Let Me Unwind That For You: Exceptions to Backward-Edge Protection

Victor Duta*§, Fabian Freyer†§, Fabio Pagani‡, Marius Muench* and Cristiano Giuffrida*
*Vrije Universiteit Amsterdam, {v.m.duta,m.muench}@vu.nl, giuffrida@cs.vu.nl
† mail@fabianfreyer.de
‡UC Santa Barbara, pagani@ucsb.edu
§Joint first authors

***Abstract*—Backward-edge control-flow hijacking via stack buffer overflow is the holy grail of software exploitation. The ability to directly control critical stack data and the hijacked target makes this exploitation strategy particularly appealing for attackers. As a result, the community has deployed strong backward-edge protections such as shadow stacks or stack canaries, forcing attackers to resort to less ideal e.g., heap-based exploitation strategies. However, such mitigations commonly rely on one key assumption, namely an attacker relying on return address corruption to *directly* hijack control flow upon function return.**

In this paper, we present *exceptions* to this assumption and show attacks based on backward-edge control-flow hijacking *without* the direct hijacking are possible. Specifically, we demonstrate that stack corruption can cause exception handling to act as a *confused deputy* and mount backward-edge control-flow hijacking attacks on the attacker's behalf. This strategy provides overlooked opportunities to divert execution to attacker-controlled *catch handlers* (a paradigm we term Catch Handler Oriented Programming or CHOP) and craft powerful primitives such as arbitrary code execution or arbitrary memory writes. We find CHOP-style attacks to work across multiple platforms (Linux, Windows, macOS, Android and iOS). To analyze the uncovered attack surface, we survey popular open-source packages and study the applicability of the proposed exploitation techniques. Our analysis shows that suitable exception handling targets are ubiquitous in C++ programs and exploitable exception handlers are common. We conclude by presenting three end-to-end exploits on real-world software and proposing changes to deployed mitigations to address CHOP.

***Keywords*—Computer security, computer languages, software, computer fault tolerance**

I. Introduction

Although safer programming languages are becoming increasingly popular, C and C++, and Objective-C remain some of the most dominant languages for mobile, embedded, and enterprise applications alike [15]. The enduring dependency on these unsafe low-level languages comes however at a high cost: classic memory corruption vulnerabilities are still painfully common [69] and continue to haunt both developers and end users. High-value targets in this space are stack buffer overflow vulnerabilities, which have historically enabled "convenient" backward-edge control-flow hijacking attacks corrupting critical control (i.e. return address) and non-control (e.g., pointers in locals and saved registers) data on the stack.

In response, researchers have devised strong mitigations such as stack canaries [20] and shadow stacks [23] to protect backward-edge integrity and cripple exploits. This effort has led attackers to resort to less-than-ideal (e.g., heap-based) strategies, either towards more elaborate exploitation techniques [62] or to circumvent backward-edge protection [19]. Indeed, the latter is generally assumed to be nontrivial since deployed mitigations enforce a key invariant: a corrupted return address cannot *directly* hijack the backward-edge control flow upon function return.

In this paper, we show this invariant insufficiently mitigates backward-edge control-flow hijacking. Our key insight is that corrupted return addresses and stack frame data cannot only be abused for exploitation purposes on the return path, but also on the stack unwinding path executed during *exception handling*. The unwinder can act as a *confused deputy* to mount backward-edge control-flow hijacks on the attacker's behalf.

This is possible since the unwinder needs to find an appropriate handler to transfer control flow to when handling an exception. As we will show, this process largely depends on the return address and other data located on the stack. As a result, using a stack buffer overflow, an attacker can control such data, luring the unwinder into diverting control flow to attacker-controlled *catch handlers*. We show that this paradigm, which we term Catch Handler Oriented Programming (CHOP), can be used to craft powerful primitives such as arbitrary code execution or arbitrary memory writes.

Additionally, we show that attacker-controlled critical data on the stack is only subject to lose sanity checks on the unwinding path (i.e., the raised exception type), even with backward-edge protections in place. Indeed, to our surprise, while Structured Exception Handling (SEH) exploitation is a well-studied and mitigated attack vector on Microsoft Windows [71], exception handling support on other systems has been a largely overlooked attack surface—and so did widely deployed backward-edge protection mechanisms.

Based on these observations, we analyze the resulting attack surface of CHOP in more detail. More specifically, we identify the available *confusion primitives* arising from the corruption of data used by the unwinder. Then, we discuss the capabilities of *gadgets* exploitable by an attacker after

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.232295
www.ndss-symposium.org

diverting the unwinding logic. Finally, we build on the gathered insights to present multiple novel CHOP techniques combining confusion primitives and gadgets for exploitation.

To demonstrate that these attacks are a realistic threat to the existing software landscape, we analyze real-world C++ binaries to assess the prevalence of exception handling semantics. We further analyze CHOP gadgets using static taint tracking to gain insights over common attacker capabilities after successful confusion of the unwinder. Our analysis shows the usage of exceptions is widespread in C++ software, and CHOP provides powerful building blocks for exploitation. We also demonstrate end-to-end CHOP-style attacks on three real-world vulnerabilities, which, to the best of our knowledge, would not be exploitable without our techniques. Finally, we show that even recent defenses, such as hardware shadow stacks, fail to mitigate CHOP-style attacks. Overall, CHOP attacks are a valuable addition to the attacker's arsenal, evidencing an important gap (i.e., the unwinding path) in the current mitigation space—as also acknowledged by the vendors in response to our vulnerability disclosure. Moving forward, we discuss possible defenses, based on extending the invariants enforced by state-of-the-art backward-edge mitigations to the stack unwinding path.

In summary, our paper makes the following contributions:

- We present the unwinding logic deployed in modern C++ programs, pinpointing operations treating critical, attacker-targeted data as trusted.
- We demonstrate CHOP, a novel exploitation paradigm based on unwinding process hijacking and describe resulting attacker capabilities.
- We conduct a large-scale study on popular C++ programs to assess the potential attack surface of CHOP attacks.
- We show that CHOP applies across all widely used operating systems and platforms.
- We showcase end-to-end exploits against real-world software and argue that unwinders should become aware of CHOP-style attacks.

II. Background

A. Exceptional Control Flow

Exceptional control flow denotes the control flow transitions of a program experiencing a fault condition or other foreseen or unforeseen exceptional situations, triggering *exception handling* [21], [22]. Many modern programming languages follow the exception handling paradigms introduced in Ada [7], [50]. In this paradigm, a program may define exception types, *throw*[1] an instance of such types upon detection of an exceptional situation, and define Exception Handlers (EHs) that are executed for a specific exception. Furthermore, EHs are bound to specific code ranges[2], and within the abstract machine defined by the programming language, exceptional control flow transitions are only allowed between a *try block* and its associated EH. A raised exception interrupts normal execution and

```
void bar() {
    throw std::exception(); ❶
    return; ❷
}
void foo() {
    try { ❸
        bar();
        ❹
    }
    catch ( std::exception &e ) {
        ❺
    }
}
```

Listing 1: C++ example with exceptional control flow.

immediately transfers control flow to the corresponding EH. Exception propagation may cross subroutine calls, in which case the stack needs to be *unwound*. Depending on language-defined semantics, the EH may transfer control flow back to the site where the exception was raised (*resumption semantics*) or to code following the EH (*termination semantics*). For the sake of clarity, in this paper we focus on termination semantics since modern programming languages with Ada-style exception handling, such as C++, almost ubiquitously implement these semantics [7].

The C++ program in Listing 1 exemplifies how exceptions are thrown and propagated. The subroutine `foo` wraps a call to `bar` into a *try block* ❸, which in turn *raises* ❶ an instance of `std::exception`. This directly transfers the control flow to the corresponding EH ❺. Due to C++ termination semantics, any code following the point the exception is raised (❹, ❷) is not executed. Crucially, the *return* statement nominally transferring the control flow back to `foo` along the *backward edge* ❷ is skipped.

B. Backwards-Edge Protections

1) Stack Canaries: A widely deployed [31], [9], [52], [68] countermeasure against backward-edge control-flow hijacking based on contiguous stack out-of-bounds writes [3] is *StackGuard* [20]. StackGuard places *stack canaries* or *cookies* on the stack before the saved return address in the function prologue and emits a check of the canary value in the function epilogue prior to the return instruction. This protects nominal control flow through subroutine returns, since the integrity of the stack cookie is checked before transferring the control flow to the return address saved on the stack.

2) Shadow Stacks: A second form of backward-edge protection is offered by shadow stacks [13], [17], [23], [24], [74]. The core idea behind shadow stacks is to save return addresses on a separate stack inaccessible to the attacker via stack-based buffer overflows. Return addresses are typically replicated between the main and the shadow stack and either used for exploit detection or protection by restoring the saved return address from the shadow stack on return. While different flavors of shadow stacks have been proposed, a key difference is how the program stack is mapped to the shadow stack. *Direct* mapping schemes replicate the structure of the program stack, while *indirect* schemes are designed to be more compact and only save the return address onto the shadow stack [23].

[1]The terms *raise* and *throw* are used interchangeably in literature and programming language implementations.

[2]The term *try block* refers to the validity scope of an EH.

Stack unwinding—e.g. when transferring the control flow to an EH—is particularly challenging for indirect shadow stacks since the main stack and the shadow stack need to be kept in sync [13]. In particular, it is crucial that any saved Instruction Pointer (IP) on the main stack as well as the stack frame used for unwinding are validated against the shadow stack. As we will show in section VII-E, failure to perform such a check can lead to exploitable conditions depending on the shadow stack implementation.

III. Exception Handling Internals

While nominal control flow (including subroutine returns) is implemented architecturally, programming languages rely on runtime support to transition the control flow from the site of raising an exception to the exception handler. This is generally implemented with the following components:

- The Exception Handling ABI, which defines the unwinding metadata embedded within the application binaries.
- The language- and implementation-specific *personality routines*.
- The unwinder, which implements the Exception Handling ABI to unwind the stack and invokes the personality routines.

A. Stack Unwinding on Unix-based Systems

On UNIX-based systems, the Itanium C++ ABI [18] has found widespread adoption. This ABI describes runtime-assisted in-process stack unwinding and defines standard interfaces for the *unwinder*, which must be implemented by libraries and language runtimes to provide compliant exception handling. The unwinding process as defined by the Itanium C++ ABI can be summarized in two phases: the *search phase* and the *cleanup phase*.

Search Phase. Upon raising of an exception, the unwinder must understand whether the exception can be handled by any exception handler. The library begins this process by examining the current IP and by retrieving the associated unwinding metadata, e.g., the Frame Description Entry (FDE). The latter contains—possibly through an additional level of indirection—a pointer to the language-specific *personality routine* function, as well as a pointer to a Language-Specific Data Area (LSDA). Different call frames may use different personality routines if their relative functions are implemented in different languages. The unwinder will invoke the personality routine for the current call frame, which will retrieve and parse the LSDA to determine whether the current call frame contains a valid exception handler for the specific exception type thrown. To achieve this, the current IP is compared against an ordered list of *call-site ranges*. These ranges are associated with a list of exceptions' types that can be handled, and with their corresponding landing pads. If no handler is found, the address of the previous stack frame is calculated using the call frame size encoded in the unwinding metadata. The saved IP of this new call frame is retrieved and the process is repeated until either a call frame with a valid handler is found or the stack is exhausted. In the latter case, a default handler terminating the program is usually invoked.

During this first phase, the stack is merely examined to find a valid exception handler. The existing stack frames are left unchanged. While this two-step process may seem suboptimal at first, it allows the default handler to access the original stack trace or even start a debugger at the exception site.

Cleanup Phase. Similarly to the previous phase, the personality routine is repeatedly invoked during the *cleanup phase*, starting from the exception site stack frame. However, this time the stack frame is *unwound* at each invocation, that is, the stack pointer is adjusted. Depending on the LSDA, the personality routine may transfer control to a "landing pad" after restoring the previously unwound frame's callee-saved registers from the stack. Landing pads are code areas implementing either the exception handler—which effectively terminates unwinding—or cleanup handlers which perform frame-specific cleanup, such as running destructors of local objects before continuing unwinding.

Exception Application-Binary Interfaces (ABIs). The SystemV AMD64 ABI draft [49] defines the implementation of the *unwinder* for AMD64 systems as well as data structures within the ELF file format. Unwinding information is stored in FDE elements within the `.eh_frame` section, which contain DWARF programs that specify how the previous call frame can be restored. The FDE elements refer to the personality routine through Common Information Entry (CIE) elements, which correspond loosely to compilation units.

On Darwin, the Mach-O file format is used with Apple's Compact Unwinding information [5], [8]. The call frame metadata is defined in table elements in the `.debug_frame` section, which may optionally refer to a DWARF program within a FDE in the `.eh_frame` section. Finally, ARM defines Exception Handling Tables in [6] for 32-bit ARM which are stored in the `.ARM.extab` and `.ARM.exidx` ELF sections, but uses SystemV AMD64 ABI-compatible exception metadata on AArch64.

Language Runtimes. The language runtimes bundled with the GNU Compiler Collection (GCC) [31] and clang[42] compilers, contain language-specific personality routines identified by their function name prefix and suffix for C, C++, Go, ADA, and Objective C. For each language, different unwinding methods are implemented:

DW2 uses unwinding information encoded in DWARF [29] tables. Denoted using a `_v0` suffix.
SJLJ uses `setjmp(3)` and `longjmp(3)` [2], [1] to restore frame state. Denoted using a `_sj0` suffix.
SEH uses Windows Structured Exception Handling to restore frame state. Denoted using a `_imp` or `_seh0` suffix.

For example, the DWARF-based personality function for C++ is called `__gxx_personality_v0`. Without loss of generality, the analysis presented in this paper is restricted to exception handling using this personality function, i.e. C++ binaries with DWARF unwind metadata. However, similar concepts apply to SJLJ and SEH exception handling, as well as other languages – the only difference between implementations here is the exact semantics of exception types and matching against catch clauses.

Unwinding Implementations. For modern Unix systems, three different unwinding libraries exist: *libgcc* [31], *llvm-libunwind* [45] and *nongnu-libunwind* [55]. While the internal implementations of the libraries are different across libraries (e.g., nongnu-libunwind supports remote unwinding, while llvm-libunwind only supports local unwinding), every library provides the implementation of the Itanium C++ ABI.

B. Unwinding from an Attackers Perspective

Unwinders operate on the call stack to determine which handlers to invoke and implicitly trust the stack contents. Call stacks therefore represent an input program for the unwinding state machine, and an invalid call stack therefore represents a *weird machine* [12], [28], [27] in the context of the language abstract machine. Attackers who are able to (1) corrupt stack data and (2) manipulate the program to throw an exception can trick the unwinder to operate on their data both on Windows and Unix-based systems. In the remainder of the paper, we will not only showcase the issues arising from exception handling on attacker-controlled data, but also find that attacks abusing exception handling are a realistic threat.

IV. THREAT MODEL

We consider an attacker seeking to lift a call stack corruption (e.g., buffer overflow) vulnerability into powerful primitives such as arbitrary control-flow hijacking or memory writes for exploitation purposes. We assume the attacker is targeting a specific program performing exception handling, triggering stack corruption and making the program throw an exception afterwards, either directly in the vulnerable function or any callee after corruption. We start with an unconstrained stack corruption primitive allowing the attacker to overwrite arbitrary data on the stack and later discuss refinements. We assume backward-edge mitigations such as StackGuard [20] or shadow stacks [13], [17], [23], [24], [74] are in place to prevent traditional Return-Oriented Programming (ROP) [64] or return-to-libc [65] attacks. However, we generally assume the attacker has orthogonal means to bypass Address-Space Layout Randomization (ASLR) [25], for instance by means of traditional pointer leaks [63], side-channel leaks [30], [37], entropy exhaustion attacks [10], generative approaches [34], or partial overwrites [66].

V. HIJACKING EXCEPTIONAL CONTROL FLOW

In this section, we introduce the concept of Catch Handler Oriented Programming (CHOP) attacks, a novel exploitation technique abusing stack corruptions in conjunction with exception handling logic. As we will show, such primitive gives rise to multiple possible exploitation scenarios and strategies, depending on the particular characteristics of the corruption.

Terminology. For describing CHOP attacks, we introduce two terms. *Confusion Primitives* refer to the outcome of an attacker-controlled stack corruption and describe the capabilities of an attacker to manipulate the unwinding process. *Gadgets* are exception and cleanup handlers executed after unwinding and provide exploitation capabilities to the attacker. Based on the confusion primitive, this can either be the legitimate handler operating on attacker-controlled data, or an illegitimate, attacker-chosen handler.

Attack Overview. At a high level, CHOP attacks typically first corrupt the saved return pointer to *confuse* the unwinder and lure it into transferring control flow to unintended handling code. In other words, the unwinder is forced to act as a *confused deputy* and hijack control flow on the attacker's behalf. Next, the attacker forces the execution of attacker-specified *gadgets* using controlled data from the stack. In the following, we present the various confusion primitives and potential gadget capabilities, as well as provide practical attack examples.

A. Confusion Primitives

As highlighted in Section III, when an exception is raised after a stack-based buffer overflow, the unwinder operates on attacker-controlled data. This allows CHOP attacks to implement a variety of *confusion primitives*.

Exception Handler Landing Pad Confusion. By corrupting the saved return address to point inside a target call-site range (III-A) and the adjacent stack frame, the attacker can lure the unwinder into transferring control flow to an arbitrary exception handler, given the exception types are compatible. The targeted exception handler will interpret the adjacent attacker-controlled stack frame as its own stack frame and be forced to operate on attacker-controlled data. Additionally, depending on the exception handler, callee-saved registers may be restored by the unwinder from potentially attacker-controlled stack locations. As we will show, exception handler landing pad confusion allows an attacker to craft powerful exploits by diverting control flow.

Cleanup Handler Landing Pad Confusion. A cleanup handler landing pad confusion primitive is similar to its EH counterpart, except the attacker corrupts the saved return address with a value corresponding to a call-site range for a cleanup handler. As long as a valid exception handler for the thrown exception is found further up in the unwound stack, this cleanup handler will be executed during the cleanup phase of the unwinding process, before the actual exception handler is called. As most functions with C++ objects on the stack typically require cleanup handlers, this confusion primitive drastically increases the amount of available gadgets for an attacker.

SigReturn Frame Confusion. This primitive lures the unwinder into executing an intermediate SigReturn handler with attacker-controlled data. This is possible due to the unwinder's necessity to handle SigReturn frames during unwinding. Such frames are pushed onto the stack by the kernel when a signal is delivered to, and handled by, a process. To identify those cases, `libgcc`'s unwinder dereferences the saved IP of each call frame during unwinding. If the memory at the target of the IP matches the encoding of the `rt_sigreturn` syscall, the following stack frame is interpreted as a SigReturn frame. Similarly to SigReturn-Oriented Programming (SROP) [11], placing a crafted SigReturn frame on the stack and corrupting the saved return address to memory containing the encoding of the `rt_sigreturn` yields control over all registers during unwinding. In particular, the stack pointer is now controlled, which allows pivoting the subsequent unwinding to an attacker-controlled location. This is a powerful primitive if the stack corruption primitive used is space-constrained or has other constraints (e.g, character restriction during overflow). The targeted exception handler is controlled by specifying the corresponding IP in the SigReturn frame.

Callee-Saved Register Confusion. Depending on the type of memory corruption, an attacker may further extend their

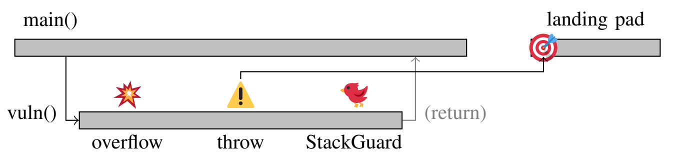

Fig. 1: Sequence diagram of vulnerable code. After overflowing a stack buffer in a vulnerable function, an exception is thrown, diverting execution to a landing pad higher up the call stack. The normal return is not executed, and the canary check of StackGuard is bypassed.

control by corrupting callee-saved registers. Callees typically preserve a set of registers on behalf of their caller. During unwinding, callee-saved registers are restored from the contents of the stack. This means that after corruption, an attacker can enter the legitimate cleanup or exception handler with registers under their control, enabling CHOP attacks without a control flow diversion requiring overflow of a saved return pointer.

Cross-DSO Applicability. Shared libraries often throw exceptions without having according handlers registered for it, as normally the program invoking the library should handle the exception. While this is a sensible approach from a programmer's perspective, it opens additional possibilities for an attacker, as this behavior has two implications: (1) the unwinder performs its operations *regardless* of whether a try/catch block for a given exception exists, and (2) exceptions thrown in one shared object in the virtual address space must be catchable in a different object. This means that the aforementioned confusion primitives are applicable across the full virtual address space. In other words, an attacker can confuse the unwinder after stack corruption by supplying addresses pointing to landing pads or sigreturn encodings in any shared object mapped in its virtual address space. We want to stress that this eases exploitation tremendously, as having access to *all* catch and cleanup handlers drastically increases the amount of viable landing pads for successful attacks.

B. Gadget Capabilities

Similar to more traditional ROP attacks, we refer to chunks of code abusable by an attacker as gadgets. For CHOP attacks, these gadgets are exception handlers and cleanup handlers executed during either exception handling or during the cleanup phase of the unwinding process. In the following, we outline common capabilities provided by these handlers.

Backwards-edge Control-flow Hijacking. Modern compilers often refrain from embedding stack canaries in every function for performance reasons—and similar optimizations are included in common software shadow stack implementations [16]. Instead, canaries are only deployed when the compiler deems a function likely to be vulnerable to an overflow, such as functions containing stack-based arrays. As a result, exception handlers contain stack canary checks *only* if the functions they belong to have such checks. Similarly, the unwinding process does *not* check for overwritten stack canaries. Therefore, using landing pad confusion (Section V-A) to divert from a throwing function with a stack canary check to a landing pad without a stack canary check (cf. Figure 1)

```
#include <memory>
void func() {
    std::unique_ptr<char> ptr(new char[20]());
    vuln(); // stack-based buffer overflow
    throws(); // throws an exception
}
```

Listing 2: Code vulnerable to arbitrary free through a smart pointer.

re-enables traditional ROP [64] and return-to-libc [65] attacks, even with backward-edge protections in place.

Forward-edge Control-flow Hijacking. Exception and cleanup handlers may perform indirect calls based on values read from the stack. This happens commonly when destructors are called for polymorphic objects through a vtable. By crafting a counterfeit vtable holding a target pointer and overwriting a stack object's vtable pointer, an attacker can abuse this mechanism for arbitrary forward-edge control flow hijacks. If forward-edge (vtable) protection mechanisms are in place, the attacker can also exploit the ability to control stack data to mount more sophisticated COOP-style attacks by means of counterfeit objects reusing existing valid vtables [62].

Arbitrary Free. Cleanup handlers are emitted to invoke destructors for objects on the stack. In many cases, this involves calling the `delete` operator, which wraps a `free` call on addresses stored on the stack. An example of this pattern are the widely used C++ magic pointers. Consider the code in Listing 2. The cleanup handler emitted for `func` contains a call to `std::unique_ptr<char>::~unique_ptr`, which in turn calls `operator delete`. With such patterns, an attacker can easily exploit an arbitrary free to turn a pointer of choice into a dangling one, escalating to an arbitrary use-after-free—a powerful exploitation primitive [66].

Arbitrary Write. The direct corruption of the stack frame of the exception handler, combined with the control over callee-saved registers oftentimes yields further powerful primitives depending on the exception handler. In particular, in many cases, we noticed that exception handlers may use stack values and callee-saved registers to store values in memory. As these are again attacker-controlled, such patterns can easily be exploited to craft arbitrary write primitives. Arbitrary memory writes are powerful exploitation primitives and can be exploited to bypass advanced mitigations [70] and mount data-only attacks [40].

C. Attack Examples

By combining the different confusion primitives with the different gadget capabilities, an attacker can craft versatile exploitation strategies. In the following, we discuss:example attacks showcasing the applicability of CHOP-style attacks in different scenarios. Table I provides an overview of the discussed attack scenarios, along with the required attacker prerequisites and information on effectively bypassed mitigations by the attack. Additionally, we visualize the stack configuration needed to mount each attack in Appendix A.

Golden Gadget. Arguably, the most direct exploitation technique via CHOP is forward-edge control-flow hijacking, as it allows an attacker to redirect execution to arbitrary locations,

TABLE I: Different Attack scenarios, their required corruption primitives, and subsequently bypassed mitigations.

Attack Scenario	Confusion Primitive	Gadget Capabilities	Corruption		Bypassed Protection			
			Saved IP	Gadget Stack	Canaries	CFI	ASLR	Shadow Stacks[a]
Golden Gadget	Exception Handler	Fwd. Control-Flow Hijack	✓	✓	✓	✗	✗	✓
Pivot-to-ROP	Exception Handler	Bkd. Control-Flow Hijack	✓	✓	✓	✗	✗	✗
Data-Only	Callee-saved Reg.	Arbitrary R/W	✗	✗	✓	✓	✓[b]	✓
Cleanup-Hdlr. Chaining	Cleanup Handler	Gadget-Dependent	✓	✓	✓	✓	✗	✓
SigReturn-to-ROP	SigReturn Frame	Bkd. Control-Flow Hijack	✓	✗	✓	✗	✗	✓

[a] we assume a traditional shadow stack implementation, such as ShadowCallStack [48].
[b] depends on frame layout.

```
void __cxa_call_unexpected (void *exc_obj_in)
{
  /* ... */
  xh_terminate_handler = xh->terminateHandler; ❻
  /* ... */
  __try
    { /* ... */ }
  __catch(... ❼)
    {
      /* ... */
      __terminate (xh_terminate_handler ❽);
    }
    /* ... */
  }

void __terminate (std::terminate_handler handler)
    throw ()
{
__try
  {
      handler (); ❾
      std::abort ();
  }
__catch(...)
  { std::abort (); }
}
```

Listing 3: Excerpts from libstdc++-v3 showing an example of a "golden handler". As `xh_terminate_handler` is a local variable ❻, it gets restored from the (attacker-corrupted) stack after entering the exception handler, passed to `__terminate` ❽, and finally invoked ❾. In short, this catch handler allows for an effortless forward-edge control-flow hijacking, when the attacker controls the stack location corresponding to `xh_terminate_handler`.

without the restriction of stitching together exception or catch handler gadgets. For this purpose, while analyzing the attack surface in widely used C++ binaries, we found what could be considered a "golden handler" inside `libstdc++`, the standard C++ library for Linux. This handler (see Listings 3) is a catch-all handler ❼ and allows arbitrary control-flow hijacking via an indirect call based on the data located on the corrupted stack. Hence, by corrupting the saved return address to refer to the golden handler in `libstdc++` and providing a controlled stack frame to the handler, an attacker can divert execution to any location in memory.

Pivot-to-ROP. As described in Section V-B, backwards-edge protection schemes such as stack canaries may only protect functions which are deemed as unsafe. Thus, depending on application logic, programs may include gadgets—i.e., exception handlers—on which the otherwise deployed backwards-edge protection mechanism are omitted. Given that the base attack primitive for CHOP attacks is a stack-based buffer overflow allowing an attacker to control stack data, this directly provides traditional ROP capabilities, even when backwards-edge protections are present in the vulnerable function.

Data-Only Corruptions. The stack frame of the vulnerable function contains not only local variables, but also callee-saved registers just before the saved return address. If any cleanup handler is registered as a landing pad for the saved return, such handler will operate on the local data. Likewise, if the correct landing pad is an exception handler capable of handling the raised exception, this exception handler will execute with callee-saved registered restored from the stack. This means that by overwriting just the local stack frame of the vulnerable function and keeping the saved return address intact, an attacker can create data-only attacks forcing *valid* cleanup and exception handlers to operate on attacker-controlled data. Moreover, since the attacker does not need to overwrite the return address to confuse the unwinder and execute gadgets, an ASLR bypass may not be necessary. For instance, an attacker may lure a catch handler into crafting a stack-relative read gadget and leak randomized pointers. The latter could then be used in arbitrary memory read or write gadgets to take control over the binary without ever leaving the boundaries of the legitimate control flow, in a data-only fashion.

Cleanup-Handler Chaining. If an attacker can corrupt a large part of the stack, they can also opt for chaining multiple fake stack frames with saved return addresses referencing call sites with registered cleanup handlers. As long as one stack frame further up the call chain is capable of handling the raised exception, all these cleanup handlers will be executed. This way, an attacker can stitch multiple gadgets together, similar to traditional ROP [64] attacks.

Besides potentially providing enough primitives for exploitation on their own, cleanup handlers can also be helpful for exploiting landing pad confusions. Since these handlers are inserted into the call chain specifically to call destructors for variables with automatic storage (i.e. stored on the stack), common gadgets obtained are destructor calls of attacker-controlled objects, leading, as mentioned earlier, to arbitrary use-after-free primitives. Furthermore, when unwinding over these stack frames, the callee-saved registers will be restored from the stack and are thus attacker-controlled, potentially extending the number of attacker-controlled registers when entering the exception handler.

SigReturn-to-ROP. Sometimes, an attacker may control the contents of a large buffer at a known location independently from the actual corruption. In those cases, a viable exploitation strategy is to pivot the unwinding logic to this buffer. This

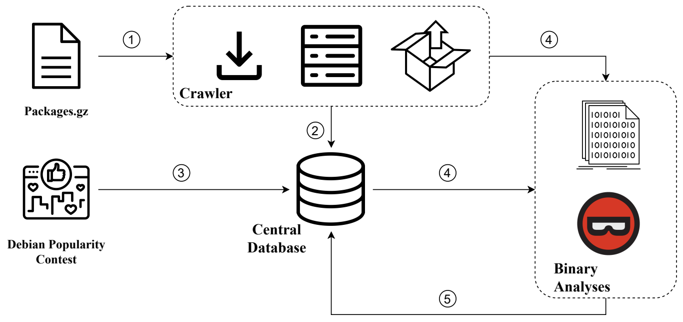

Fig. 2: Overview over our dataset creation and analysis process. Packages are crawled from the `main` debian package index and metadata over the extracted files are stored in a database. For analysis, we select relevant binaries based on the debian popularity contest and store analysis results back to the database.

strategy can be easily implemented by means of a SigReturn frame confusion primitive. This allows an attacker to pivot the unwinding process from the original stack to the fully attacker-controlled buffer. This technique is especially powerful when paired with gadgets that allow an attacker to ROP, as the ROP chain is now located in the attacker-controlled buffer, rather than the actual stack.

VI. Attack Surface & Gadget Analysis

In the previous section, we examined the security implications of unwinding and exception handling on attacker-controlled data. However, this alone provides little insight into the actual attack surface present in real-world software, both in terms of confusion primitives and gadget capabilities. To better assess the actual impact, we took a closer look at popular C++ binaries.

While investigating the prevalence of exception handling code would in principle be possible via source-level analysis, we operate on already compiled binary executables and libraries since the low-level semantics depend on compiler internals such as register allocation and automatic handler generation. Most of our automated analyses are based on the Binary Ninja reverse engineering framework [72]. In total, our analyses are implemented in roughly 4,000 lines of Python, 3,400 lines of C++, and 900 lines of Rust code, accounting for both standalone tools and Binary Ninja plugins.

A. Dataset Collection & Processing

Our dataset collection and analysis approach is shown in Figure 2. For dataset creation, we crawl the main AMD64 repository of Debian Buster (①). We use the official Debian `packages` file[3] to obtain download links for all packages. After downloading the packages, we unpack them and collect metadata of the extracted files, such as file name, associated package, file type or original directory location. The collected metadata is stored in a PostgreSQL database (②).

[3]http://ftp.us.debian.org/debian/dists/buster/main/binary-amd64/Packages.gz (retrieved 2021-10-09).

To prioritize which packages to analyze, we enrich the metadata with the number of installations for each package, as reported by the Debian popularity contest (③). Extracted files are then forwarded to a Binary Ninja based analysis pipeline (④), which we describe in more detail in the following sections. The analysis results are fed back to the database (⑤), allowing easy cross-correlation of different analysis passes.

B. Identifying Target Binaries

The most intuitive approach to select packages that handle or throw exceptions is to check whether they list `libstdc++` as a dependency. However, we quickly note that this filter is inaccurate for two reasons. First, packages often bundle different binaries and libraries together, and a dependency relation exists as soon as a single one of those requires the C++ standard library. Second, this library is not only used by binaries compiled from C++, but also required for binaries generated from other languages, such as Objective-C and Rust.

Thus, we decide to use a finer-grained method to select C++ binaries that include exception handling. After unpacking a Debian package, we first of all select only binary programs and shared libraries. We then analyze these programs and libraries using Binary Ninja and flag them as programs with exception handling semantics if they fulfill either one of the following conditions: 1) the ELF file contains the `.gcc_except_table` section or 2) the code calls any of the `libstdc++` functions to raise exceptions (e.g., `__cxa_throw`). The first condition indicates that a binary catches exceptions or that it has cleanup handlers, while the second one selects binaries that explicitly throw exceptions.

C. Attack Surface

CHOP attacks pose very explicit requirements to an attacker for launching an attack: (1) the possibility to trigger a stack-based spatial memory corruption and (2) the capability of throwing an exception to force the unwinder to operate on the corrupted data. In the following, we describe our methodology to estimate the prevalence of such initial primitives in C++ software and later present experimental results.

Memory Corruptions. Stack-based memory corruptions have been studied extensively in previous research and identifying them automatically is an orthogonal problem to our work. To assess the attack surface for CHOP attacks, we are more interested in the question whether a function is *likely* to be corruptible and, thus, serve as an entry vector for an attacker.

As a proxy for this property, we rely on the binary artifacts produced by modern compilers. To defend against sequential overflows, compilers embed stack canaries for functions they deem unsafe. In other words, canaries are only present in functions with potentially vulnerable local variables, such as stack arrays. Hence, knowledge of which functions deploy canaries and which do not provides a good estimation for possible attacker entry points. To identify the functions deploying stack canaries, our analysis simply checks whether a given function calls `__stack_chk_fail`—as generated by gcc.

Raising Exceptions. Evaluating how likely an attacker can raise exceptions after corruption requires a more sophisticated analysis strategy. The reason for this is that a hijackable throw may occur in any subsequent called function after corruption, as long as the vulnerable function did not return. We therefore need to analyze not only whether a function itself can throw an exception, but also any callee in the call tree with the potentially vulnerable function as root. Additionally, since exception handling and unwinding is not local to objects in the address space, we also need to expand this analysis to resolve dependencies across Dynamic Shared Object (DSO) (i.e., considering throws from imported functions).

To account for this, we order binaries topologically based on the DSO relationship between them and store the analysis result for each binary in the database. Then, whenever we encounter a call to an imported function, we fetch the analysis summary from the previously analyzed DSO. The resulting information on whether one of the callees of the imported function can raise an exception is then propagated to the function under analysis.

To create the exception summaries over a function, we developed a special analysis pass. For each function in a binary, it traverses the call graph in a breadth-first fashion, until a given threshold depth is reached[4]. While traversing the call graph, we collect further callees and determine if they can throw an exception by checking whether a call to `cxa_throw` exists. This function is the default interface in `libstdc++` for raising an exception and wraps the unwinder's language-independent `_Unwind_RaiseException` function, which then in turn invokes the unwinder.

D. Gadget Analysis

To characterize the impact of CHOP-style attacks, we also want to study the corruption primitives that CHOP gadgets provide to an attacker. Our gadget analysis starts by identifying all the gadgets available to an attacker, assuming an arbitrary unwinding-based control-flow hijack. It then leverages static taint tracking to obtain insights into the primitives provided by a specific gadget.

Gadgets. To find every CHOP-gadgets present in a binary, we identify all the catch and cleanup handlers by parsing the exception metadata contained in the `.eh_frame` and `.gcc_except_table` sections of the binary. These sections hold the unwinding metadata required to associate landing pads for catch and cleanup handlers with their parent function based on the provided call-site information. We note that the identification of handlers is a one-time pre-exploitation step as their location is static with regard to the binary base.

Attacker Model. When analyzing the capabilities provided by individual gadgets, we assume that the attacker is able to corrupt the stack and overwrite the backward edge to confuse the unwinder. In our attacker model we assume only the callee-saved registers to be controlled by an attacker. This model is practical for two reasons: First, since we don't assume the gadget stack is under an attacker's control, we can loosen the requirements on the initial memory corruption. Second, callee-saved registers are restored by the unwinder during the cleanup phase and stitching together different confusion primitives can easily provide control over all callee-saved registers.

[4]In our experiments we limit the analysis to a depth of 7 to guarantee a timely termination of the analysis.

Taint Analysis. To assess the impact of attacker-controlled registers in these gadgets, we developed a static taint analysis engine based on Binary Ninja's High-Level IL (HLIL) intermediate representation. Developing a taint analysis at this level of abstraction has several benefits, from being able to build on top of Binary Ninja's Static Single Assignment (SSA), to leveraging built-in analysis passes (e.g., constant propagation, dead-code elimination, etc.). While only highlighting the most important points of our analysis here, we describe the taint algorithms in more detail in Appendix B.

When processing a gadget, we taint all callee-saved registers that have been read before being written, by marking their corresponding SSA variables as taint sources. Our taint analysis then follows the path from a gadget's entry until the end of its respective parent function *without* analyzing called functions. Whenever we encounter an expression marked as a source or derived from tainted data, we propagate its taint to its target destination. In case multiple variables are used by a single expression, we propagate their combined taint to the destination. More precisely, taint propagation happens on the following HLIL abstractions: `HLIL_INIT_SSA`, which initializes a variable based on an expression; `HLIL_ASSIGN`, which models register-to-register and register-to-stack operations; `HLIL_ASSIGN_UNPACK`, which propagates a function's return to a SSA variable; and `HLIL_ASSIGN_MEM_SSA`, which models stores to memory (i.e., memory dereferences).

While the taint propagates through a gadget's instructions, our analysis defines the following sinks which model the gadget capabilities introduced in Section V-B:

1) **Arbitrary free sinks**, which model the capability to call a deallocation routine on an attacker-controlled pointer.
2) **Forward-edge hijacking sinks**, which model a forward-edge control-flow transfer to an attacker-controlled target.
3) **Attacker-controlled write sinks**, where an attacker controls either the location of data to be written, the contents of data to be written, or both.

More precisely, our analysis reports an *arbitrary free* primitive whenever it finds a call to the C++ `delete` operator or the `free` function with tainted parameters. *Control-flow hijacking* primitives are instead reported when our analysis finds an indirect call or jump with a tainted target. Finally, *attacker-controlled writes* are reported when memory operations (e.g., `HLIL_ASSIGN_MEM_SSA`) are performed with a tainted source and/or a tainted destination. The resulting write primitives are further classified in three categories based on the taint assigned to their operands: (a) *write-where* primitives when only the destination is tainted, (b) *write-what* primitives when only the source is tainted, and (c) *write-what-where* primitives when both the destination and the source are tainted.

Gadget Score. Each gadget is augmented with control information (e.g., which callee-saved register controls the target of the forward-edge control flow hijacking gadget) and a *feasibility score* that aids an analyst in choosing a gadget. This score is based on a weighted average of the number of nested branches, nested loops, basic blocks and function calls that the taint analysis encountered before reaching the specific gadget. In other words, the lower the score the higher the chances to reach the gaget's sink.

Our reported sinks are an overapproximation, as we do not deploy any symbolic reasoning (for scalability reasons) to assess the constraints of data controlled by an attacker landing on the taint sinks. However, we manually confirmed that most sinks reported by our engine indeed provide usable primitives for exploitation, assuming control over callee-saved registers. Finally, we want to note that an attacker is likely to also control the stack frame of the gadget which gives additional attack surface—conservatively ignored by our methodology.

VII. Evaluation

In this section, we set out to estimate the prevalence of exceptions in C++ software and their susceptibility to CHOP attacks. We also present three case studies demonstrating the application of our technique to real-world software. Finally, we study the relationship between recent mitigations and CHOP attacks, and whether our findings apply to targets beyond x86-64 based Linux systems. In summary, in the following sections, we aim to answer the following research questions:

RQ1: How prevalent is the usage of exception handling in modern software?

RQ2: How large is the attack surface for CHOP attacks?

RQ3: How powerful are the CHOP gadgets available to an attacker?

RQ4: Can CHOP be used to develop an exploit against real-word software?

RQ5: Which platforms are affected beyond Linux?

RQ6: Are recent mitigations effective against CHOP?

Unless otherwise specified, we run all our analyses on an Ubuntu 20.04.04 LTS virtual machine with 32 vCPUs and 64 Gb of RAM.

A. Exception Handling in C++ Software

As discussed in Section VI-B, to answer **RQ1** we consider multiple proxies. Using the most intuitive approach–i.e. checking whether a package lists `libstdc++` as a dependency—would select 6,533 out of 51,626 available packages on the `main` Debian buster packet index (~12.5%).

Thus, we apply a finer-grained filter (c.f. Section VI-B) to better estimate the prevalence of exception handling in the Debian ecosystem. We unpack the top 1,000 most popular packages and select all binary programs and shared libraries. This corresponds to a total of 3,303 files, roughly the 7.5% of the total number of files. Of these, 56 executable programs and 266 libraries use exception handling totaling 322 binaries (~9.5%) which we use as a data set in the following sections.

To summarize: roughly 10% of programs rely on exception handling. Extrapolated to all software, this shows a significant number of potential targets for CHOP-style attacks.

B. Attack Surface

Mounting CHOP attacks requires an attacker to first exploit a stack-based memory corruption vulnerability and then trigger an exception to be raised. While finding vulnerabilities is a problem outside the scope of this paper, in this section we aim to understand the prevalence of the latter condition, i.e., functions that throw an exception after a potential corruption of data on the stack.

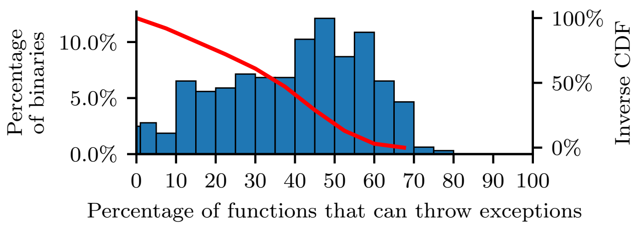

Fig. 3: Histogram of the distribution of functions that can throw exceptions in 322 popular binaries, along with the corresponding inverse cumulative distribution function, showing the number of binaries that have at least a given number of potentially throwing functions.

To answer **RQ2**, we rely on the attack surface analysis presented in Section VI-C. This analysis explores the call-graph of a binary, recursively labeling functions that can throw an exception, along with their use stack of canaries. The histogram in Figure 3 shows the distribution of functions that are capable of throwing an exception. An interesting observation is that exceptions are quite common: half of the binaries have at least 40% potentially throwing functions. From manual analysis, we conclude this is partially caused by shared library functions used in a variety of places. For instance, the standard method `vector::push_back` can throw an exception if the allocation request fails.

To estimate the amount of functions possibly affected by a stack-based memory corruption, we use the presence of stack canaries as proxy, as discussed in Section VI-C. On average, across all binaries under evaluation, 35.4% of all throwing functions check a canary on the return path and, hence, operate on stack buffers. We conclude that, while requiring both a stack-based memory corruption and an exception raising primitive limits the available attack surface, a significant portion of program logic is at reach of CHOP-style attacks.

CHOP-style Vulnerabilities in Open Source Software. To further study the prevalence of stack buffer overflows on applications using exception handling, we investigate all bugs reported by OSS-Fuzz [36] between *May 2016* and *June 2022*. This dataset consists of 38464 crawled bugs reported across 344 software projects written in C++. Based on the bug description in the OSS-Fuzz reports, we select 442 stack-related bugs found in C++ applications that can potentially be exploited with CHOP. These bugs are either buffer overflow writes or buffer overflow reads. While the latter do not directly lead to CHOP-style exploits, we decided to include these bugs as recent research shows that reads can lead to writes [75]. This is based on the insight that out of bounds reads can mask more dangerous side-effects—such as out-of-bounds writes—but sanitizers are typically configured to panic on the first erroneous access.

To select only bugs belonging to projects that use exception handling, we build the vulnerable binary and use the approach discussed in Section VI-B—i.e., checking for the `.gcc_except_table` section and searching for calls

TABLE II: OSS-Fuzz bugs related to stack vulnerabilities triaged based on their appartenance to C++ projects that make use of exceptions.

Category	Total	C++	C++ w/except-handling
Buffer-overflow write	213	192	80
Buffer-overflow read	276	250	110

to `cxa_throw`. Where the build process fails, we search for throw and catch statements in the project's source code. Overall, 309 buffer overflows were classified using the first heuristic, while 133 buffer overflows were classified based on the second. The results are presented in Table II. In particular, we find 80 buffer overflow writes (~38%) and 110 buffer overflow reads (~40%) are present in C++ projects relying on exception handling. This suggests that OSS-Fuzz stack-based buffer overflows have a good chance to be exploitable via CHOP techniques. Finally, we report that in respect to all C++ projects targeted by OSS-Fuzz, 105 projects (~30.5%) use exceptions, and 56 projects (~16.3%) had at least one stack-based buffer overread or overwrite.

C. Gadget Capabilities

To answer **RQ3**, we run the taint analysis described in Section VI-D on our dataset of Debian binaries. The results are summarized in Table III. This table shows, for each gadget type, the percentage of binaries containing a given number of gadgets. The first interesting result is that cleanup handlers often allow arbitrary free–around 79.2% of the analyzed binaries contain at least one instance of this gadget. On the other hand, write-what gadgets—allowing an attacker to write controlled values to an uncontrolled location—are frequently missing in both cleanup handlers (98.1%) and catch handlers (83.5%). Forward-edge control-flow hijacks, however, are commonly found in both cleanup and catch handlers with over 35% of analyzed binaries containing tens to thousands of them. In summary, approximately 90% of analyzed binaries contain at least one arbitrary free, forward-edge control flow hijack, or write-what-where gadget and are therefore very likely to be exploitable with CHOP-based techniques.

Moreover, since the exception handlers present in linked libraries are valid targets to redirect the exceptional control-flow, the number of available gadgets is likely even larger. As a matter of fact, the vast majority of binaries ($\approx 97.2\%$) use one or more libraries with exception handling. The most common library, used by $\approx$ 97.2% of binaries, is the standard C++ library (libstdc++) for which our analysis reports 2113 gadgets. Other common libraries are those included in the *uno-libs3* and *ure* packages, which are used by $\approx 49\%$ and $\approx 18.3\%$ of the binaries in our data set, respectively.

Finally, Figure 4 shows the correlation between the size of the binaries and the number of gadgets. We found a moderate to strong correlation ($\rho = 0.72$). On average, a 1% increase in the size of a binary is expected to add ~1.1% more gadgets.

D. Case Studies

To answer **RQ4**, we present three case studies to show how CHOP can be used to exploit bugs in real-world software, which otherwise would be mitigated by StackGuard. In line

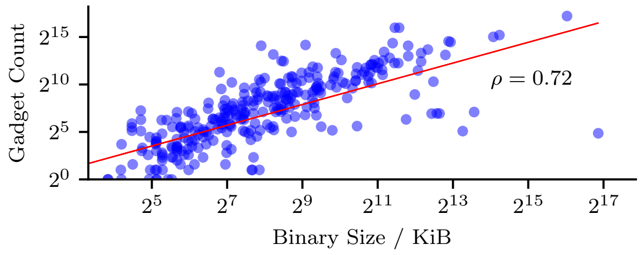

Fig. 4: Scatter plot showing the correlation between binary sizes and number of gadgets, along with `log-log` regression and Spearman's rank correlation coefficient.

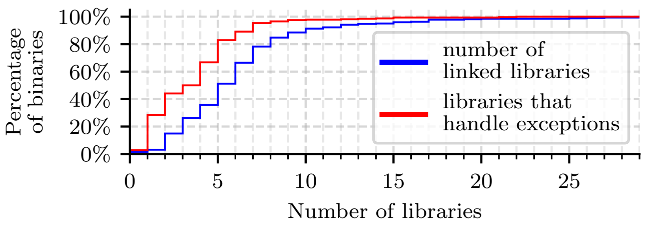

Fig. 5: Cumulative distributions of dependencies (i.e., libraries) and dependencies with exception handling semantics for all binaries in our data set.

with our threat model, we assume ASLR to be broken, either via a single pointer leak or other bypassing techniques [30], [37], [10], [34], [66]. After initial setup of the vulnerable program, none of the exploits required more than two days of development or 150 lines of code.

CVE-2009-4009. Our first case study is a vulnerability in the PowerDNS recursor, and serves as a concise example for CHOP attacks. We target the Debian Lenny package of PowerDNS running on an Ubuntu 20.04.4 system. We apply small modifications to this package to revert the patch fixing the vulnerability and to enable additional hardening measures originally not present (i.e., stack canaries).

Listing 4 shows the vulnerable code path within the `questionExpand` function. This function is called when answering a DNS query, which requires forwarding the query to another DNS server via UDP. When parsing the answer of this second server, stored in the `packet` variable, `questionExpand` is called with a 512 byte fixed-size stack buffer `qname` as parameter. The vulnerability resides in the while loop ➓. The code explicitly checks whether more data has been written into the buffer than it can hold ① and throws a runtime error if that is the case ②. Unfortunately, this check happens only *after* the data was written to the `qname` output buffer ③ during the next loop iteration. This allows an attacker to first overflow the buffer and then raise the `runtime_error` exception.

To launch the exploit, we initiate a DNS query to the PowerDNS Recursor causing the vulnerable binary to issue a request to an attacker-controlled server, which responds with the exploit payload. After a valid DNS response header, the

TABLE III: Relative frequencies of counts of detected gadgets in cleanup and catch handlers of C++ binaries in the top 1000 most popular packages.

Gadget type	**Cleanup Handlers**					**Catch Handlers**				
	0	1 – 10	10^1 – 10^2	10^2 – 10^3	10^3 – 10^4	0	1 – 10	10^1 – 10^2	10^2 – 10^3	10^3 – 10^4
Arbitrary Free	20.8%	24.8%	31.1%	16.5%	6.5%	41.6%	28.6%	20.2%	8.4%	1.2%
Control-flow Hijack	35.4%	19.9%	19.3%	17.7%	6.8%	49.1%	14.0%	14.9%	15.5%	6.2%
Write-What-Where	55.9%	19.3%	15.8%	6.8%	2.2%	53.7%	21.7%	12.1%	7.8%	4.3%
Write-Where	33.9%	31.1%	20.8%	10.9%	2.8%	50.9%	20.2%	15.8%	8.4%	4.3%
Write-What	98.1%	0.3%	0.9%	0.6%	0.0%	83.5%	5.6%	8.1%	2.5%	0.3%

```
void questionExpand(const char* packet, uint16_t len
    , char* qname, int maxlen, uint16_t& type)
{
  type=0;
  const unsigned char* end=(const unsigned char*)
    packet+len;
  unsigned char* lbegin=(unsigned char*)packet+12;
  unsigned char* pos=lbegin;
  unsigned char labellen;

  // 3www4ds9a2nl0
  char *dst=qname;
  char* lend=dst + maxlen;

  /* ... */
  while((labellen=*pos++) && pos < end) { ⓿
    if(dst >= lend) ①
      throw runtime_error("Label length exceeded
    destination length"); ②
    for(;labellen;--labellen)
      *dst++ = *pos++; ③
    *dst++='.';
  }
  /* ... */
}
```

Listing 4: Vulnerable code path for CVE-2009-4009. The contents of `packet` are attacker-controlled. By supplying crafted labels and content, a `runtime_error` can be raised after overflowing the `qname` buffer.

payload almost exceeds the 512 bytes buffer by including four labels with a length of 126 characters each. The following label of length 255 overflows the stack buffer, overwriting both the stack canary and the saved return address. We set the return address to the "golden gadget" (c.f. Section V-C) in `libstdc++`, diverting execution to a catch-all handler which eventually calls `__cxxabiv1::__terminate` with an attacker-controlled handler. This provides a powerful forward-edge control-flow hijacking primitive, as shown in Listing 3. As a target for the hijack, we use a stack-lifting gadget which will then point `$rsp` to our ROP chain, which uses more traditional technique to open a remote shell for the attacker.

CVE-2018-5809. Our second case study targets a vulnerability in LibRaw [43], a popular image processing library, demonstrating looser requirements between the overflowing and throwing code paths. We exploit LibRaw v0.18.8 compiled from source without modifications, running on Ubuntu 20.04.4. As a LibRaw consumer, we use the built-in tool `raw-identify`, which provides information about an image passed as command line argument.

The vulnerability lies in the `parse_exif` function. When parsing a malicious image with a MakerNote EXIF [67] tag, an attacker-controlled amount of data is read from the image into a fixed-size stack buffer in case earlier meta-data indicated that the image was created with a RaspberryPi camera. Interestingly, no code path reachable under these conditions in `parse_exif` throws an exception. However, for images not created by other cameras, a specialized `parse_makernote` function is called instead, which throws a `LIBRAW_EXCEPTION_IO_CORRUPT` exception under special circumstances.

To exploit this vulnerability, we modified an existing image created with a RaspberryPi camera. We modify the MakerNote tag to smash the stack and overwrite the saved return address with an address pointing to a pivot-to-ROP gadget. To trigger the exception, we abuse that (a) EXIF tags are parsed in a loop, and (b) the camera model is stored in a higher stack frame. Since do not hit any backwards-edge protection mechanisms in the loop, we extend the overflow to also overwrite the model, so that a subsequent MakerNote tag is parsed by `parse_makernote`. The second MakerNote tag is crafted such that it triggers the throw, causing the previously corrupted stack to be unwound. It is important to note that the code path reaching the throw is only made reachable through corruption of stack data. Similarly to our first case study, the final ROP chain opens a shell for the attacker.

SCSSU-201801.[5] Our third case study shows the applicability of CHOP to other platforms than Linux. In particular, we exploit the Common Access Card (CAC) module of smartcardservices [32] v2.1.2 running on macOS Sierra v10.12[6].

While retrieving a certificate, data is read from the card into a stack buffer in a loop until the card indicates the end or a communication error occurs. Although exceptions can be thrown in the loop, these are not exploitable via CHOP, due to a catch-all handler within the function, leading to an early return. To exploit the vulnerability, we use an error condition later in the function: The certificate may be compressed. If decompression fails, an exception is thrown, causing the unwinder to operate on attacker-controlled data.

While the exploit could be implemented on a physical smart card, we use the Virtual Smart Card Architecture [57] with `virt_cacard` [58] for emulation, modified to deliver the exploit payload when responding with a certificate. The

[5]This vulnerability is also referred to as CVE-2018-4300 [4] but does not match the corresponding CVE entry.

[6]This version of macOS was chosen for the availability of pre-built smartcardservices installers. The macOS version is not required for exploitability of the bug.

payload overwrites the saved return address with a pointer to a pivot-to-ROP gadget and the ROP chain executes attacker-specified commands on the victim system.

Discussion. All of our exploits rely on control-flow hijacking via CHOP. We note that for every vulnerability, traditional backwards-edge control-flow hijacking would not be possible without circumventing StackGuard. Furthermore, due to the lack of mitigations on the unwinding path, we did not require any data-only attacks, greatly minimizing attack complexity. Additionally, we demonstrated that stack corruptions in C++ may enable triggering exceptions in code-paths *adjacent* to the actual vulnerability. Overall, we believe that these three case studies show the versatility and attainability of CHOP attacks and argue that additional hardening measures for the unwinding logic are needed.

E. Exploitability for additional Platforms and Mitigations

To answer **RQ5**, we run the test program in Appendix C on seven different configurations—Linux (x86_64 and AArch64), macOS (x86_64 and AARCH64), Windows (x86_64), Android (AArch64), and iOS (AArch64). The control flow of the application was effectively diverted *on each* tested system.

Finally, to answer **RQ6**, we test whether shadow stacks—a modern mitigation offering backward-edge control flow integrity—are vulnerable as well. We select several different shadow stack implementations, including six schemes presented by Burow et al. in their SoK on shadow stacks[7] [13], LLVM's ShadowCallStack implementation for AArch64 on Android [48], and Intel CET on Windows[8]. All the tested implementation failed to mitigate the attack.

Overall, our results confirm that the state-of-the-art unwinder implementations and backward-edge protection mechanisms are insufficient to mitigate the attacks proposed in this paper. The only exception we could find is LLVM SafeStack [47], which adopts a split (rather than shadow) stack design to enforce return address isolation (rather than replication). This design can effectively hinder CHOP exploits such as the one presented in Listing 5. Nonetheless, the split stack design cannot detect attacks and preclude return address corruption by means of more sophisticated corruption primitives [33]. Moreover, SafeStack [47] is incompatible with dynamic libraries, which means it cannot mitigate CHOP exploits based on dynamic library vulnerabilities such as our LibRaw exploit. Another avenue to mitigate CHOP exploits is to rely on forward-edge protection mechanisms. However, this strategy can only mitigate direct forward-edge control-flow hijacks through the "golden handler" (Listing 3) or vtable hijacking (Section V-B), but leave backward-edge hijacks (Section V-B), data-only attacks (Section V-C), and arbitrary writes in general a viable option.

VIII. Mitigations

As an immediate mitigation against CHOP-style attacks, we recommend extending the widely deployed stack canary checks to the unwinding path. This strategy can detect stack corruption at throw time and prevent the execution of unwinding logic over attacker-controlled data. Given that unwinding should only occur under *exceptional* circumstances, we estimate the performance overhead of such mitigation to be low.

However, to be fully effective, this strategy would require every function invoking `__cxa_throw` to be protected by a stack canary. Similarly, to address exception handler based pivot-to-ROP attacks, not only throwing functions but also other functions deploying exception handlers should be protected by stack canaries. These changes require updates to production compiler toolchains.

More advanced hardening strategies assume an attacker can bypass stack canaries (i.e., either by leaking the canary value or by means of noncontiguous stack-based corruption primitives). To limit the attacker in this scenario, an option is to rely on a *context-aware* unwinding mechanism. That is, the unwinder can store information about legitimately registered exception handlers in a given program context and initiate the unwinding process only if an exception handler for the thrown exception is present. Alternatively, in situations where shadow stacks are available, another option is to modify the unwinder to operate on the shadow stack data, rather than the original one. However, both of these approaches collide with the Itanium C++ ABI and implementing them would likely require nontrivial updates to the unwinding logic.

Another avenue for reducing the attack surface is to limit the number of the exploitation-friendly catch-all and other overly permissive catch handlers. Having every exception handled by a specific handler and reducing the number of handlers available in widely used libraries (such as `libstdc++`) would greatly reduce the capabilities of an attacker. While some popular C++ style guides (e.g., [35], [46]) disallow the use of exceptions, we however note that the use of third-party libraries using exceptions exposes a program to CHOP attacks.

Finally, randomization-based defenses can minimize the risk posed by CHOP-style attacks. These defenses are based on the concept of automated software diversity, which raises the bar for attackers by randomizing certain programs' aspects [41]. For example, *function reordering*—where the order of functions in binaries is randomized—and *stack layout randomization* belong to this category, and they can protect against CHOP-style attacks. However, fine-grained code randomization techniques rarely support C++ exceptions [61]. This happens because a defense can violate a key assumption of C++ exceptions, or because a research prototype fails to update the metadata used during exception handling. Finally, note that randomization-based defenses (especially coarser-grained versions such as ASLR) cannot stop more sophisticated CHOP-style attack variants, e.g., those based on data-oriented programming [38] or position-independent code reuse [34].

IX. Discussion

Recovery & Post Exploitation. In this paper, we focused on confusion primitives as attack entry vectors, and gadgets provided by cleanup and exception handlers as means for exploitation. However, for practical end-to-end attack, recovery to the normal program control flow after executing the

[7]We tested *reg*, *parr*, *mem_scheme*, *seg*, *con* and *mpx*, while *mpk* did not compile.

[8]Note that we exclude tests for ShadowCallStack on x86, which was deprecated in LLVM 7, as well as CET on Linux, which lacks readily available kernel and runtime support.

malicious gadgets may be desired. We argue that, for most cases, this is straightforward, for instance by using a golden handler to execute the program from start, or by restoring the program state via a ROP chain after successful pivoting. However, in other cases recovery may be more complex and require additional alignment of stack frame sizes, careful selection of used registers, or manual stitching of gadgets to fulfill the constraints presented by a specific vulnerability. While we acknowledge the difficulty of the problem, we leave the exploration of "exception handler feng-shui" and similar techniques as future work.

Using taint analysis for exploitation. The taint analysis allows analysts to assess the feasibility of exploiting a CHOP-style vulnerability, rather than providing fully automated exploit generation capabilities. Given a function vulnerable to stack based overflow, a human analyst can use the analysis's detected reachable exception types identify useful gadgets, and determine attacker-controlled data to guide exploitation.

Applicability to other languages. We focus on unwinding and exploitability of C++ programs due to their popularity and their history of memory corruption vulnerabilities. However, we want to stress that exception handling, and especially unwinding, are also widely used primitives in other programming languages, often directly adopting the unwinding process from the Itanium C++ ABI (e.g, Objective-C and Rust [14]). Assuming an attacker can corrupt the saved return pointer in those cases (e.g., using cross-language attacks [51]), the unwinder can be similarly confused as in the C++ case.

Threats to validity. Our attack surface analysis is based on a dataset obtained from the Debian package index. While we conduct our analysis at the binary level to obtain the same view on the program as the unwinder, relying on freely available software imposes some restrictions for the dataset. More specifically, much widely used C++ software is not free (e.g., game engines or complex proprietary desktop applications). However, we argue that the attack surface for CHOP is likely comparable to that of free software, as the unwinding techniques used and the potential for memory corruptions are the same in both cases.

X. Related Work

Windows SEH Exploitation. Structured Exception Handling has been a prime target for exploitation on Microsoft Windows in the past. Attacks exploit Windows' unique exception handling approach, featuring dynamic exception handling metadata stored on the stack [44]. Microsoft has implemented different defenses to counter this exploitation vector, namely `SAFESEH` [54] (which operates at the compiler level) and `SEHOP` [53] (which is implemented in the runtime). While `SAFESEH` restricts the set of valid exception handlers, `SEHOP` ensures that the entire exception handler list is a valid linked list. Both mitigations focus on enforcing the integrity of exception handling metadata, but do not secure the unwinding process itself, thereby failing to mitigate the exploitation strategies we showcase.

The crucial difference between SEH-based exploitation and CHOP is that the former relies on corrupting exception handling metadata—stored in read-only sections in ELF binaries—while the latter confuses the unwinder by tampering with the data it operates on. This makes our techniques more general since they apply to all the unwinders operating on the call stack, even on Windows systems equipped with protections against SEH hijacking.

Malicious Unwind Metadata. Attacks abusing unwind metadata have previously been studied by Oakley and Bratus [56]. Their work considered the scenario of a trojanized binary containing malicious unwind metadata in the form of manipulated DWARF instructions. This allows "hidden" malicious code to exist in otherwise benign-looking binaries. While these techniques are relevant for malware obfuscation, manipulating unwind metadata—normally stored in read-only sections of the binary—is not of interest for memory corruption exploits. In contrast, CHOP focuses on abusing the benign unwinding information already present in the binary in the presence of memory corruption vulnerabilities.

Obfuscation and Exception Handling. Quite recently, new techniques have been proposed to obfuscate the execution of code using exception handling mechanisms [26], [73]. In a nutshell, these techniques leverage Vectored Exception Handlers (VEH) as a way to execute single instructions already present in shared objects. As a result, the entire control flow of the application is encoded in these handlers, complicating static and dynamic program analysis. Therefore, despite the name similarity, these techniques are not related to CHOP.

Unwind Metadata Use in Reverse Engineering. Unwind metadata can be readily used for reverse-engineering tasks. Priyadarshan et al. [60] show how EH metadata can help identify function boundaries, but also how fine-grained code randomization techniques can be defeated if the attacker is able to leak EH metadata. Pang et al. [59] further explore this subject and find that combining recursive disassembly with exception handling information provides nearly perfect function identification.

XI. Conclusion

In this paper, we explored the overlooked attack surface in the stack unwinding logic invoked during exception handling. We demonstrated the ability to lure the unwinder into bypassing backward-edge protection techniques (i.e., stack canaries and shadow stacks) and derive a variety of Catch Handler Oriented Programming (CHOP) attacks.

To assess the impact of these attacks, we analyzed a dataset of popular C++ programs. Our results show that binaries employing exception handling are likely to provide both primitives for confusing the unwinding logic and usable gadgets for exploitation. Overall, we conclude that CHOP attacks provide concrete evidence that additional sanity checks are needed in production unwinding software.

Availability

To enable reproducibility and foster further research on this topic, we publicly release every artifact produced during this research, including our dataset of Debian binaries, the source code of our analyses, and the PoC binaries at https://github.com/chop-project/chop.

Coordinated Disclosure

We disclosed the issues described in this paper to Apple, Google, Intel, ARM, LLVM and Microsoft. Google labeled our findings as "Not Security Bulletin Class" and rather as issues for the feature team. LLVM, Apple, Microsoft and Intel acknowledged our findings. Moreover, LLVM plans to add mitigations as part of feature work. Intel classified our findings as security-critical and issued CVE-2022-40196 with mitigations underway [39].

We further reached out to the gcc and libunwind developers inquiring for a security contact. While libunwind did not respond, the gcc developers preferred to publicly discuss fixes after the coordinated disclosure period.

Acknowledgments

We thank the anonymous reviewers for their feedback. We'd also like to thank Gregor Kopf for finding and pointing out CVE-2009-4009. This work was supported by EKZ through the AVR "Memo" project and by NWO through projects "TROPICS", "Theseus", and "INTERSECT".

References

[1] longjmp(3) linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man3/longjmp.3p.html

[2] setjmp(3) linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man3/setjmp.3.html

[3] Aleph One, "Smashing the Stack for Fun and Profit," in *Phrack Magazine*, 1996.

[4] Apple. Smart Card Services - Product Security. [Online]. Available: https://smartcardservices.github.io/security/

[5] Apple. (2011) libunwind. [Online]. Available: https://opensource.apple.com/source/libunwind/libunwind-35.3/include/mach-o/compact_unwind_encoding.h

[6] ARM, *Exception Handling ABI for the Arm Architecture*, 2022.

[7] T. P. Baker and G. A. Riccardi, "Implementing Ada Exceptions," *IEEE Software*, 1986.

[8] A. Beingessner. (2021) The Apple Compact Unwinding Format: Documented and Explained. [Online]. Available: https://faultlore.com/blah/compact-unwinding/

[9] B. Bierbaumer, J. Kirsch, T. Kittel, A. Francillon, and A. Zarras, "Smashing the Stack Protector for Fun and Profit," in *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2018.

[10] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *IEEE Symposium on Security and Privacy*, 2014.

[11] E. Bosman and H. Bos, "Framing Signals—A Return to Portable Shellcode," in *IEEE Symposium on Security and Privacy*. IEEE, 2014.

[12] S. Bratus, M. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit Programming - From Buffer Overflows to Weird Machines and Theory of Computation," *USENIX: login*, 2011.

[13] N. Burow, X. Zhang, and M. Payer, "SoK: Shining Light on Shadow Stacks," in *IEEE Symposium on Security and Privacy*, 2019.

[14] A. Burtsev, D. Appel, D. Detweiler, T. Huang, Z. Li, V. Narayanan, and G. Zellweger, "Isolation in Rust: What is Missing?" in *Workshop on Programming Languages and Operating Systems*, 2021.

[15] S. Cass, "Top Programming Languages 2021," in *IEEE Spectrum*, 2021.

[16] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[17] T. Chiueh and F. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," in *IEEE Conference on Distributed Computing Systems*, 2001.

[18] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ ABI. [Online]. Available: https://itanium-cxx-abi.github.io/cxx-abi/abi.html

[19] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[20] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." in *USENIX Security Symposium*, 1998.

[21] F. Cristian, "Exception Handling and Software Fault Tolerance," in *Reliable Computer Systems*, 1985.

[22] ——, "Exception handling," in *Dependability of Resilient Computers*, 1989.

[23] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[24] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[25] T. De Raadt. (2005) Exploit Mitigation Techniques. [Online]. Available: http://www.openbsd.org/papers/ven05-deraadt/index.html

[26] B. Demirkapi. Abusing Exceptions for Code Execution, Part 1. [Online]. Available: https://billdemirkapi.me/exception-oriented-programming-abusing-exceptions-for-code-execution-part-1/

[27] T. Dullien, "Exploitation and State Machines," *Infiltrate*, 2011.

[28] ——, "Weird Machines, Exploitability, and Provable Unexploitability," *IEEE Transactions on Emerging Topics in Computing*, 2017.

[29] DWARF Debugging Information Format Committee, "DWARF Debugging Information Format version 5," 2017.

[30] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking Branch Predictors to Bypass ASLR," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[31] GCC Authors. GNU C Compiler Collection. [Online]. Available: https://gcc.gnu.org/

[32] S. Geddis and L. Rousseau. Smart Card Services. [Online]. Available: https://smartcardservices.github.io/

[33] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining Information hiding (and What to Do about It)," in *USENIX Security Symposium*, 2016.

[34] E. Göktaş, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida, "Position-Independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure," in *IEEE European Symposium on Security and Privacy*, 2018.

[35] Google. Google c++ style guide. [Online]. Available: https://google.github.io/styleguide/cppguide.html

[36] ——. OSS-Fuzz. [Online]. Available: https://google.github.io/oss-fuzz/

[37] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[38] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE Symposium on Security and Privacy*, 2016.

[39] Intel. INTEL-SA-00773. [Online]. Available: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00773.html

[40] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block Oriented Programming: Automating Data-Only Attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[41] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *IEEE Symposium on Security and Privacy*.

[42] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *IEEE/ACM International Symposium on Code Generation and Optimization (GCO)*, 2004.

[43] LibRaw LLC. LibRaw. [Online]. Available: https://www.libraw.org/

[44] D. Litchfield, “Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server,” Black Hat Asia, 2003.

[45] LLVM. Libunwind. [Online]. Available: https://bcain-llvm.readthedocs.io/projects/libunwind/

[46] LLVM. LLVM Coding Standards. [Online]. Available: https://llvm.org/docs/CodingStandards.html

[47] LLVM. SafeStack. [Online]. Available: https://clang.llvm.org/docs/SafeStack.html

[48] ——. ShadowCallStack. [Online]. Available: https://clang.llvm.org/docs/ShadowCallStack.html

[49] H. Lu, M. Matz, M. Girkar, J. Hubicka, A. Jaeger, and M. Mitchell, “System V application binary interface,” *AMD64 Architecture Processor Supplement*, 2018.

[50] D. C. Luckham and W. Polak, “ADA Exceptions: Specification and Proof Techniques.” Tech. Rep. STAN-CS-80-789, 1980.

[51] S. Mergendahl, N. Burow, and H. Okhravi, “Cross-Language Attacks,” in *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[52] Microsoft. /GS (Buffer Security Check). [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/reference/gs-buffer-security-check

[53] Microsoft. How to enable Strucatured Exception Handling Overwrite Protection (SEHOP) in Windows operating systems). [Online]. Available: https://support.microsoft.com/en-us/topic/how-to-enable-structured-exception-handling-overwrite-protection-sehop-in-windows-operating-systems-8d4595f7-827f-72ee-8c34-fa8e0fe7b915

[54] ——. /SAFESEH (Image has Safe Exception Handlers). [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/reference/safeseh-image-has-safe-exception-handlers?view=msvc-170

[55] NonGNU-Libunwind Authors. Libunwind. [Online]. Available: https://www.nongnu.org/libunwind/

[56] J. Oakley, “Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code,” in *USENIX Workshop on Offensive Technologies (WOOT)*, 2011.

[57] D. Oepen and F. Morgner. Virtual Smart Card. [Online]. Available: http://frankmorgner.github.io/vsmartcard/

[58] P.-L. Palant and J. Jelen. virt_cacard. [Online]. Available: https://github.com/PL4typus/virt_cacard

[59] C. Pang, R. Yu, D. Xu, E. Koskinen, G. Portokalidis, and J. Xu, “Towards Optimal Use of Exception Handling Information for Function Detection,” in *Conference on Dependable Systems and Networks (DSN)*, 2021.

[60] S. Priyadarshan, H. Nguyen, and R. Sekar, “On the Impact of Exception Handling Compatibility on Binary Instrumentation,” in *ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2020.

[61] ——, “Practical fine-grained binary code randomization,” in *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[62] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications,” in *IEEE Symposium on Security and Privacy*, 2015.

[63] F. J. Serna, “The Info Leak Era on Software Exploitation,” *Black Hat USA*, 2012.

[64] H. Shacham, E. Buchanan, R. Roemer, and S. Savage, “Return-Oriented Programming: Exploits Without Code Injection,” *Black Hat USA Briefings*, 2008.

[65] Solar Designer, “Getting around non-executable stack (and fix),” *Bugtraq*, 1997.

[66] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal War in Memory,” in *IEEE Symposium on Security and Privacy*, 2013.

[67] Technical Standardization Committee on AV & IT Storage Systems and Equipment, “Exchangeable image file format for digital still cameras: Exif Version 2.2,” Japan Electronics and Information Technology Industries Association, Tech. Rep. JEITA CP-3451, April 2002.

[68] The Clang Team, “Clang documentation.” [Online]. Available: https://clang.llvm.org/docs/

[69] The MITRE Corporation. 2021 CWE Top 25 Most Dangerous Software Weaknesses. [Online]. Available: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

[70] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, “The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later,” in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[71] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, “Memory Errors: The Past, the Present, and the Future,” in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2012.

[72] Vector 35. Binary Ninja. [Online]. Available: https://binary.ninja/

[73] x86matthew. WindowsNoExec - Abusing Existing Instructions to Executing Arbitrary Code without Allocating Executable Memory. [Online]. Available: https://www.x86matthew.com/view_post?id=windows_no_exec

[74] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, “Architecture support for defending against buffer overflow attacks,” Tech. Rep. UILU-ENG-02-2205, CRHC-02-05, 2002.

[75] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, “SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux Kernel,” 2022.

Appendix

A. Stack layouts

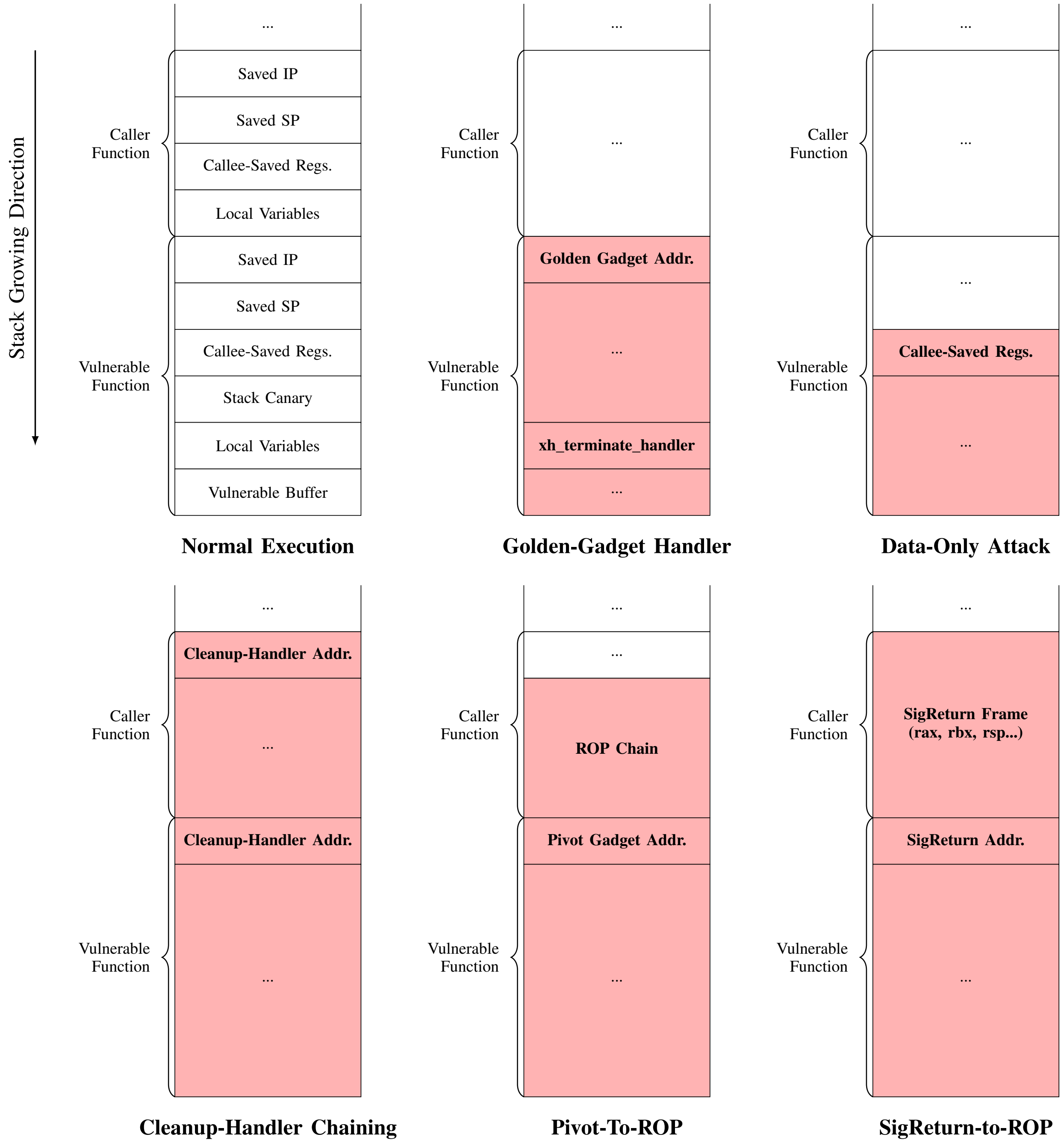

Fig. 6: Stack layouts for the different attacks presented in Section V-C. For each case, the attacker-controlled region is indicated in red. The positions of overflowed data corresponds to the positions indicated in the Normal Execution case.

B. Taint analysis algorithm

In this section we present a simplified version of the taint analysis algorithm. Algorithm 1 depicts the algorithm's main function (lines **1-12**) and the taint initialization function (lines **14-19**). Taint analysis receives as input the binary and a bitwise mask (i.e., `taint_model`) representing the user assumed control model (i.e., which callee-saved registers and stack relative slots are under the control of the user).

Algorithm 1: Taint analysis algorithm

```
   Input: binary, taint_model
   Data: sinks
   /* taint analysis main function                          */
1  Function Main:
2      view ← get_SSA_view(binary)
3      Handlers ← exception_handlers(binary)
4      forall handler ∈ Handlers do
           /* mount function at landing pad address
              */
5          func ← view.add_function(handler.lp)
           /* get SSA variables representing the
              function parameters                            */
6          Params ← func.get_params()
           /* taint parameters specified by model
              */
7          init_taint (Params, taint_model)
8
           /* loop over instructions, propagate
              taint and report tainted sinks                 */
9          forall expr ∈ func.instrs do
10             eval(expr)
       /* save sinks to database                             */
11     save_sinks(sinks)
12     return
13
   /* loop over function parameters and taint
      parameters based on assumed control model
      (i.e., what callee-saved registers and
      stack slots we control)                                */
14 Function init_taint(Params, model):
15     forall ssa_var ∈ Params do
16         if fits_model(model, ssa_var) then
               /* get taint color: ssa_va is either
                  a callee-saved register or a
                  stack slot                                 */
17             color ← taint_color(ssa_var)
18             set_taint (ssa_var, color)
19     return
```

The algorithm uses Binary Ninja to obtain a view over the binary's High-Level SSA IL (line **2**) after which it parses the (.gcc_except_table) section and extracts all catch/cleanup handlers present in the binary (line **3**). Then, the analysis starts evaluating every exception handler (lines **4-10**) for the presence of gadgets (i.e., taint sinks). For each exception handler we mount a function (line **5**) starting at the landing pad address of the handler. The landing pad is the address of the first instruction where the unwinder transfers control to the handler.

Algorithm 2: Taint evaluation function

```
   Data: sinks
   /* general eval function                                  */
20 Function eval(expr):
21     switch expr.operation do
22         case SSA_VAR do
23             return get_taint (expr)
24         case VAR_INIT ∨ ASSIGN do
               /* get taint of source                        */
25             taint ← eval(expr.Src())
               /* propagate taint to destination             */
26             propagate_taint (expr.Dst(), taint)
27             return taint
28         case MEM_ASSIGN do
29             lh_of_assign ← 1
               /* destination is a memory
                  dereference                                */
30             dst_taint ← eval(expr.Dst())
31             lh_of_assign ← 0
32             src_taint ← eval(expr.Src())
               /* if taint on src or dst this is a
                  w-what, w-where or w-what-where
                  sink                                       */
33             if dst_taint ∨ src_taint then
34                 sinks.add_sink(expr)
               /* propagate src taint to the
                  dereferenced location                      */
35             propagate_taint (expr.Dst(),
                src_taint)
36             return src_taint
37         case PTR_DEREF ∨ ARRAY_DEREF
            do
38             taint ← 0
39             if lh_of_assign ≠ 1 then
                   /* get the taint of dereferenced
                      location if we are not in the
                      left-hand side of a mem-assign
                      */
40                 taint ← get_expr_taint(expr)
               /* get pointer taint and combine
                  with taint of dereferenced
                  location                                   */
41             taint ← eval(expr.Src()) | taint
42             return taint
43         case CALL_INST do
44             return eval_call (expr)
           /* Rest of the evaluated operations               */
45         ...
46     return 0
```

Binary Ninja will automatically set as function parameters all SSA variables that are read before being written by the function. These variables either point to CPU registers or to stack relative locations. The function parameters and the bitwise mask representing the taint model are forwarded to the initialization function (line **7**) which sets taint on all parameters that fit the user supplied control model. Each parameter will be assigned a different bitwise mask representing its taint color (lines **17-18**). The taint color depends on which callee-saved register the parameter refers to, or it depends on whether the parameter points to a stack relative location under the control of the user. After this initialization step, the algorithm loops over each instruction in the handler function propagating taint and reporting any sinks discovered in the process (lines **9-10**).

Taint Propagation. Algorithm 2 shows the core part of our taint analysis. This function, depending on the instruction type received as parameter, evaluates and propagates the taint, and also reports any matched taint sinks. For brevity, we included only a portion of the existing propagation rules. Binary Ninja represents both instructions and (SSA) variables by means of expressions which themselves might be comprised of multiple nested sub-expressions. For this reason the evaluation function is recursive. Taint propagation happens on specific assign and variable initialization instructions (cases at line **24** and **28**) or on `phi` instructions (not included in the snippet). During propagation we evaluate the taint on source sub-expressions and propagate the taint to the destination part of an expression (lines **25-26** and **32,35**). In most cases the destination is a SSA variable, case in which propagation simply adds the variable to the list of tainted variables. For memory assigns the destination will be a pointer or array dereference expression. In this case we taint the entire expression representing the dereferenced location. If taint analysis encounters the tainted dereference as part of a source sub-expression (lines **37-42**) it will combine the taint of the dereferenced location (line **40**) with the taint on the pointer (line **41**), akin to traditional pointer tainting.

Algorithm 3: Call instruction evaluation function

```
   Data: $sinks$, $FREE\_FUNCS$
   /* call instruction eval function */
47 Function eval_call($expr$):
48     $target \leftarrow expr.Target()$
49     $Params \leftarrow expr.Params()$
50     $param\_taint \leftarrow 0$
       /* combine parameter taint */
51     forall $p \in Params$ do
52         $param\_taint \leftarrow param\_taint \mid \texttt{eval}(p)$
53     $target\_taint \leftarrow \texttt{eval}(target)$
       /* if target is tainted we have an icall sink */
54     if $target\_taint \neq 0$ then
55         $sinks.add\_sink(expr)$
56     else if $param\_taint \neq 0$ then
           /* if target is a free function with tainted parameters then we have a free sink */
57         if $target.name \in FREE\_FUNCS$ then
58             $sinks.add\_sink(expr)$
       /* taint of the call is the combined taint of all of its parameters */
59     return $param\_taint$
```

Sinks. Memory assignment instructions are also checked for arbitrary write sinks. For these instructions, the `eval` function evaluates taint on the source and destination and reports the appropriate gadget depending on the characteristics of the taint: **write_what_where** ($src\_taint \neq 0$, $dst\_taint \neq 0$), **write_what** ($src\_taint \neq 0$) or **write_where** ($dst\_taint \neq 0$). Algorithm 3 depicts the `eval_call` function, which evaluates taint on call instructions. If `eval_call` infers that the target of the call is tainted then the call is reported as a **control-flow hijacking** (lines **54-55**). Otherwise, if the parameters of the call are tainted and the target is a constant pointer to a `free` function then the call is reported as an **arbitrary free** gadget (lines **56-58**).

C. Mimicking CHOP Attacks

```
#include <iostream>
using namespace std;

void catcher() {
  try {
    throw 1;
  }
  catch (...) {
    cout << "win" << endl;
    exit(0);
  }
}

void vuln() { ④
  void* data[1];
  data[SAVED_RETURN_OFFSET] = (char*) catcher + ((
    size_t) TRY_OFFSET);
  throw 1337;
}

int main() {
  try {
    vuln();
  }
  catch (...) {
    cout << "catch" << endl;
  }
}
```

Listing 5: Test code used to evaluate mitigation efficacy.

We show our program for testing mitigations in Listing 5. The preprocessor macros `SAVED_RETURN_OFFSET` and `TRY_OFFSET` are chosen based on the implementation's stack layout and offset of the try block. On ARM architectures, we further insert an additional call into `vuln` to force the link register to be spilled onto the stack.

This program effectively mimics a CHOP attack. The crucial part of the code is the function `vuln` ④, since it effectively mimics a CHOP exploit: this function overwrites its own saved return address with an address pointing inside the try block of function `catcher` and then throws an exception. On systems where the unwinding library is vulnerable to CHOP exploits, this will divert the execution to the exception handler of `catcher`.