in other programming languages (e.g., goroutines in Go), coroutines in C++20 are *stackless*.

On the one hand, *stackful* coroutines can suspend themselves at any point during their execution (e.g., the coroutine may call another function and suspend itself from there.) Therefore, they have their own program stack, which keeps track of any arguments, local variables, and called functions during the coroutine execution.

On the other hand, C++20 stackless coroutines may only be suspended from the top-level coroutine function. This means that if the coroutine calls a function during its execution, it may not suspend itself until the function returns. This has low-level memory implications since coroutines no longer need to save the call stack and the variables from other functions, only their local variables and arguments. In this case, it is not optimal to assign a stack for every coroutine; all the data that the coroutine needs to preserve its state after suspending is allocated dynamically in the heap.

This chunk of heap data, unique for every coroutine, is known as the *coroutine frame*. The frame is, in essence, the compilation result of the coroutine *task* object—a struct that holds the coroutine state during its whole lifetime, just like all data needed during a normal function execution is stored at the stack. The compiler decides the exact frame structure, preparing a *frame template* that defines the type and organization of the data needed for every coroutine.

Figure 1 shows the most relevant data in the coroutine frame. The *resume pointer* is a pointer to the function responsible for resuming the coroutine after a suspended state (the *resume stub*, see § 3.2.2). The *destroy pointer* points to the function responsible for deleting the coroutine frame from the heap, freeing its corresponding memory, after the coroutine returns (the *destroy stub*, see § 3.2.2). The *promise object* incorporates the developer-defined *promise* object. The *parameters* store any arguments that the coroutines have been passed from a function or another coroutine. Arguments passed by value get the value copied into the frame, and those passed by reference get the pointer stored in the frame. *Local variables* are variables that are used during the coroutine execution (which in normal functions would usually be found in the stack). Finally, the *coroutine index* is a numerical value that indicates the last suspension point (SP) reached by the coroutine.

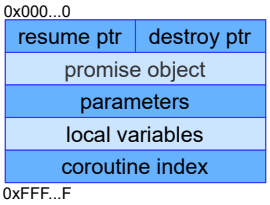

Figure 1: Coroutine frame structure in C++20.

Parameters and local variables are not simply saved into the frame when the coroutine suspends; they actually reside in the heap all the time (before, during, and after the coroutine suspension). The compiler arranges them such that: (1) integer local variables are stored by value in the frame; (2) if the coroutine allocates an object (e.g., via *new()*), then a heap chunk is allocated for the object, and only its pointer is saved in the frame; and (3) if the coroutine uses a variable that would commonly be stack-based (e.g., an array of chars), then the compiler reserves enough space inside the frame for the array, turning the array from a stack-based variable to heap-based. Note that, just like with any other function, if an object is allocated, its destructor is automatically called after the coroutine returns (i.e., before calling *final_suspend*).

3.2.2 Coroutine Lifetime

Along with a coroutine frame for every declared coroutine, the compiler generates a series of function *stubs* responsible for managing the coroutine frame behind the scenes. These stubs are automatically incorporated into the program and are invoked when transferring the control flow from a function to a coroutine or between coroutines.

Currently, compilers generate three stubs, which are called by the function using the coroutine, as we illustrate in Figure 2. The *creation stub*, invoked when a coroutine is used for the first time, is responsible for allocating memory in the heap for the coroutine frame and initializing the *resume* and *destroy* pointers. The coroutine then gets suspended, returning back to the caller function. The *resume stub*, invoked when resuming the coroutine, evaluates the coroutine index and makes a *direct* jump to the corresponding SP. Finally, the *destroy stub*, invoked for destroying the coroutine frame, frees the frame memory. When a coroutine co_awaits another coroutine, these three stubs are used together with the evaluation of the coroutine *awaiter* that we described in § 2.1.1. We illustrate the combined process in Figure 8.

The *resume* and *destroy* pointers point to the *resume* and *destroy* stubs, respectively. When the coroutine is resumed or destroyed, the program makes an *indirect* jump using them. We include, for reference, in Appendix B, the simplified assembly code generated for creating, resuming, and destroying coroutine *c1* from function *main* corresponding to Listing 2.

One may wonder why coroutines follow such a complex structure—having two dynamic pointers to the *resume* and *destroy* stubs—instead of using a direct jump to the corresponding stub address. Indeed, the *resume* and *destroy* pointers do not change once the coroutine frame is first created. However, this architecture is necessary to support many of the use cases of coroutines. For example, in many cases, a coroutine will be created and saved for later execution, then resumed from some other function blindly, using the handle, without knowing which type of coroutine is being resumed. Thus, the compiler may not always know which coroutine is getting resumed or destroyed; hence, the frame must contain information about how to resume and destroy the coroutine.

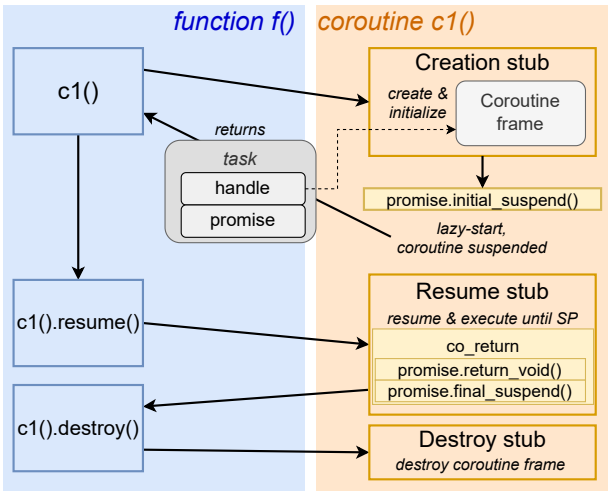

Figure 2: Coroutine lifetime (when executed from a function).

```
1  task coro(char* arg){
2      //SP1
3      char arr[10];
4      std::vector<int> vec;
5      co_await some_task;
6      //SP2
7      for(int ii=0; ii<3; ii++){
8          vec.push_back(ii);
9          co_await some_task;
10         //SP3
11     }
12     char arr2[] = "a_string";
13     std::cin.getline((char*)arr, 100);
14     /*arr, arr2 and vec are used somehow*/
15     system(arg);
16 }
```

Listing 3: Simple coroutine with three SPs and a vulnerability.

3.3 Coroutine Attack Primitives

We will now use our knowledge of coroutines' low-level internals to present their weaknesses and develop exploitation techniques. We strive to achieve arbitrary code execution (i.e., running any number of coroutines or functions with controlled arguments) while respecting the attacker capabilities in the threat model at § 3.1. To this end, we first introduce two general *coroutine attack primitives*, (1) frame manipulation and (2) frame injection. These two primitives will allow us to craft concrete attacks in the following subsections.

3.3.1 Frame Manipulation

The coroutine frame holds the current state and data from the coroutine, therefore becoming a prime target if we aim to influence the control flow. Since the frame is allocated at the heap (marked with R+W permissions), it may be modified by the attacker's arbitrary write primitive.

Overwriting an existing frame is a very valuable primitive, as the attacker may target its *resume* and *destroy* pointers, hijacking subsequent calls to *resume()* and *destroy()*, or modifying the data some coroutine is using. We will elaborate on both these scenarios in § 3.4 and § 3.5.

The frame itself does not incorporate any security mechanisms against memory corruption. Moreover, the default malloc implementation in glibc (*ptmalloc* [25]) does not incorporate any security mechanisms to prevent heap chunks from overflowing (like stack canaries [16, 52] to prevent stack corruption across stack frames). Therefore, it is not only possible to corrupt the whole frame with a single contiguous memory write operation but also to overflow the corresponding heap chunk of a frame, proceeding to overwrite any adjacent memory (and frames) using a single memory write.

3.3.2 Frame Injection

It is not only possible to modify an existing frame; an attacker may inject their own frames into the heap that are as valid as any other created by the *creation stub*. The only thing needed is that some coroutine references the injected frame, so that the injected frame can be resumed or destroyed. At that point, subsequent calls to *resume()* and *destroy()* on the injected frame will use attacker-controlled code pointers.

As we explain in § 3.5, every coroutine program features hijackable pointers we can redirect where needed, that we can use to point to injected frames. In addition, functions using coroutines save the coroutine handle in the stack, where it is subject to being overwritten either by a stack-based buffer overflow or an arbitrary write primitive before the next use.

3.4 Data-Only Attacks

Modifying the value of variables and arguments inside the coroutine frame can result in the coroutine using this data at runtime. Although such data-only attacks (DOAs) do not directly hijack the control flow, they can influence the program execution. Data-based attacks have shown to be a useful primitive in code-reuse attacks [11, 26, 41].

For example, consider the coroutine in Listing 3. This vulnerable code—a buffer overflow in line 13—allows for several DOAs using frame manipulations. Here, attackers can change the number of times *some_task* is *co_await*-ed, and manipulate the command executed by *system()* by modifying *arg*.

However, it is essential to consider how the compiler uses the variables in the frame to understand if they can actually be leveraged in an exploit. During our testing, we have extracted the following compiler behavior during coroutine compilation (assume *-O3* optimization): (1) Arguments are copied into the frame during the *creation stub*, so their value is present from the first coroutine resumption. (2) A variable is initialized in the frame at the same SP where the variable is first initialized. (3) Stack-based variables are saved into the heap

Table 1: Variables in the frame of the coroutine in Listing 3. On a given SP, the variable may be uninitialized (`U`), getting initialized (`I`), or its value can be hijacked (`H`).

Variable	Creation stub	SP1	SP2	SP3
arg	`I`	`H`	`H`	`H`
arr	`U`	`U`	`U`	`I`
vec	`U`	`U`	`I`	`H`
ii	`U`	`U`	`I`	`H`
arr2	(Stack-based)			

frame; heap-based objects get a pointer to the object saved in the frame. (4) If a variable is declared and used only at the last SP, it is only saved at the stack since the coroutine does not need to remember its value. (5) The compiler considers coroutine parameters and local variables one group and aligns them in the frame according to common buffer overflow protections: overflowable buffers (e.g., an array) are always positioned in higher memory addresses, preventing them from ever overwriting a variable in the event of an overflow. (6) The *coroutine index* is always positioned after any variable; it can always be overwritten with an overflow.

Therefore, as a general rule, modifying a variable in the coroutine frame alters its value during the next SPs if it is overwritten after the SP where it is initialized. However, we are free to modify the value of coroutine arguments at any point. For our example in Listing 3—a coroutine with three SPs, one argument *arg* and four local variables—this implies that (1) *arg* can be hijacked at any SP and then resume the coroutine; (2) overwriting *arr*'s frame value never leads to any change during execution; (3) *vec* is a pointer, which is only put in the frame during SP2 (when it is first used) and can only be hijacked for SP3; (4) *ii*'s value can be hijacked for SP3 (e.g., we can run the loop as many times as we want); and (5) *arr2* is only initialized in SP3, so it is stack-based. We summarize when variables can be hijacked in Table 1.

Since timely modification of variables in specific SPs is relevant for DOAs, attackers may benefit from setting the SP value themselves. Just like variables and pointers in the frame are writable, an attacker can set the coroutine index to an arbitrary value. By modifying the index, attackers can control at which SP a coroutine resumes, as the *resume stub* uses this index to decide at which SP the coroutine is resumed.

3.5 Control-Flow Hijacking

Control-flow hijacking using coroutines is possible using a concept we dub *Controlled Frame Pointers* (CFP). We define as a CFP any object which the attacker can control (overwritable), and that is directly or indirectly used to issue a *resume()* or *destroy()* call. We found three generic sources for such CFPs: (1) the coroutine frame (§ 3.5.1), (2) schedulers (§ 3.5.2), and (3) the *awaiter* object (§ 3.5.3).

3.5.1 Attacking the Pointers in the Frame

The weakest point in the frame is the *resume* and *destroy* pointers. Both of these pointers qualify as a CFP, as they are directly used to issue *resume()* and *destroy()* calls. The insecurity of these function calls continues the history of vulnerable code pointers found in writable memory, such as the return addresses at the stack (that enable Ret2Libc [15], ROP attacks), C++ virtual pointers, or code pointers in the Global Offset Table (GOT) [24]. Any such pointer has needed its own protection scheme to protect against memory corruption attacks, introducing canaries and Shadow Stacks [12,27], CFI for virtual pointers [43,53], and RELRO for the GOT [30].

An attacker could modify the pointers in the coroutine frame either by (1) directly overwriting them via an arbitrary write primitive, (2) overflowing a buffer in the heap, which may even be a stack-based buffer used in another coroutine (turned into heap-based), or by (3) escalating another (more limited) vulnerability within the program to corrupt the pointer (e.g., a limited overflow in some other coroutine frame that corrupts a heap-based object, leading to a controlled write). Either way, these pointers are then used in *call* and *jmp* operations when jumping to the *resume* and *destroy* stubs, as we described in § 3.2.

In addition to the *resume* and *destroy* pointers, overwriting any function pointers used by a coroutine—that are saved in the frame—can lead to execution flow hijacking.

3.5.2 Attacking CFPs in Schedulers

A common design pattern in coroutine programs is a scheduler that maintains a queue of awaiting coroutines, as described in § 2.1. The scheduler calls *resume()* on any enqueued coroutine, and *destroy()* when finished. Each coroutine runs until it reaches an SP, then it enqueues again in the scheduler queue.

Each of the frames in the scheduler features at least two CFPs since every coroutine frame in the scheduler queue will be resumed and destroyed at some point: The CFP corresponding to the *resume* pointer will be called once or more (since the coroutine may be resumed multiple times), and the CFP corresponding to the *destroy* pointer is called exactly once.

3.5.3 Attacking Awaiter CFPs in Awaitable Coroutines

Another common design pattern in coroutine programs is finding coroutines that directly transfer the control flow from one to another without needing a scheduler. In these programs, we use the operator *co_await* and the coroutine *awaiters*, which we explained in § 2.1. Depending on how the operators are used and the implementation of the methods inside the *awaiters* (e.g., in the *await_suspend()* function), we can find that the compiler generates fundamentally different code. We have detailed these differences in Appendix C.

Independently on the implementation differences, execution transfer between coroutines is commonly based on *con-*

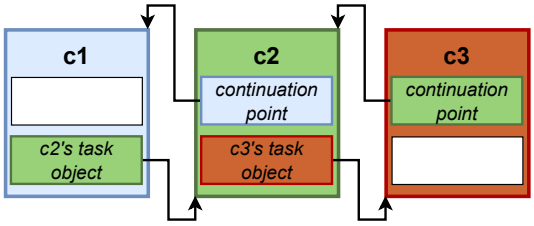

Figure 3: Internal objects that point to previous and next coroutine frames in a program with three nested coroutines.

tinuation points. If we have a coroutine *c1* and we want to transfer execution flow to a lazy-start coroutine *c2*, and we want coroutine *c1* to be resumed back after *c2* finishes, the process goes as follows: (1) *c1* co_awaits *c2*. The frame of *c2* is created and then suspends due to its lazy start. (2) *co_await* evaluates the awaitable from *c2*, running *await_suspend()*. During this function, *c1* stores inside *c2*'s *promise* the handle to *c1*'s frame—the so-called *continuation point*. Then, we resume coroutine *c2*. (3) *c2* completes its execution and reaches the final SP. At this point, *final_suspend* is evaluated with a custom awaiter. Inside, *await_suspend()*, we check whether a *continuation point* has been set (maybe the coroutine was called from a function, in which case it would not have one). If there is a *continuation point* set, we call *resume()* on it.

A critical observation from this control flow transfer strategy is that the frame will contain two internal pointers, as shown in Figure 3. The first corresponds to the *continuation point*—the coroutine to resume after the current one reaches its final SP. Secondly, we find a pointer to the coroutine resumed inside the other coroutine. This pointer exists because if we *co_await c2* from coroutine *c1*, then *c2*'s *task* object is a local variable that must be stored inside *c1*'s frame.

This second *task* pointer is generally used to destroy the coroutine after it returns. We find that, by convention, many real-life coroutine programs resort to placing a *destroy()* call in the destructor of the *task* object. Since the scope of the *task* object is limited to the *co_await* call, the destructor of the *task* is called right after the *co_await* call. At the assembly level, we then find an indirect *call* instruction to the *destroy* pointer of the awaited coroutine frame.

Considering this architecture, we identify that, firstly, continuation points are CFPs. If the awaiter uses continuation points to issue a *resume()* or *destroy()* call, attackers may hijack the execution flow by setting the continuation point to an area of memory where an arbitrary *resume* or *destroy* pointer is placed. Secondly, *task* objects saved in coroutine frames are also CFPs. If a coroutine saves in the frame the *task* object of another called coroutine *and* then uses it for subsequent operations (e.g., to destroy the coroutine), attackers may hijack the execution flow just as with continuation points.

Finally, we would like to point out that the existence of a CFP does not necessarily always mean that the attacker may be able to leverage it—just like a vulnerability may not be able to be exploited. Awaitable coroutines are a perfect example of this; we can take the program in Listing 2 as reference. If we suppose an attacker can overwrite *c2*'s frame, we quickly realize that the CFP corresponding to *c3*'s *resume()* call is out of the attacker's control. As we detailed previously, we need to wait at least until the next SP after a variable is initialized in order to hijack it successfully in the frame. Therefore, we cannot overwrite the frame if it has not been created yet or if the variable is being created at the same SP. In our example, *c2* creates the *c3* coroutine in the first SP, resuming it in the same SP. For this reason, in an awaitable coroutines scenario, it is often not possible to count the initial call to *resume()* as a useful CFP. On top of this, not every *task* object saved in the frame qualifies as a CFP—some programs return *suspend_never* from *final_suspend()*, immediately destroying the coroutine, in which case there would later not be a call to *destroy()* from the destructor. However, most awaitable coroutine programs return values or run code after the coroutine returns (e.g., in await_suspend()), so they need to implement a custom awaiter; in these cases the *task* is a CFP.

3.6 Coroutine Chaining Attacks

So far, we have described how to hijack the control flow to execute *a single* arbitrary function. However, just like in other code-reuse attacks such as ROP, attackers may potentially call arbitrarily many functions—even if the attacker can just write to memory *once*. The ability to call multiple attacker-controlled functions depends highly on the coroutine program at hand. During our study, however, we found that these programs use common patterns: either a scheduler is in charge of organizing which coroutines are executed, or the coroutines themselves await each other independently, or a combination of both. Therefore, we will describe how to achieve multiple arbitrary calls in these two common scenarios.

3.6.1 Multiple Arbitrary Calls With Scheduler

Schedulers maintain a queue of coroutines and iterate over it to check for coroutines to resume. Unfortunately, this queue can become a target for attackers aiming to hijack the control flow. Due to their looping nature, attacks on schedulers can result in multiple arbitrary calls. For this, the attacker could (1) use frame manipulation over existing coroutine frames, hijacking their *resume* and *destroy* pointers, and (2) overwrite/-expand the scheduler queue with coroutine *handles* that point to the injected frames. In consequence, the scheduler will issue *resume()* and *destroy()* calls using the hijacked pointers.

3.6.2 Multiple Arbitrary Calls Without Scheduler

A program without a scheduler is far more challenging to exploit if the goal is to issue *multiple* arbitrary calls. In these programs, we find that coroutines transfer the execution flow via

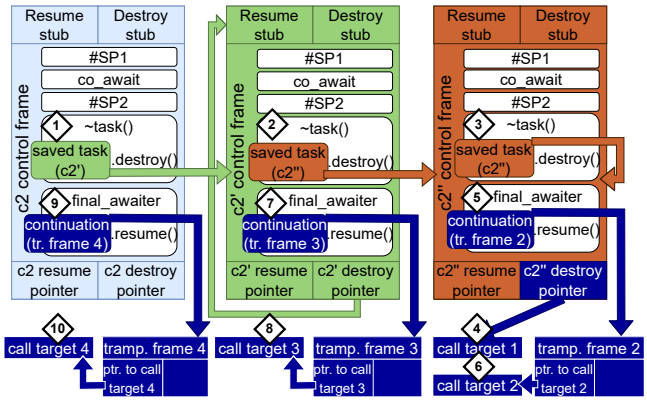

Figure 4: ICC attack on the program at Listing 2, showing how CFPs redirect the execution throughout the injected control and trampoline frames.

```
1  c2{
2      #SP1
3      co_await c3();
4      #SP2 <-- Coroutine index = SP2
5      ~task(){
6          destroy(); # call [rdi+0x8]
7      }
8      final_awaiter(){
9          continuation.resume(); # call [rdi]
10     } # ret
11 }
```

Listing 4: Pseudocode of each injected control frame in the program at Figure 4 while launching an ICC attack.

co_await (awaitable coroutines). As we described in § 3.5.3, this results in multiple CFPs in the form of continuation points and *task* objects saved in the frame.

CFOP can also leverage this architecture. The general idea is to create a chain of corrupted coroutines that will issue arbitrary calls during their execution, in a process we call *Infinite Coroutine Chaining* (ICC). ICC consists of creating an arbitrarily long chain of coroutines that are called sequentially, where each one is in charge of executing an attacker-defined arbitrary call. We find that it is always possible to build an ICC system if there exist *two or more useful CFPs* in a coroutine—one CFP calls the next element in the chain, and the other issues the arbitrary calls. Finding two CFPs is generally possible. While some CFPs are not exploitable (see § 3.5.3), we usually find CFPs in continuation objects and *task* destructors. These two CFPs are sufficient for ICC.

Figure 4 shows one possible setup that uses these two CFPs to achieve four arbitrary calls via ICC, targeting coroutine *c2* in our running example (Listing 2). The attacker first leverages an arbitrary write vulnerability to corrupt the heap, injecting multiple coroutine *frames*, twice as many as arbitrary calls desired to be executed (minus one), followed by a series of pointers, where each pointer points to one of the arbitrary calls targets where to jump. At least one of the injected frames will need to overwrite an existing one, in order to initially redirect the execution to the ICC chain. Half of the injected frames in ICC are *trampoline* frames, solely used as an indirection to jump to the desired call targets and point to the call target addresses. The other half are *control* frames, used to exploit the CFPs of the vulnerable coroutine. Inside each of them, the attacker puts (1) a continuation object that points to the address of one of the trampoline frames; (2) a *task* object that points to the next control frame in the chain; and (3) a suspension index such that the coroutine, when resumed, will start right at the *destroy()* call corresponding to the *task* destructor. In a nutshell, the ICC exploit chains the execution of each control frame by leveraging one of their CFPs, while the other CFP is used to resume a trampoline frame, jumping to the desired call targets.

Starting from the injected *c2*, Listing 4 shows the code run for every control frame injected by the attacker. The attack proceeds as follows (each step corresponds to one step in Figure 4): ① *c2* gets resumed at SP2 and, being at the end of the function, calls *destroy()* on its *task* object, which points to the *c2'* control frame. Note how the attacker has set the *destroy* pointer in *c2'* as the normal *resume* pointer of *c2'*. The result is that *c2'* is resumed as if a normal *resume()* call was issued from *c2*. ② Once *c2'* is resumed at SP2, it repeats the same process of resuming the next control frame *c2"*: *c2'* uses its *task* object, which points to *c2"*, to call *destroy()*, resulting in a *resume()* call, since the *destroy* pointer of *c2"* was set as its *resume* pointer. ③④ Once *c2"* is resumed, the chain starts with the next phase—executing the arbitrary attacker-defined calls. Since the attacker introduced six frames (three control and three trampoline frames), attackers achieve (n/2)+1=4 arbitrary calls. Only the first call target can be executed directly—the rest will use the trampoline frames. The first step is to use the *destroy* call on the *c2"* task object to issue the first call. For this, it sets the destroy pointer of *c2"* to the address of the first call target to make. When the *task* is destroyed, the call target is executed and then returns. ⑤⑥ After the call target function returns, the execution continues in *c2"* right after the previous *destroy* call. At this point, the coroutine evaluates its continuation point to know which coroutine to resume next. The *c2"* continuation point was set to point to the first trampoline frame, whose *resume* pointer points to the next call target to execute. As a result, *resume()* is called on the trampoline frame, executing the second call target, then returning. ⑦⑧⑨⑩ After the second call target returns, the coroutine reaches the end of its execution, and *c2"* returns like any other function. The execution returns to *c2'*, where its continuation point is evaluated. At this point, the same process as with *c2"* is repeated for the remaining call targets: we evaluate *resume()* on a trampoline frame whose *resume* pointer points to the next call target. Every *resume* call results in the execution of a call target.

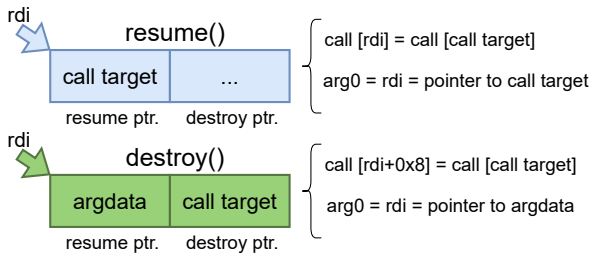

Figure 5: Hijacking of the call target and first argument via the *resume* and *destroy* pointers.

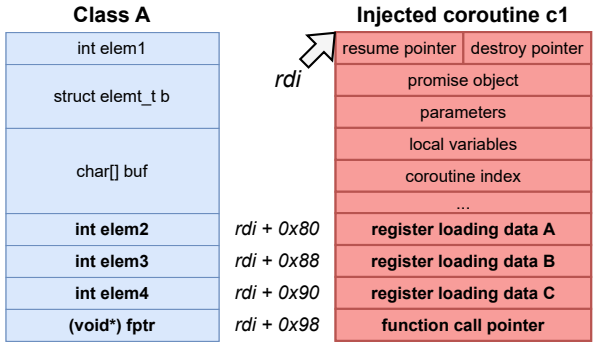

Figure 6: Use of a *golden gadget* at a member function to load arbitrary register values from an injected coroutine frame.

One may question the necessity of using trampoline frames at all—in the end, the attacker could move the call target address from the trampoline frame to the *resume* pointer in the same control frame. However, coroutines mark themselves as *done* after reaching their last SP, right before reaching the second CFP. Compilers implement this by zeroing the value of the *resume* pointer, so any value saved there is unusable at the time of executing the second CFP.

The previous ICC setup is independent of the implementation details of the coroutine execution transfer implementation that we explained in Appendix C. For the readers interested in this and other low-level differences while creating ICCs, we have detailed them in Appendix D.

3.7 Passing Arbitrary Arguments

So far, we ignored how attackers can pass arguments to target functions. To this end, attackers must control the values of the registers used in the respective calling convention. Without losing generality, we focus on Linux x64, where arguments are passed using `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, and the stack.

Controlling the first parameter is straightforward. Every *resume()* or *destroy()* call will always set `rdi` to the coroutine's frame pointer. Hence, if a CFP is based on hijacking *destroy()*, we always pass a first parameter `rdi=<pointer to 8 bytes>`, where the bytes correspond to the section in the frame where the *resume* pointer is stored, which the attacker may corrupt. For *resume()*, we always pass a first parameter `rdi=<pointer to 8 bytes>`, but the bytes are fixed to be the *resume* pointer. Both setups are represented in Figure 5.

Controlling the subsequent arguments is more challenging. In fact, all other registers' values are mostly unpredictable when any of the CFP is used. In rare situations, attackers may find and simply reuse other (stub) functions that prepare the exact arguments for them. Alternatively, attackers may leverage DOAs (§ 3.3.1) to modify any coroutine variables that the coroutine later uses when calling a desired target function. For example, suppose a coroutine reads some data from a file, and the arguments of the call (e.g., *read(buffer, bytes_to_read)*) are local variables or coroutine parameters. Still, attackers often do not find such attack requisites being met. In particular, the latter technique cannot call an *arbitrary* function, so the attacker may only control the arguments of *existing* calls within coroutines.

Seeing these limitations, we propose a generic technique to pass arguments in a reliable way. The basic idea is to leverage C++ member functions that use an attacker-controlled heap object to set all argument registers. We call such functions *golden* and *silver gadgets*, respectively. The only difference between these two gadget types is that golden gadgets also call the target function, while silver gadgets only prepare function arguments and then return.

Golden gadgets. An attacker can use the controlled `rdi` register to their advantage. In C++, `rdi` is used in functions belonging to a class (*member functions*) to pass an implicit first argument—the *this* object. This means that calling a member function from a CFP results in executing the function considering the attacker-controlled coroutine frame as the *this* object. Many member functions do, naturally, access member variables to perform operations with them. This results in *mov* operations that load data from an offset at `rdi` since accessing a member variable is equivalent to accessing data at an offset of the start of the object. However, instead of loading data from an actual C++ object into the target registers, this copies data from the attacker-controlled coroutine frame. Therefore, we define as a *golden gadget* any member function that (1) loads data from a member variable into the desired registers *and* does not corrupt these values before the final call, and (2) uses a member variable to perform an indirect call/jump.

If an attacker finds a *golden gadget*, and a CFP is available, an arbitrary call with a controlled argument is generally possible. The attack is as follows: (1) We inject a coroutine frame that leads to a CFP hijack. (2) We use the CFP to call a member function, which interprets `rdi = frame` as the *this* pointer. (3) The member function operates on member variables, loading them from an offset of the coroutine frame to the target registers (`rsi`, `rdx`, `rcx`, ...). These register values must not be clobbered until the next step. (4) The member function makes a call using a function pointer (or similar),

which is a member of the class, loading it from an offset of the coroutine frame, too.

Figure 6 shows an example class (left) that could be the basis for a *golden gadget*. Each object of the class has a member at the same offset, including a function pointer. A *golden gadget* arises when a member function moves these member variables (*elem2*, *elem3*, ...) to the respective argument registers and then issues an indirect call using *fptr*. Since all such members are loaded from an attacker-controlled fake object (the frame), the attacker fully controls all member variables and, hence, call target and arguments.

Silver gadgets. Although the existence of a *golden gadget* in a program is convenient for the attacker, not all programs will feature such functions. We thus relax the attack assumptions and define the concept of a *silver gadget*: a *golden gadget* without the requirement of making an additional call. Such silver gadgets are fairly common, as most member functions operate over member variables. We set every register to the desired values using the same technique as described with *golden gadgets* (we need a CFP and inject a coroutine frame with the data to load into the registers). Since silver gadgets do not issue a controlled call, the target function has to be called via a second CFP, using either two hijacked pointers (e.g., *resume()* and *destroy()*) or using one of the general chaining techniques described in § 3.6. If two CFPs exist and between both the argument-passing registers are not modified, *silver gadgets* allow to call an arbitrary function with arbitrary arguments. Although this may sound uncommon, this is usually the case for schedulers (that resume a coroutine and then proceed to destroy it) and awaitable coroutines (the continuation point and destroyer CFPs that we described for the ICC scenario can always be used to leverage a *silver gadget*).

Automating CFOP attacks. Searching for *golden* and *silver gadgets* can be challenging, since we must (1) find gadgets that fill the registers with attacker-controlled content; and (2) identify which functions do not overwrite the registers later with other data. The typically large number of member functions in programs complicate manual analysis. Instead, attackers may use automated analysis methods, such as taint analysis or symbolic execution [32, 46]. We partially automated the gadget search for a large real program, as we show in § 4.3.

Attackers can similarly automate the search for CFPs to construct automatic CFOP payloads. By leveraging static analysis tools, attackers can infer the structure of coroutine frames and locate the CFPs. Once memory randomization protections (e.g., ASLR) are bypassed, they can inject a payload to hijack the CFPs, enabling arbitrary function calls with controlled arguments.

4 Evaluation

This section evaluates how common coroutines are in real-world programs, the feasibility of our exploitation techniques when facing state-of-the-art CFI schemes, and PoC attacks spanning exemplary (see § 8.2) and real programs.

4.1 Prevalence of Coroutines

We will first analyze the prevalence of coroutines in real programs. For this, we gathered statistics about the most popular open-source projects in GitHub using C++ coroutines.

We found 131 repositories with more than 500 GitHub stars ("*popular*") using C++ coroutines. Moreover, 24 of these have more than 10k GitHub stars ("*very popular*"). These include well-known projects such as the Windows terminal (94k+ stars) [38], the SerenityOS operating system (30k+ stars) [49], and Facebook projects such as RocksDB (28k+ stars) [23] and HHVM (27k+ stars) [22]. In total, our search rendered 2.9k unique projects currently using coroutines in GitHub.

We find these numbers substantial, considering that coroutines were introduced only recently with C++20. Also, C++ coroutines are complex to use (due to their low-level nature) and using them requires adjusting to a paradigm new in C++. We expect that their adoption will increase over time.

We have identified that the community has used C++ coroutines for two main types of programs: First, *databases* are asynchronous by nature, as operations may involve long waiting times and require scheduling strategies. Examples include RocksDB (28k+ stars), FoundationDB (14k+ stars) [3], ArangoDB (13k+ stars) [4], and ScyllaDB (13k+ stars) [47]. Second, we found them in *frameworks and libraries*, which provide abstractions over C++ coroutines for easier programming. Examples include the coroutine libraries CppCoro (3.3k stars) [33] and ConcurrenCPP (2.2k stars) [19]; the large-scale coroutine-based frameworks Seastar (8.2k stars) [48] and Folly (27.8k stars) [21] that are used in ScyllaDB and Facebook programs respectively; and other general-purpose libraries that already incorporate coroutines, such as WinRT (1.6k stars) [36] and asio (4.8k stars) [13].

Finally, we analyzed *how* coroutines were used by manually inspecting the most popular 79 coroutine-based projects on GitHub (i.e., $\geq$ 1k stars). 41 of them (52%) use continuation points inside the coroutines, following a similar paradigm to the one we described in Listing 2. Another 22 projects (28%) use a scheduling strategy, where coroutines are enqueued and managed by a scheduler. Relevant examples include Folly and CppCoro, which feature continuation points, while WinRT and Seastar are found with schedulers.

Overall, we have found that coroutines are a growing presence in a varied population of projects and are already a very commonly used feature in some of them, such as databases. This underlines the relevance of our study and the need for defense strategies against coroutine-based exploitation attacks. Moreover, the continued repetition of coroutine architectural patterns—schedulers and continuation-based awaitable coroutines—shows the easy applicability of our techniques in the majority of coroutine programs.

Table 2: Considered CFI Schemes. A CFI scheme supports Linux ( ) or Windows ( ) systems; targets indirect calls (`I`), backward jumps (`B`), or virtual pointers (`V`); utilizes code instrumentation ( ) or hardware enforcement ( ); provides coarse-grained ( ) or fine-grained ( ) CFI; offers full ( ), partial ( ) or no protection ( ) against coroutines exploitation.

Name	OS	Features	Prot.
IBT (Intel CET)		`I`	◑
ShadowStack (Intel CET)		`B`	○
LLVM Clang CFI (Vcall)		`V`	○
LLVM Clang CFI (Icall)		`I`	○
KCFI		`I`	○
Control Flow Guard (CFG)		`I`	◑
SafeDispatch		`V`	○

4.2 CFI Guarantees for Coroutines

Next, we study the feasibility of CFOP when facing state-of-the-art CFI defenses. To this end, we evaluated widely-deployed CFI schemes and other academic-proposed schemes with respect to coroutine coverage. For this, we tried to compile simple programs using coroutines with each CFI scheme and analyzed if the scheme supported compiling coroutines and was aware of coroutine semantics.

We summarize the CFI schemes compatible with coroutines in Table 2. In short, our coroutine exploitation techniques can bypass both the forward-edge and backward-edge integrity protections present in programs compiled using *both* fine-grained and coarse-grained CFI schemes. The main reason is that all but two CFI schemes (CFG and CET's IBT) omit any protection for function pointers generated in C++ coroutines. That is, *any* indirect call using coroutine-internal function pointers can be hijacked, as we described throughout the paper, by hijacking CFP objects. We could not further assess additional 8 CFI schemes, which we did include in our evaluation, as they either did not support coroutines or our coroutine program crashed (MCFI/piCFI [39], ReCFI [8], PathArmor [51], CFIXX [9], PittyPat [20], VTrust [53], TyPro [5] and VfGuard [43]).

Having said this, irrespective of not protecting coroutine pointers, two of the CFI schemes still *generally* restrict the targets of any indirect call/jump. In particular, Intel CET's IBT allows to jump to arbitrary destinations as long as it is the start of a function; attackers need to leverage silver and golden gadgets, instead of any arbitrary piece of code (e.g., ROP-style gadgets). Still, it is possible to leverage the existing CFPs for all CFOP attacks. CFG sets the same restrictions, with the addition of preventing jumps to functions that are never the target of indirect jumps during normal program execution. Thus, CFG decreases the number of available silver and gold gadgets, but still allows a wide range of jump targets (e.g., coroutine stubs, library APIs, functions jumped indirectly).

4.3 Real-World PoC Attacks

On top of the exemplary PoCs that showcase CFOP in different scenarios (see § 8.2), we extended our tests to two real programs.

ScyllaDB PoC: We developed a PoC exploit for one of the most popular databases using coroutines, ScyllaDB (13k+ stars). This is a NoSQL database that features an internal coroutine-based scheduler known as *Seastar*. We picked Scylla-2025.1.0-rc0, which is (at the time of writing) the latest version of ScyllaDB. We used a Linux x86_64 executable, compiled with Clang-18.1.8 and Intel CET enabled.

ScyllaDB users can interact with the database using CQL [10]. Following our threat model, we introduce (1) an arbitrary write vulnerability in the code part for receiving CQL input, writing the payload to an attacker-chosen memory address, and (2) a memory leak to bypass ASLR as a means to obtain the memory location of one of the scheduler's queues.

We aim for an exploit that executes an attacker-controlled system call with three attacker-controlled arguments. Without losing generality, we aim to execute *execve("/bin/sh", "-c", "/usr/bin/whoami")*. We exploit one of the CFPs in the coroutine scheduler, specifically when it calls *resume* on a coroutine. In this case, the coroutine frame in the scheduler queue is hijacked so that the *resume* pointer points to a golden gadget. This golden gadget sets the register values needed, then jumps to the *execve()* call.

In a nutshell, the exploit uses frame manipulation to hijack the coroutine frames in the scheduler (circular) queue. This queue is periodically used to dispatch groups of operations, including coroutines. We also hijack the pointers that mark the beginning and end of the circular queue to ensure that the coroutine gets executed exactly when the register `rsi` holds an appropriate value—`rdi` is already fixed to the coroutine frame. Both these registers are used from the golden gadget to load attacker-supplied values, then issue the *execve* call. We have described the full exploit details in our project repository.

To automate the tedious process of finding the necessary gadgets, we developed a script leveraging *radare2*'s symbolic execution framework. This tool extracts and evaluates every function in a given program to determine if they satisfy two key conditions: (1) The function loads a memory value from an address using an offset from the `rdi` register (e.g., `mov rsi, [rdi+0x90]`) and stores it in an argument-passing register (`rsi`, `rdx`, `rcx`, etc.). (2) Once the value is loaded into the argument-passing register, it remains unaltered until a *ret* or *jmp* instruction is encountered. Using this tool, we identified over 100 potential *silver gadgets* in ScyllaDB. After manual inspection, we also uncovered several *golden gadgets*.

SerenityOS PoC: We also developed a PoC exploit for the second most popular project currently using coroutines,

SerenityOS (30k+ stars), a Unix-like operating system. For this PoC, we took the latest version of the project in GitHub and reintroduced the old vulnerable code responsible for CVE-2021-4327, an integer overflow vulnerability—that can be exploited for an arbitrary memory write. We showcase an exploit in the vulnerable Linux x86_64 (*lagom*) version of the SerenityOS browser, Ladybird, compiled with GCC-13.3.0 and Intel CET enabled.

The CVE-2021-4327 vulnerability resides in the internal *LibJS* library, which Ladybird uses for website JavaScript parsing. Meanwhile, the browser and many other OS components rely on *LibCore*, a core SerenityOS library providing foundational functionality, in which developers are actively integrating coroutines. Although Ladybird itself does not currently utilize these coroutines, the *LibCore* library is linked to every binary. Thus, we exploit the CVE-2021-4327 vulnerability to inject malicious coroutine frames into memory, and to redirect the execution flow to the *LibCore* coroutine code responsible for parsing them. The result is the execution of an ICC chain, leveraging coroutines to execute an arbitrary number of functions indefinitely.

In a nutshell, our exploit consists of a malicious HTML file—with embedded JavaScript code—that triggers the vulnerability in the browser when the website is visited. We exploit the *WebContent* process of the browser, injecting multiple coroutine frames into the process memory. In addition, a vtable pointer is overwritten to redirect the execution flow to the *LibCore* coroutine function *adopt_coroutine()*, which will parse the injected frames. Needing to overwrite a vtable pointer will be obsolete as soon as the *WebContent* process actively uses coroutines. The exploit then leverages the two CFPs in the *adopt_coroutine()* function (a destroyer, and an *await_suspend()* call) to launch an ICC chain. In this chain, an infinite number of functions can be executed; without losing generality, we execute *system("whoami")* three times.

5 Defenses

CFI schemes that use code instrumentation (see Table 2) can be extended to account for coroutine-related indirect jumps by restricting *resume* and *destroy* calls to their corresponding coroutine code. Having said this, instead of relying on CFI, we suggest to address the problem at its root. Drawing inspiration from prior binary protections like Relocation Read-Only (RELRO), which mitigates GOT overwrite attacks by marking the Global Offset Table read-only, we advocate for a similar approach to protect coroutine frames.

In this section, we propose conceptual changes to the coroutine implementation to mitigate control-flow hijacking attacks (§ 5.1) and DOAs (§ 5.2). Our security requirements (SR) and functionality requirements (FR) are as follows:

SR1: There must not exist any pointers in writable memory that are used for issuing indirect calls or jumps.

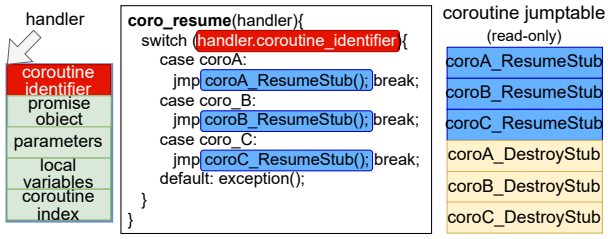

Figure 7: Proposed CFOP defense, moving vulnerable pointers to read-only memory, referenced by a *coroutine identifier*.

SR2: A memory corruption attack must be limited to affecting a single coroutine frame per attack.

SR3: The variables stored in the coroutine frame must have, at a minimum, the same level of protection as the data saved in the stack during the execution of normal functions.

FR1: A coroutine may be resumed or destroyed using just its handle (e.g., a scheduler must be able to resume a coroutine handle from a queue without knowing the coroutine type).

FR2: All the information about the coroutine must be contained in the coroutine frame; the coroutine must be stackless.

5.1 Protecting Coroutine Pointers

Just like any other code pointer, the *resume* and *destroy* pointers should not be stored in writable memory, as this is the basis of any control-flow hijacking attack. Therefore, in our proposed coroutine implementation concept, and fulfilling *SR1*, these pointers are therefore stored in read-only memory.

Figure 7 shows how the frame and other components look like in this new concept. Firstly, the *resume* and *destroy* pointers of every coroutine are no longer stored in the corresponding coroutine frame. Instead, they are moved into a list in read-only memory. Since the *resume* and *destroy* pointers are the same for every coroutine type (all instantiations of a coroutine type share the same *frame template*), the list includes a single *resume* and a single *destroy* pointer per coroutine type.

Currently, the *resume* and *destroy* pointers are the only elements of the frame that indicate which type of coroutine it belongs to (since these pointers point to the *resume stub* and *destroy stub*, respectively). In our new design, these pointers are no longer in the frame, which in principle would not satisfy *FR1*. Also, we would not satisfy *FR2* because, from a coroutine handle, there would not exist a link to the corresponding *resume* or *destroy stubs*. For this reason, we introduce a new element in the frame: the *coroutine identifier*.

The *coroutine identifier* is an integer number unique for each coroutine type (i.e., the compiler assigns the same identifier to each instantiation of coroutine coroA). It acts as the link between the coroutine frame and the read-only list of *resume* and *destroy* pointers. When a coroutine is resumed or destroyed, the identifier is evaluated in a *switch* clause. Each switch case corresponds to one of the possible *identifiers*, i.e.,

coroutine types used in the program. The program will use the *identifier* as an offset to access the read-only *jumptable* of *resume* and *destroy* pointers.

This new design prevents control-flow hijacking without the need for additional CFI schemes. Attackers are now limited to overwriting the *identifier* instead of the *resume* and *destroy* pointers. The best remaining attack is to alter which coroutine gets resumed or destroyed, but it is impossible to redirect the execution to an arbitrary code location.

5.2 Protecting Data in the Coroutine Frame

We have shown in § 3.4 how the data in the coroutine frame can be one of the weakest links in the chain. Although its data is protected against buffer overflows (via variable reordering), frames are generally vulnerable to overflows from adjacent frames. While a targeted write primitive can always modify writable memory, we strive to offer a similar degree of security as if the coroutine frame was in the stack.

We can draw inspiration from existing stack-based protections: (1) the compiler reorders the variables in a function so that any potentially overflowing buffer is positioned in higher memory addresses, where it cannot corrupt other variables; and (2) canaries protect stack frame boundaries. Both defenses fulfill *SR2* and *SR3* and also apply to the coroutine frames in the heap: (1) Programs can use heap allocators that randomize the chunks in the heap. If every coroutine frame is allocated next to each other, as in *ptmalloc*, this eases the required memory leaking work by an attacker. (2) Programs can use heap canaries that protect heap chunks so that a single continuous memory write cannot corrupt more than one coroutine frame—necessary in most CFOP attacks.

6 Discussion

This section reviews multiple aspects to bear in mind while developing CFOP attacks, completing our guide on exploiting the current implementation weaknesses of C++ coroutines.

6.1 ICC with Two Coroutines

In § 3.6, we describe an ICC scenario where we exploit a program using three nested coroutines. Arguably, not every program—even if using continuation points—will feature three nested coroutines; more often, we would find two coroutines in a parent-child fashion. This does not necessarily mean we can no longer attain the minimum of two CFPs needed for CFP, contrary to what would be understood from Figure 3. Firstly, the coroutine program could feature other useful CFPs other than the continuation point and the saved *task*. Secondly, both of these CFPs are usually still available in the parent coroutine in a two-coroutine scenario. Since the *awaiter* implementation is the same for the parent and child coroutine, this means that if the child evaluates a continuation point, the parent also features the same code for this. The continuation would be set to NULL—as no coroutine called the parent—, but an attacker can leverage this as a CFP by setting any other value in the frame. Therefore, every parent coroutine commonly features at least two CFPs, and ICC is possible.

6.2 Compilers and HALO

The coroutine implementation we have described in this paper is common to the three major compilers: GCC, Clang and MSVC implement coroutines following the same principles. The main difference at the assembly level is the degree of optimization applied to coroutines code. One such optimization alters how coroutines work: Heap Allocation eLision Optimization (HALO) [28]. HALO intends to eliminate the individual dynamic allocations that take place every time a coroutine is created, which can be resource-demanding.

When a coroutine is optimized under HALO, its frame is moved from the heap to the stack but maintains the same elements as usual. However, we find that, at the assembly level, the program no longer uses the *resume* or *destroy* pointers. Instead, the call addresses are hardcoded into the *call* instruction, ignoring the values in the frame and performing a direct jump to the resume and destroy stubs, respectively. This effectively renders control flow hijacking attacks impossible. However, programs are still vulnerable to DOAs.

In practice, the HALO optimization is rarely present in coroutine programs; the difficulty of its implementation and the strict criteria that programs must meet to be eligible for the optimization have limited its widespread availability. Currently, only the latest versions of Clang fully support HALO for coroutine programs. In a nutshell, for HALO to be applied, the compiler must have a high degree of information on the coroutine, ensuring that its lifetime is limited to the scope of a function, i.e., the coroutine is created, resumed, and destroyed within a function scope. In addition, many of the coroutine-related functions need to be inlinable (see Appendix F).

7 Conclusion

We found that all three major C++ compilers implement coroutines that are vulnerable to CFOP, a novel code reuse attack. CFOP encompasses techniques for altering the data used during program execution (DOAs) and executing arbitrary code (CFP exploitation, ICC, argument passing techniques). CFOP evades both coarse-grained and fine-grained CFI schemes. We studied the general applicability of our techniques and the population of programs currently vulnerable to CFOP. We found that coroutines gain popularity, particularly in databases, and prepared PoC exploits for two popular programs. Finally, we proposed an implementation alternative that would mitigate and prevent CFOP attacks.

8 Ethics and Open Science

8.1 Ethics considerations

All experiments carried out in this paper were done in an isolated manner against locally-hosted services. We did not interact with real-world systems to avoid any damage.

Since this paper shows that the current implementation of C++ coroutines in the three major compilers—Clang/LLVM, GCC and MSVC—is vulnerable, we initiated a disclosure process with the corresponding parties to raise awareness and to provide developers the opportunity to implement our proposed mitigations. We shared a copy of this paper with the affected compiler developers in November 2024. The general consensus was that this should not be classified as a security vulnerability but rather as a hardening opportunity against a novel exploitation technique. The Clang/LLVM developers acknowledged the issue and are actively discussing the best way to address it. The GCC developers have expressed their desire to open an upstream bug report when the paper is published. Meanwhile, the Microsoft Security Response Center referred the matter to the MSVC developers, from which we have not yet received further feedback.

We declare no conflicts of interest or other ethical issues. We have not received funding from any of the affected parties.

8.2 Open Science

As part of our work, we have prepared a Proof of Concept (PoC) in two real programs using coroutines (ScyllaDB and SerenityOS, see § 4.3). On top of that, aiming for the reproducibility of the different techniques in CFOP, we have prepared a compilation of PoC exploits, each consisting of a vulnerable coroutine program and an exploit script—compiled with CET or CFG. The PoCs exploits include multiple examples of ICC exploitation, showcasing how to chain coroutines infinitely under any control flow transfer strategy (see Appendix C); DOAs, showing how they can be used to access an arbitrary file and how to call a function with arbitrary arguments using the frame; and a combination of ICC exploits and argument-passing techniques using a silver gadget, showcasing how to spawn a shell and execute arbitrary code.

All the code, scripts and materials we mention in this paper are available in our Zenodo record (*10.5281/zenodo.14738036*) and our GitHub repository (*https://github.com/coroutine-cfop/cfop*).

References

[1] Merge coroutines ts into c++20 working draft. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0912r2.html. (04-09-2024).

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[3] Apple. Foundationdb. https://github.com/apple/foundationdb. (04-09-2024).

[4] ArangoDB. Arangodb. https://github.com/arangodb/arangodb. (04-09-2024).

[5] Markus Bauer, Ilya Grishchenko, and Christian Rossow. Typro: Forward cfi for c-style indirect function calls using type propagation. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, page 346–360, New York, NY, USA, 2022. Association for Computing Machinery.

[6] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security, ASIACCS*, 2011.

[7] Erik Bosman and Herbert Bos. Framing signals - a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258, 2014.

[8] Oliver Braunsdorf, Stefan Sessinghaus, and Julian Horsch. Compiler-based attack origin tracking with dynamic taint analysis. In *Information Security and Cryptology – ICISC 2021: 24th International Conference, Seoul, South Korea, December 1–3, 2021, Revised Selected Papers*, page 175–191, Berlin, Heidelberg, 2021. Springer-Verlag.

[9] Nathan Burow, Derrick McKee, Scott Carr, and Mathias Payer. Cfixx: Object type integrity for c++. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 01 2018.

[10] Apache Cassandra. The cassandra query language (cql). https://cassandra.apache.org/doc/stable/cassandra/cql/. (04-09-2024).

[11] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. Exploitation techniques for data-oriented attacks with existing and potential defense approaches. *ACM Trans. Priv. Secur.*, 24(4), sep 2021.

[12] Tzi-Cker Chiueh and Fu-Hau Hsu. Rad: a compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 409–417, 2001.

[13] chriskohlhoff. asio. https://github.com/chriskohlhoff/asio. (04-09-2024).

[14] Clang/LLVM. Control flow integrity design documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html. (04-09-2024).

[15] cOntext. Bypassing non-executable stack during exploitation using return-to-libc. http://staff.ustc.edu.cn/~bjhua/courses/security/2014/readings/return-to-libc.pdf. (04-09-2024).

[16] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of Buffer-Overflow attacks. In *7th USENIX Security Symposium (USENIX Security 98)*, San Antonio, TX, January 1998. USENIX Association.

[17] CppReference. C++20 core language features. https://en.cppreference.com/w/cpp/compiler_support#C.2B.2B20_features. (04-09-2024).

[18] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of Coarse-Grained Control-Flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[19] David-Haim. concurrencpp, the c++ concurrency library. https://github.com/David-Haim/concurrencpp. (04-09-2024).

[20] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. Efficient protection of Path-Sensitive control security. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 131–148, Vancouver, BC, August 2017. USENIX Association.

[21] Facebook. Folly: Facebook open-source library. https://github.com/facebook/folly. (04-09-2024).

[22] Facebook. Hhvm. https://github.com/facebook/hhvm. (04-09-2024).

[23] Facebook. Rocksdb. https://github.com/facebook/rocksdb. (04-09-2024).

[24] Linux Foundation. *System V Application Binary Interface AMD64 Architecture Processor Supplement*, 2012.

[25] Wolfram Gloger. Wolfram gloger's malloc homepage. http://www.malloc.de/en/index.html. (04-09-2024).

[26] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, 2016.

[27] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual.*, 2020.

[28] ISO/IEC. Halo: coroutine heap allocation elision optimization: the joint response. https://open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0981r0.html. (04-09-2024).

[29] ISO/IEC. *N4775: Working Draft, C++ Extensions for Coroutines*.

[30] Seunghoon Jeong, Jaejoon Hwang, Hyukjin Kwon, and Dongkyoo Shin. A cfi countermeasure against got overwrite attacks. *IEEE Access*, 8:36267–36280, 2020.

[31] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, page 339–348, USA, 2006. IEEE Computer Society.

[32] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[33] lewissbaker. Cppcoro - a coroutine library for c++. https://github.com/lewissbaker/cppcoro. (04-09-2024).

[34] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 941–951, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Microsoft. Control flow guard for platform security. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard. (04-09-2024).

[36] Microsoft. The c++/winrt language projection. https://github.com/microsoft/cppwinrt. (04-09-2024).

[37] Microsoft. Powertoys. https://github.com/microsoft/PowerToys. (04-09-2024).

[38] Microsoft. terminal. https://github.com/microsoft/terminal. (04-09-2024).

[39] Ben Niu and Gang Tan. Modular control-flow integrity. *SIGPLAN Not.*, 49(6):577–587, jun 2014.

[40] PaX. Address space randomization. https://pax.grsecurity.net/docs/aslr.txt, 2003. (04-09-2024).

[41] Jannik Pewny, Philipp Koppe, and Thorsten Holz. Steroids for doped applications: A compiler for automated data-oriented programming. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 111–126, 2019.

[42] Phantasmal Phantasmagoria. The malloc maleficarum. https://seclists.org/bugtraq/2005/Oct/118. (04-09-2024).

[43] Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 01 2015.

[44] Ahmad-Reza Sadeghi, Lucas Davi, and Per Larsen. Securing legacy software against real-world code-reuse exploits: Utopia, alchemy, or possible future? In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 55–61, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the IEEE Symposium on Security and Privacy, SP*, 2015.

[46] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[47] ScyllaDB. Scylladb. https://github.com/scylladb/scylladb. (04-09-2024).

[48] ScyllaDB. Seastar. https://github.com/scylladb/seastar. (04-09-2024).

[49] SerenityOS. Serenity. https://github.com/SerenityOS/serenity. (04-09-2024).

[50] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM conference on Computer and Communications Security, CCS*, 2007.

[51] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 927–940, New York, NY, USA, 2015. Association for Computing Machinery.

```
1  //The coroutine object
2  class task_aw {
3  public:
4      ///Omited other boilerplate code///
5      //Promise object
6      class promise_type {
7      public:
8          //Lazy-start coroutine
9          suspend_always initial_suspend() noexcept
               {return {};}
10         //Custom awaiter for final_suspend
11         struct final_awaiter {
12             bool await_ready() noexcept {return
                   false;}
13             void await_suspend(coroutomine_handle<
                   promise_type> h) noexcept {
14                 if(h.promise().cont){
15                     h.promise().cont.resume();
16                 }
17             }
18             void await_resume() noexcept {}
19         };
20         final_awaiter final_suspend() noexcept
21             {return {};}
22         coroutine_handle<> cont;
23     };
24
25     //Awaiter, evaluated by co_await
26     class awaiter {
27     public:
28         bool await_ready() noexcept {return false
               ;}
29         void await_suspend(coroutine_handle<> cont
               ) noexcept {
30             coro_.promise.cont = cont;
31             coro_.resume();
32         }
33         void await_resume() noexcept {}
34     };
35     ~task() {if (coro_)coro_.destroy();}
36     coroutine_handle<promise_type> coro_;
37 };
```

Listing 5: Definition of the coroutine *task* object corresponding to the program shown in Listing 2.

[52] Perry Wagle, Crispin Cowan, and Immunix. Stackguard: Simple stack smash protection for gcc. 2004.

[53] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chenyu Song, Mathias Payer, and Dawn Song. Vtrust: Regaining trust on virtual calls. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

A C++ Coroutines *Task* Definition

The code shown in Listing 5 corresponds to a simplified definition of the *task_aw* object in Listing 2. As we have explained throughout § 2.1.2, the *task* object always includes the definition of the coroutine handle (line 36), the coroutine promise (lines 6-23), the awaiter used when invoking

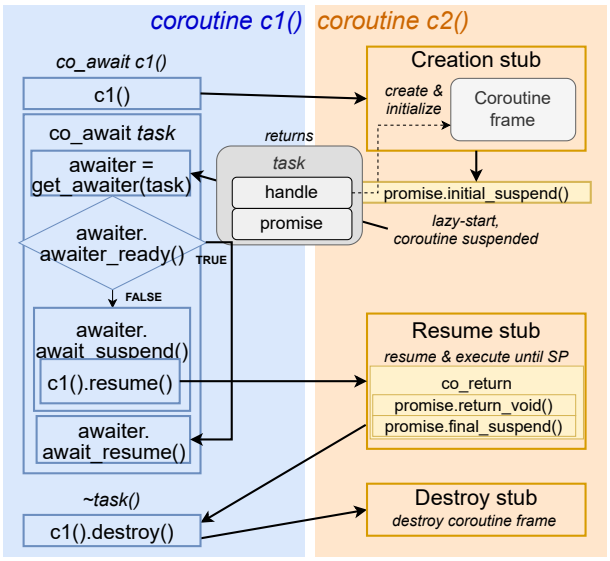

Figure 8: Coroutine lifetime (when a coroutine is co_await-ed from another coroutine, as in the program at Listing 2).

co_await on the coroutine (lines 26-34) and—in this case—the custom awaiter *final_awaiter*, evaluated when the function reaches its final suspension point in *final_suspend()*.

In this example, the implementation of the *await_suspend()* functions in the awaiters allows the coroutines in Listing 2 to suspend and resume each other. This is the simplest case of nested coroutine program; more complicated behaviors (e.g., asynchronony with multi-threading) would similarly be implemented in the awaiters and other parts of the *task*.

B Assembly Code in Coroutine Stubs

We show in Listing 6 the assembly code generated by a compiler during the transformations that take place once it detects a coroutine. We can find the three different stubs—*creation*, *resume*, and *destroy stubs*— and how the program always uses indirect jumps to find the address of the last two.

C Transferring Execution Between Coroutines

In § 3.6.2, we describe how programs may use coroutines to transfer the execution to one another via the *awaiters* and the operator *co_await*. The C++ coroutines API is very flexible, enabling different implementation mechanisms inside the *awaiter* (e.g., in the *await_suspend()* function). Depending on the implementation, we can find that the compiler generates fundamentally different code. These details should be considered when preparing CFOP attacks, although every technique described in this paper is still applicable in every case.

***Asymmetric transfer* with void-returning *await_suspend()*.** This strategy is the one we detailed

```
1  c1.CreationStub:
2      call new() ;heap-based coroutine frame
3      mov rcx, <address of c1.ResumeStub>
4      mov [rax], rcx ;write to frame
5      mov rcx, <address of c1.DestroyStub>
6      mov [rax + 0x8], rcx ;write to frame
7      mov [rax + 0x30], 0x0 ;initial SP value
8          ...
9  c1.ResumeStub: ;rdi = ptr to coroutine frame
10     mov rax, [rdi+0x30] ;gets the SP
11     cmp rax, 0x0
12     je c1.ResumeStub.FirstSuspensionPoint
13     cmp rax, 0x1
14     je c2.ResumeStub.SecondSuspensionPoint
15 c1.ResumeStub.FirstSuspensionPoint:
16     ;executes code until co_awaiting c2
17     mov [rdi+0x30], 1 ;sets new SP
18         ...
19 c1.ResumeStub.SecondSuspensionPoint:
20     ;executes code right after co_awaiting c2
21     mov [rdi+0x30], 2 ;sets new SP
22         ...
23 c1.DestroyStub: ;rdi = ptr to coroutine frame
24     call delete() ;frees frame in the heap
25         ...
26 main:
27     call c1.CreationStub ;create coroutine c1
28     ;rdi holds ptr to created coroutine frame
29     call [rdi] ;resume() = call ResumeStub
30     call [rdi+8] ;destroy() = call DestroyStub
```

Listing 6: Assembly code used to create, resume and destroy coroutine c1() from function main() in Listing 2.

in § 3.6.2. Taking Listing 2 as a reference, it works purely by setting *continuation points* inside the awaited coroutine (*c2*).

Although programs built using this asymmetric transfer version are perfectly functional and can be found in real-world programs, they suffer from a fundamental flaw: Every time we issue a *resume()* call to a continuation point, we are executing an assembly *call* instruction equivalent to the one shown in Listing 4. By design, when we execute a *call* instruction, the return address is stored in the stack. Since we keep calling *resume()* every time we execute a new coroutine, a program where a pair of coroutines is suspended and resumed in a loop could eventually grow the stack sufficiently to cause a stack overflow. For this reason, and although many programs will never face this problem, it is recommended to use any of the other two control flow transfer strategies.

***Asymmetric transfer* with bool-returning *await_suspend()*.** This strategy intends to fix the stack overflow problem of normal asymmetric transfer by altering the *awaiter*: now the function *await_suspend()* can also return a boolean value.

Assuming the same scenario with coroutine *c1* co_awaiting coroutine *c2* as previously, we find that: (1) If *await_suspend()* from *c2*'s *awaiter* returns true, then the coroutine is suspended and the execution returns to *c1*. (2) If *await_suspend()* from *c2*'s *awaiter* returns false, then *c2* is resumed.

```
1  coroutine_handle<> final_awaiter::await_suspend(
       coroutine_handle<promise_type> h) noexcept {
2      if(h.promise().cont){
3          return h.promise().cont;     //jmp <cont>
4      }
5      return noop_coroutine(); //return to previous
           coroutine
6  }
7  coroutine_handle<> awaiter::await_suspend(
       coroutine_handle<> cont) noexcept {
8      coro_.promise().cont = cont;
9      return coro_; //jmp <coro_>
10 }
```

Listing 7: Awaiters using symmetric transfer, equivalent to the asymmetric awaiters shown in Listing 2.

Unlike with the void-returning asymmetric transfer version, where the continuation points may resume any arbitrary coroutine, using this strategy, we are only allowed to either return to the previous coroutine or continue with the same one. The main benefit is that this transfer does not involve any *call* instruction at all, but rather the coroutine issues a *ret* instruction to come back to the previous coroutine, avoiding the stack grow problem.

It is important to notice that, in real-life programs, programs using the boolean version of *await_suspend()* do not simply use the boolean return value to control every execution flow transfer between coroutines. Most commonly, programs combine the boolean return value with *continuation points* and calls to *resume()* and *destroy()* inside *await_suspend()* too. In this way, they avoid growing the stack during the most common execution transfers while conserving the flexibility of being able to resume any other coroutine (e.g., *lazy-start* coroutines still include a call to *resume()* inside the *await_suspend()* method even when using its boolean version).

Symmetric Transfer Symmetric transfer is the most recommended strategy for managing the control flow transfer between coroutines. Instead of returning void or a boolean from *await_suspend()*, we can now return a *coroutine handle* from it. The compiler is then prepared to take this handle and call *resume()* on it, but as an optimized tail-call, instead of using the instruction *call*, it uses a *jmp* instruction. This prevents the stack from growing altogether while it guarantees maximum flexibility.

Listing 7 shows how awaiters return handles instead of calling *resume()* on them. Note the existence of the *noop_coroutine handle*; this is a dummy handle offered by the coroutines API whose *resume pointer* points to a *ret* instruction. Therefore, it is used when we do not want to resume another coroutine, but rather suspend the current one and resume the previous. Notably, we found that the *noop handler* is stored in writable memory and contains a code address used to make a jump operation. Thus, attackers may corrupt this pointer and easily redirect the execution flow to an arbitrary address.

D ICC Details depending on Coroutine Control Flow Strategy

The ICC setup we describe in § 3.6 corresponds to a program using asymmetric transfer (see Appendix C) to control coroutines execution flow. However, the same setup also works for symmetric transfer, although the internal behavior is different. As we explained, to use symmetric transfer means that the compiler uses *jmp* instead of *call* instructions when transferring the execution flow inside the *awaiter*. Since a *jmp* never returns, the compiler positions error code after the *jmp* instruction, assuming that the code will never get executed.

The previous has implications for ICC since we can no longer rely on the coroutine reaching the end of its body and issuing a *ret* instruction. However, our technique still works: since the call target the attacker wants to execute is a function itself, it also includes a *ret* instruction at the end. Therefore, since we never issued a *call* to get to the function in the first place, the *ret* instruction takes the execution back to the previous coroutine in the frame, as that was the last instruction pointer saved in the stack.

Finally, a relevant aspect to consider while creating an ICC chain is that the *resume* pointers of a coroutine may not be usable depending on the moment we are trying to access the *frame*. When a coroutine reaches its final SP, executes all code belonging to it, and then finishes executing all code related to the awaiter from *final_suspend*, then the coroutine marks itself as "*done*". This is useful for developers to know from a *handler* when the coroutine has finished its execution. At a low level, we find that a coroutine marks itself as *done* by zeroing out the *resume* pointer. This is relevant when developing an ICC chain, as we must be particularly cautious when dealing with CFPs using *resume*, ensuring that the injected frames are not in a finished state.

E Relaxation of the Threat Model

Throughout the paper, we have detailed multiple possible exploitation techniques under the threat model in § 3.1. This model was not chosen arbitrarily but rather with the purpose of presenting a realistic scenario with similar restrictions as faced by any attacker looking to use code-reuse techniques in a machine protected by state-of-the-art defenses. However, it can be interesting to explore how the exploitation techniques described here can be performed while lifting some restrictions in the model.

Specifically, we will explore how the attacker may reproduce the exploitation techniques described while not having access to an arbitrary memory write primitive. Instead, we assume that the attacker may leverage a single buffer overflow that results in overwriting the directly contiguous memory with attacker-controlled data. We will distinguish between stack-based and heap-based overflows.

E.1 Stack-based Buffer Overflow

We first consider a regular function that (1) is vulnerable to a stack-based buffer overflow and (2) will use a coroutine or has callers that will use coroutines after the overflow. Note that we will discuss heap-based overflows, i.e., stack-based overflows *within* a coroutine, in § E.1.1. Overflowing a buffer on the stack is common for arrays and other types of variables used to store user-supplied data. To mitigate this risk, compilers typically position non-primitive ("risky") variables at the higher addresses such that an overflow does not overwrite other (e.g., primitive) local variables. This provides some implicit protection of the coroutine *handle* if we overwrite a buffer in a function using a coroutine.

However, it is still possible to modify a handle if a function creates a coroutine and then calls other—nested—functions: (1) Some function A holding a coroutine handle, which is later used, calls function B. (2) The attacker overwrites some buffer used in function B. The attacker injects a fake coroutine frame into the stack and overwrites the coroutine *handle* in function A to point to this newly injected frame. It uses memory leaks to avoid changing the value of the return address in the stack (so as not to trigger backward-edge CFI). (3) Function B returns. Function A then uses the coroutine *handle*, overwritten to the injected coroutine *frame* in the stack, leading to arbitrary code.

E.1.1 Heap-based Buffer Overflow

The attacker may trigger a heap-based buffer overflow either by overflowing a heap-based object or overflowing a typically stack-based buffer used from a coroutine.

We first look at overflows within a coroutine. Similar to stack-based overflows, the compiler positions potentially vulnerable buffers at the end of the coroutine *frame*. Hence, regarding the frame layout (Figure 1), an attacker may (1) overwrite the *coroutine index*; (2) overwrite other coroutine *frames* positioned in higher memory addresses; (3) corrupt other heap-based objects, such as the chunks where the value of objects used from a coroutine are saved.

If the attacker overflows some other coroutine-unrelated heap-based object, it may similarly overwrite coroutine frames positioned in higher memory addresses and corrupt other heap-based objects. In practice, this translates to the same capabilities as in the case of an arbitrary memory write.

E.2 Escalating a Limited Vulnerability

A limited arbitrary write vulnerability may not be sufficient for an attacker to carry out an exploit (e.g., because this is a heap-based buffer overflow that does not overwrite the *resume* or *destroy* pointers). In these cases, it can be interesting to explore how the attacker may use a vulnerability to pivot to other attack primitives. We find that the variables saved as pointers inside the coroutine frame are helpful for this.

Firstly, it is possible to modify a pointer so that when a value is assigned to the variable, it leads to an arbitrary write: (1) The attacker modifies the pointer in a coroutine frame to an arbitrary memory address. (2) The coroutine is resumed. During its execution, it reads an arbitrary value (potentially attacker-controlled) into the object whose pointer was modified. (3) The value is written to the attacker-set position.

Secondly, an attacker can tinker with heap structures using the coroutine frames. In many heap allocator implementations (such as the widespread *ptmalloc*), the heap presents a series of structures known as the *bins*, where heap chunks are saved after their deallocation for optimization purposes. These chunks are later retrieved in successive allocations. Historically, a wide range of heap-based exploitation techniques have existed that target this and other features of the heap allocator. With coroutines, we can reproduce some of these heap-based attacks, such as the House of Spirit [42]: (1) The attacker uses a limited buffer overflow to forge a valid coroutine frame, including the corresponding *ptmalloc* heap headers. (2) The attacker also overwrites a coroutine frame with a variable saved as a pointer. The pointer is set to point to the previously forged coroutine frame, and the resume index is set to the last one possible of the execution. (3) The coroutine gets resumed and, at the end of its execution, calls free() on every object variable, which corresponds to the overwritten pointer. This deallocates the memory corresponding to the forged heap frame. (4) When the forged heap frame is deallocated, it goes to the corresponding *bin* from the heap allocator, such as the *tcache*. (5) Any subsequent call to *malloc* with the same size as the forged heap frame will return the attacker-controlled chunk from the *tcache*.

F Application of HALO

The HALO optimization is only applied when a coroutine's lifetime (including creation, resuming, and destruction) is limited to one function scope. For the compiler to know this, it needs to inline most of the function stubs belonging to the coroutine in the function. This includes the *resume()* and *destroy()* functions, the constructor and destructor of the coroutine *task* object, and other boilerplate functions (e.g., the internal awaiter functions, such as *await_suspend()*). Any of these functions may result in being non-inlinable for various reasons: (1) the use of indirect calls inside the coroutine (e.g., a call to a vptr), since the compiler does not know at compile time the type of function being called and thus cannot calculate the stack space to reserve for the coroutine frame; (2) function definitions being at a different compilation unit (e.g., the coroutine declaration in a header file imported and used by multiple implementations), although this can be partially solved by the use of Link Time Optimization (LTO); and (3) the program interacting with coroutine objects outside the scope of the coroutine (e.g., a program accessing a value returned by a coroutine after it has finished).